On the timed automata-based verification of Ravenscar systems

Iulian Ober[1] and Nicolas Halbwachs[2]

[1] Université de Toulouse - IRIT
118 Route de Narbonne, 31062 Toulouse, France
`iulian.ober@irit.fr`
[2] CNRS - VERIMAG
2, av. de Vignate, 38610 Gières, France
`Nicolas.Halbwachs@imag.fr`

Abstract. The Ravenscar profile for Ada enforces several restrictions on the usage of general-purpose tasking constructs, thereby facilitating most analysis tasks and in particular functional and timing verification using model checking. This paper presents an experiment in translating the Ravenscar fragment of Ada into the input language of a timed model checker (IF [7, 8]), discusses the difficulties and proposes solutions for most constructs supported by the profile. The technique is evaluated in a small case study issued from a space application, on which we present verification results and conclusions.

1 Introduction

This paper discusses an experiment in applying model checking techniques to the verification of functional and non-functional (timing) aspects of Ada systems complying with the Ravenscar profile [10, 1].

We targeted the IF model checker [7, 8] for several reasons including its ability to handle complex structured data, dynamic object allocation (necessary to simulate the procedural control flow of Ada), both timed and non-timed execution aspects, and last but not least for the automatic abstraction features of the IF tool which help to cope efficiently with large specifications.

Ravenscar is a standardized set of restrictions for the Ada language and runtimes, set forward in order to facilitate the verification of concurrent real-time programs and to make their implementation more reliable and efficient. The incentive to apply model checking to Ravenscar systems was the fact that the profile is used as the runtime and semantic baseline by several work tracks [17, 5, 22] within the IST ASSERT European project[1]. ASSERT aims at developing novel systems engineering methods for distributed and embedded real time systems in the aerospace domain, based on formal model-centric techniques.

[1] This work was partially supported by ASSERT, an Integrated Project of the 6th Framework Programme IST of the EU, see http://www.assert-project.net.

The paper is structured as follows. Section 2 is a brief overview of the Ravenscar Ada profile. Section 3 presents the IF language and tools. Section 4 is the main part of the paper, describing the mapping of the main Ada concepts to IF, and discussing how the Ravenscar profile restrictions may help. Then, in Section 5 we present experimental results on a small case study, before concluding.

2 The Ravenscar Ada profile

The Ada language exhibits a rich set of constructs for programming concurrent and time-aware systems. However, as analyzed in [10, 1], the total freedom granted by the tasking and inter-task communication model of the language comes at a high cost. Firstly, this freedom exposes the systems to various run-time problems related to schedulability (like unbounded blocking times or priority inversion) and concurrency control (like deadlocks). Secondly, it makes it impossible to apply most currently available analysis techniques to verify the resulting system: known schedulability analyses [12] like rate monotonic analysis [20] or response time analysis [19] do not work on such general models, and the application of more general purpose techniques like model checking [13] is hindered.

Several factors cause the aforementioned problems, among which: tasks may be created dynamically (either explicitly or implicitly), their activation patterns are arbitrary (a task may be suspended and activated by arbitrary `delay` statements, by rendez-vous, etc.), they may communicate in various ways (through rendez-vous, through shared protected objects, etc.).

The Ravenscar profile makes a number of restrictions on the tasking and synchronisation constructs that may be used, and additionally it imposes a scheduling policy – namely *fixed priority* preemptive scheduling with *priority ceiling protocol* (for a description, see for example [12]) – thus aiming to render the resulting systems analysable and to guarantee by construction certain properties such as absence of deadlocks and mutual exclusive access to shared resources. The restrictions, defined in [1] and motivated in [10], are the following:

- The task set is static and flat, i.e. there is no dynamic task creation, and all tasks are created at package level and depend directly on the environment task. Tasks are also not permitted to terminate or abort.
- Interrupt handlers are only attached statically to interrupts.
- Task rendez-vous is forbidden, only protected objects can be used for inter-task communication and synchronization. Moreover, the set of protected objects is static as they are only allowed to be created at package level. Protected objects are also subject to further restrictions: they may have *at most one protected entry* (along with any number of procedures and functions), the maximum queue length at the entry is *one* (otherwise, an exception is raised), the entry *barrier* must be a simple Boolean variable, and *requeue* is not allowed.
- All delays must be absolute (that is, only `delay until` statements are legal).

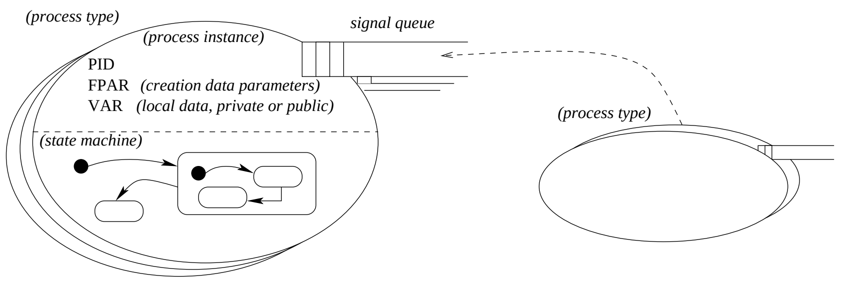


Fig. 1. Constituents of an IF model.

To achieve analysability of the resulting systems, all tasks must be structured such that they are either periodic or sporadic (i.e., event driven with a minimum inter-arrival time enforced). This restriction is not formally imposed by the Ravenscar profile since it can be verified neither statically nor at run-time. However, [10] contains coding patterns for these two types of tasks.

To end this section, we note that there are currently two runtimes which comply with the Ravenscar profile requirements: the commercial ObjectAda Real-Time RAVEN from Aonix [4] and the open source Open Ravenscar Real-Time Kernel ORK [14].

3 The IF model checker

The validation approach proposed in this work is based on the formal model of communicating extended timed automata, as it is embodied in the IF language and the execution and validation environment built around this language [7, 8]. This section provides a brief overview of IF, necessary for understanding the rest of the paper.

The IF language is dedicated to modeling *distributed systems* that can manipulate *complex data*, may involve *dynamic process creation* and *real time constraints*. The language constructs allow to represent the behavior of a system at an arbitrary level of abstraction, ranging from very abstract descriptions involving lots of non-determinism and un-interpreted actions, to very concrete descriptions involving concrete data manipulation, algorithmic structures, etc. In particular, the richness of the language allows to describe the semantics of higher level formalisms, like UML [16] or SDL [18], and has been used as a format for inter-connecting modelling and validation tools.

The main purpose of IF models is validation by formal techniques (simulation, property verification), which explains why some features of the language presented below, such as default transition atomicity, could be considered "unrealistic" from an implementation perspective.

Communicating extended timed automata. An IF system is composed of a set of communicating *processes* that run in parallel (see figure 1). Processes are instances of *process types*, they have an identity (PID), may own data variables and their behavior is defined by a *state machine*. The state machine may be hierarchical (i.e., making use of composite states) and the effect of transitions is described by usual structured imperative statements.

Data variables are statically typed, and may contain either simple scalars (boolean, integer, real) or structured data (arrays, records, objects).

Processes inter-communicate by sending *asynchronous signals*, which are stored in per-process signal *queues* until the destination process is ready to handle them. Additionally, processes may communicate via *shared variables*, which are public variables exported by some process and that every other process can read/write. Parallel processes are composed asynchronously (i.e., they progress independently from each other). The model also allows *dynamic creation* of processes, which is an essential feature for modeling object-based systems or procedural control flow, as will be shown later in this paper.

The link between system execution and time progress may be described in a precise manner, and thus offers support for modeling real time constraints. For this, IF uses the constructs of *timed automata with urgency* [6]: there are special variables called *clocks* which measure time progress. All clocks progress at the same rate during a system run, and they differ only by the moments when they are started; a dedicated statement, *set x:= 0*, is used to (re)start a clock *x*. Comparisons between a clock and an integer value may be used in transition guards. A special attribute of each transition, called *urgency*, specifies if time may progress when the transition is enabled.

Dynamic priorities. On top of the above model, *priority rules* allow for specifying dynamic priorities as partial orders between processes. The theoretical foundation of this framework is given in [3].

A priority rule has the following form:

```
p1 < p2 if state_condition(p1,p2)
```

where `state_condition(p1,p2)` is a boolean expression with free PID variables `p1` and `p2`, which can be interpreted in the context of a given system state. The semantics of the rule is the following: given a system state, for any pair of processes `P1` and `P2` which have enabled transitions in that state, if the (closed) formula `state_condition(P1,P2)` evaluates to true then the transitions of `P1` are not allowed to execute (i.e., `P2` has priority over `P1`).

It is shown in [3] how this kind of rules may be used to model different scheduling policies, including fixed priority scheduling, Earliest Deadline First (EDF) and others.

Property specification with observers. Behavioral properties of IF models may be expressed using *observer automata*. These are special processes that monitor the changes in a system's *state* (variable values, contents of queues, etc.) and the *events* occurring during the execution of transitions (inputs, outputs, creation and destruction of processes, etc.). To express desired *safety* properties of a system, some of the observer states are labeled as *error* states: if a system execution leads to such a state then the property represented by the observer was violated. This allows to express arbitrary *safety properties*.

Analysis techniques: the IF toolbox. The IF toolbox [7, 8] is the validation environment built around the formalism presented before. It is composed of three categories of tools. **Behavioral tools** are used for simulation, verification of properties, automatic test generation, state space minimization and comparison. The tools implement techniques such as *partial order reductions* and *symbolic simulation* of time, and thus present a good level of scalability. **Static analysis tools** provide source-level optimizations (*data flow analysis* such as dead variable reduction, *slicing*, etc.) that help reducing further the state space of the models. **Front-ends and exporting tools** which provide coupling to higher level languages (UML, SDL) and to other verification tools.

4 The mapping of Ravenscar Ada to IF

In this section we describe the principles of the mapping of Ada programming constructs to IF model elements. In IF the only first class language citizen is the process: it is used for encapsulating data and behavior, it is the only one that can be referenced (by PID), can be dynamically created and killed. Therefore, most of the constructs of Ada like packages, tasks, procedures, protected objects and referenced data will be encoded using processes.

Note that although this encoding significantly increases the number of processes employed to "simulate" a Ravenscar system, it does not add to the combinatorial complexity of system behavior, as most of the processes are just passively encapsulating data or waiting for some event (see §4.2). In general, the number of "active" processes in a given state is equal to the number of active threads in the corresponding configuration of the Ada system.

4.1 Packages, data and statements

Ada packages are static containers for various types of content: data variables, tasks, protected objects, procedures, types and other packages. Some of the content has an actual runtime existence (variables, tasks, objects) while in the case of procedures, types and nested packages, the encapsulating package acts only as a static namespace.

In IF, a package is mapped to a process which only contains variables corresponding to the Ada data, task or object variables. The static namespace function of a package cannot be fulfilled directly by an IF process since nesting is not allowed, therefore we use a naming scheme for mapping qualified Ada entity names to flat IF process names. This kind of manipulation is common in all compilers which generate low-level object code from a high level language with complex scoping rules. Similarly, generic packages and instantiation are also handled using naming rules and code replication (code size optimization is not important for verification).

Scalar data is restricted to the types supported by IF: `boolean`, `integer` and `real`. This limitation is not very strong when the goal is formal verification of high-integrity systems, since the control flow of such systems is rarely affected by other types of data. Complex data types are constructed using the IF `array` and `record` constructors, or by encapsulation within dedicated processes (e.g., for constructing records with *variants*). Processes are also used to represent any entity which is handled *by reference*, like tasks and protected objects (the reference is then the PID).

The mapping of computation statements is defined as follows:

– The points of control before and after a statement are represented by IF *states*, and the statements are represented by *transitions* between states.
– Assignments and elementary operations have a direct counterpart in IF.
– Procedure and function calls and evaluation of expressions containing calls is done according to the simple principles described in §4.2.
– Control flow statements like alternatives and loops are encoded in the structure of the state / transition graph.

We note that several statements, like those involved in (bounded) rendez-vous, etc., are forbidden by the Ravenscar profile. Other statements, which are not forbidden (e.g., `delay until`), are explained below.

4.2 Procedural control flow and tasks

Procedures, functions and protected object entries (collectively referred to as subroutines in this section) are represented by processes which are dynamically created upon call and killed upon return of control. Thus, the runtime call stack of a task has a direct representation in IF as a linked list of processes. The processes hold the subroutine local variables and parameters, and realize the subroutine behavior by their automaton.

IF allows passing data parameters (`fpars`) at process creation which allows us to represent very directly the passing of `in` parameters. Along with these, a caller also passes as parameter its identity and the identity of the *task* on behalf of which the call is made. These references are used for returning the control, which is represented by the sending of a `return_<procedure name>` signal from the callee to the caller, just before the callee kills itself. The signal also carries the `out` parameters of the procedure, if any.

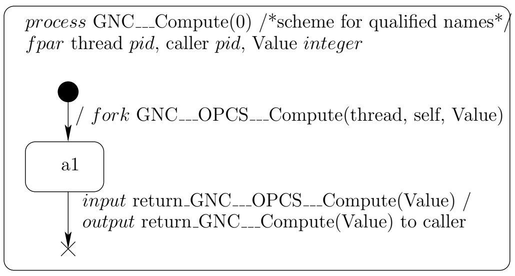
Figure 2 shows the mapping of a complete procedure, illustrating the mechanisms of call and return (as well as the implementation of actions by automata, the access to variables, etc.). For clarity, we have presented the IF process in a graphical form which is isomophic to the textual form actually used by the tool.

Finally, *tasks* are mapped to processes which encapsulate local task variables and realize the task behavior (body) by their automaton.

4.3 Protected objects

Protected objects are the synchronization mechanism used in Ravenscar Ada. They provide functions (which may only read but not modify object attributes) that can be executed concurrently, together with procedures and entries that are

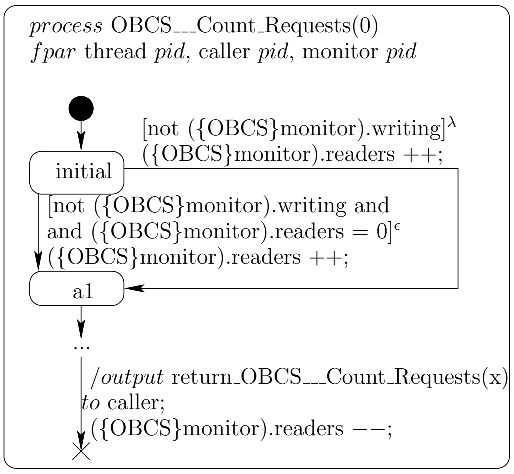
```
procedure Compute
 (Value : in out Natural) is
begin
  OPCS.Compute (Value);
end Compute;
```



process GNC___Compute(0) /*scheme for qualified names*/
fpar thread *pid*, caller *pid*, Value *integer*
/ *fork* GNC___OPCS___Compute(thread, self, Value)
a1
input return_GNC___OPCS___Compute(Value) /
output return_GNC___Compute(Value) to caller

Fig. 2. Mapping of procedural control flow.

```
protected body OBCS is
 ...
 function Count_Requests
   return Integer is
 begin
   ...
   return x;
 end Get_Request;
end OBCS;
```



process OBCS___Count_Requests(0)
fpar thread *pid*, caller *pid*, monitor *pid*
[not ({OBCS}monitor).writing]$^\lambda$
({OBCS}monitor).readers ++;
initial
[not ({OBCS}monitor).writing and
and ({OBCS}monitor).readers = 0]$^\epsilon$
({OBCS}monitor).readers ++;
a1
...
/*output* return_OBCS___Count_Requests(x)
to caller;
({OBCS}monitor).readers −−;

Fig. 3. Mapping of protected objects: functions.

executed in mutual exclusion from each other and from functions. This corresponds to the classical *readers-writers* problem and Ada runtimes solve it using lower level mutual exclusion mechanisms. In IF however, the solution is much facilitated by the fact that the language offers mutually exclusive and atomic *transitions* by default, and transitions are provided with *guard conditions* that may be readily used for conditional waiting.

A protected object is mapped to a process that encapsulated its data, together with two additional variables used for implementing the readers-writers protocol: a boolean `writing` and an integer `readers`. Functions, procedures and entries are, as mentioned before, mapped to processes that are created upon calling and they receive an additional parameter – `monitor` – pointing to owning the protected object. The readers-writers protocol implemented in our mapping is a variant of the classical solution that may be found in many textbooks (see for ex. [11]):

– Procedures and entries begin by waiting in an `initial` state until `readers=0 and not writing`. For entries, this condition is augmented with the barrier.

When this condition is true, it is followed atomically by `writing := true`. At the end of procedures/entries, `writing` is reset to false.

– Functions begin by waiting in an `initial` state until `not writing`. When this condition is true, they atomically increment `readers`. At the end of the function, `readers` is decremented.
  To preserve the generality of the solution, we also allow functions to stay in the `initial` state if at least one other function is executing in the object. Thus, the model includes both the behavior where functions eagerly begin execution regardless of waiting procedures/entries (possibly leading to the starvation of the latter), and the behavior where functions lazily wait giving the possibility to procedures/entries to start (possibly leading to the starvation of tasks executing the functions).
  At the level of IF (see Figure 3), this is achieved by declaring the transition from `initial` as having *lazy* urgency (see [6]). However, a reader function is not allowed to wait when the protected object is free of any access, i.e. when `readers = 0 and not writing`. To enforce this, a second (*eager*) transition is added.

The mapping of functions is illustrated in Figure 3 (for space reasons we do not include an illustration for procedures, described above). Note that the transition *guards* are represented inside square brackets, with the urgency specified as an exponent (λ for *lazy*, ϵ for *eager*). They are followed by a slash and the actions executed (atomically) by the transition. Also, note that the expression `({A}p).B` in IF (used for accessing variables from the `monitor` process) denotes the casting of an untyped PID `p` to the type of process `A` followed by access to variable `B` exported by `p`.

Since Ravenscar disallows more than one task waiting on an entry, we need not represent the waiting queues in the IF model. The waiting tasks are simply those whose calls are in the `initial` state. Note that one interesting use of IF may be to verify the satisfaction of the queue length restriction.

4.4 Time and delays

As mentioned before, in Ravenscar tasks may suspend themselves only using absolute delays. In practice, the coding patterns presented in §5 of [10] show that the absolute dates are always computed relatively to a “timestamp” (e.g., obtained with `Ada.Real_Time.Clock`). This kind of waiting falls within the expressive power of timed automata. In Figure 4 we show how the *Cyclic* task from [10] is mapped to IF using clocks (see the description of clocks in §3).

4.5 Scheduling policy and timing model

The Ravenscar profile fixes the scheduling policy to FIFO within priorities with priority ceiling locking. One can suppose that every task and protected object contains a `pragma Priority` directive, and that the assigned priority levels observe the ceiling rule.

```
task body Cyclic is
 Next_Period : Ada.Real_Time.Time;
 Period : constant
   Ada.Real_Time.Time_Span := ...;
begin
  Next_Period :=
    Ada.Real_Time.Clock + Period;
  loop
    delay until Next_Period;
    Next_Period := Next_Period
      + Period;
    ...
  end loop;
end Cyclic;
```

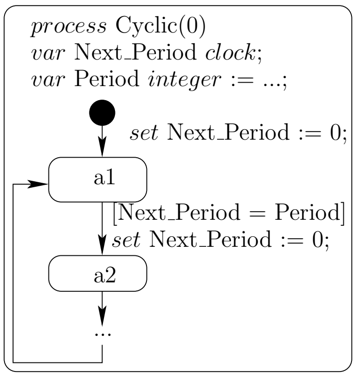


Fig. 4. Mapping of a cyclic task.

The dynamic priority framework of IF is more expressive than this policy, and therefore it allows a quite straightforward mapping for it. The mapping idea is that every process corresponding to a task has a `priority` attribute, which is dynamically updated to reflect the ceiling priority when entering/exiting subroutines of a protected object. Then, the policy is simply modeled by an IF dynamic priority rule equivalent to this:

```
x < y if x.priority < y.priority
```

(The actual IF rule, equivalent to this one, is slightly more complex because attribute access like `x.priority` cannot be directly made for non-typed PIDs like `x` and `y`. Such implementation details are out of scope here.)

The only aspect which is not captured by the mapping above is the FIFO rule for equal priority tasks. In such cases, there will be a non-deterministic choice between tasks `x` and `y`. This induces an overapproximation of the system behavior (in the sense that the set of behaviors modeled in IF is a superset of those of the real system), which is a *conservative* abstraction for any *safety* properties verified on the IF model [13].

Timed automata are in principle not expressive enough to model *execution times* in preemptive systems (for which a *stopwatch* concept strictly stronger than the timed automata clocks is in general necessary). There are particular cases in which an encoding is possible, like the one in [15] which only works for systems of tasks with fixed execution time (i.e., best case and worst case execution times are equal), but they are rarely applicable in real systems. Another possibility is to use discrete (integer) counters instead of clocks to model execution times (or, equivalently, to use discrete representations for clocks in analysis), but this yields complex models and worsens the perspectives for combinatorial explosion during verification. For Ravenscar systems, the use of such techniques for schedulability analysis is *not justified*, since the constraints imposed by the profile render them analyzable for example by response time analysis (RTA) techniques rooted in [19].

Consequently, we base the timing of the IF model not on the worst case execution times of tasks, but on the response times previously computed by RTA. Concretely, every task has an associated `response` clock which is not allowed to grow past a `max_response_time` issued from RTA, a condition that is clearly expressible with classical timed automata clocks. This timing model also yields a conservative overapproximation of the real system's timing (the proof of this statement is considered out of scope here).

4.6 Interrupts and the environment

In order to verify properties on the IF model resulting from an Ada system, one has to close it with a model of the environment, which embodies the hypotheses that are made about what it is reasonable to expect from it. Typically, the environment can interact with a system either by triggering interrupts or by calling sub-programs of the system, and the hypotheses concern the order in which such events arrive, the inter-arrival times, etc. Currently the environment's behavior is modeled directly in IF and we do not pose any restrictions on how this is done. An interrupt can be modeled as a call to the *attached* procedure.

We note that the behavioral and temporal non-determinism allowed in IF is key to a simple and expressive modelling of the environment hypotheses, which generally contain some degree of uncertainty.

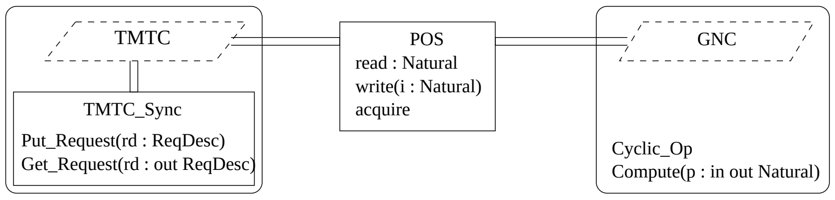
5 Experimental results

5.1 The case study

We validated the mapping defined in §4 on a typical task synchronization example issued from a spaceborne application provided by Astrium Space Transportation within the ASSERT project. Although the functionality of the example is quite simple, the number of Ada objects involved and the size of the code is significant owing to the fact that the code is automatically generated from a high-level architecture description (conforming to the approach described in [5]) and contains different mechanisms for separating functional and architectural aspects, for implementing "archetypical" architectural elements like cyclic and sporadic tasks, etc. The code features two tasks, three protected objects and some 20 procedures, functions and object entries, all spread across 8 packages (including generic ones).

A simplified view of the architecture of the example is depicted in Figure 5. (The notation is inspired from AADL [21]. Rounded rectangles stand for packages, rectangles stand for protected objects, dotted parallelograms stand for tasks, double line connectors signify access to operations. For simplicity, we have renamed some of the system entities to more meaningful names.) In short, the functionality of the example is as follows:

– A task `TMTC` receives sporadic requests (in reality, telecommands from a ground system) upon which it attempts to update an attribute of a protected object (`POS`). For receiving the requests, `TMTC` uses a protected object



TMTC
TMTC_Sync
Put_Request(rd : ReqDesc)
Get_Request(rd : out ReqDesc)
POS
read : Natural
write(i : Natural)
acquire
GNC
Cyclic_Op
Compute(p : in out Natural)

Fig. 5. Architecture of the example.

(`TMTC_Sync`) which exhibits a `Put_Request` procedure and a `Get_Request` entry. The sporadic task implements a protection mechanism against requests made more often than a minimum inter-arrival time (*MIAT*).
- Another task, `GNC` (for *Guidance-Navigation-Control*), periodically reads the attribute of `POS`, performs some computation based on its value, and finally updates it.
- It is required that, when a `TMTC` request comes, the value written to `POS` shall not be overwritten by a value written by the `GNC`, (so that the next cyclic read by the `GNC` reads the value sent by the ground). In the example, this is achieved by encapsulating the entire `GNC` read-compute-write cycle in a protected operation (called `acquire`) of `POS`, thus rendering it mutually exclusive with the writes from `TMTC`.

The property that one wants to verify on this model corresponds to the requirement stated informally above. The requirement is however not sufficiently precise, and one has to express it in terms of strictly defined events like the reception of a `Put_Request` by the `TMTC_Sync`, the effective execution of `Write` by `POS` after acquiring the monitor lock, etc. While doing so, we realized that the requirement is actually a conjunction of two simpler safety properties:

P1 After the effective execution of `POS.write` with value `p` on behalf of the `TMTC` task, the next execution of `POS.read` on behalf of `GNC` returns `p`.
By itself this property is not sufficient since it does not exclude unfair executions in which an effective `TMTC` write is delayed for several `GNC` cycles after the request arrives. Consequently, we added the following property which expresses the fact that a telecommand is effectively handled at latest at the end of the current GNC cycle.

P2 The `POS.write` executed by the `TMTC` after receiving `TMTC_Sync.Put_Request` must start before the next cycle of GNC starts execution.

The two properties can be expressed as IF observers, the automata structures are shown in Figure 6 (the event observation details are omitted).

In order to verify the model, we had to close it with a model of the environment. The chosen environment is a time-non-deterministic ground component which calls `TMTC_Sync.Put_Request` from time to time (though no more often than 1 time unit). In order to reduce the verification state space, we discard traces where a second TC is sent while the previous TC is still pending in `TMTC_Sync`.

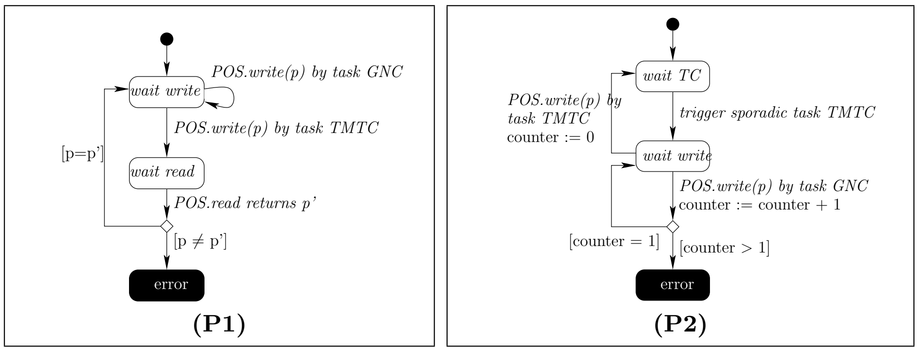


Fig. 6. Desired properties of the example.

5.2 Verification results

Two temporal parameters come into play when verifying the satisfaction of properties by the model: the minimum inter-arrival time (*MIAT*) enforced for `TMTC` requests, and the cycle time of the `GNC`. The values specified initially in the Ada code were respectively 10ms and 8ms. However, it should be noted that the *MIAT* is seen as a protection mechanism of the sporadic `TMTC` task, and may not be actually observed by the environment, which may try to send requests more often.

We have experimented with the following combinations of values (state space size, verification times and results are summarized in Table 1):

– `GNC` *Cycle=8ms,* `TMTC` *MIAT=10ms, actual MIAT observed by the environment < 10ms (1ms).* In this case **P1** is verified, but **P2** may be violated. The result is not surprising, since when a second `TMTC` request comes sooner than expected, it may be delayed for a certain time, and more than one Read request from the GNC may overtake it.
– `GNC` *Cycle=8ms,* `TMTC` *MIAT=10ms, actual MIAT observed by the environment=10ms.* Both P1 and P2 are verified.
– `GNC` *Cycle=8ms,* `TMTC` *MIAT < 8ms (e.g., 6ms), actual MIAT observed by the environment=1ms.* Both **P1** and **P2** are verified. Note that when the `TMTC` MIAT is less than the length of the `GNC` cycle, **P2** is verified regardless of the frequency with which the environment sends the requests.

Note that **P1**, which is the essential property of the system (no value written by the `TMTC` is ever overwritten by the `GNC`) is satisfied in all configurations.

6 Conclusions

We have proved the feasibility of formal verification of Ada Ravenscar systems by translation to a sufficiently powerful formal language based on timed automata, which is supported by the model-checker IF [7, 8]. Due to particular

Parameters	Results	State space metrics	User time
GNC Cycle=8ms, TMTC MIAT=10ms, actual env. MIAT=1ms	P1:true P2:false	(stopped after 30 error scenarios)	$< 1s$
GNC Cycle=8ms, TMTC MIAT=10ms, actual env. MIAT=10ms	P1:true P2:true	45332 states / 122953 transitions	$< 5s$
GNC Cycle=8ms, TMTC MIAT=6ms, actual env. MIAT=1s	P1:true P2:true	336487 states / 902696 transitions	$< 46s$

Table 1. Verification results

features of the IF language, like dynamic process creation, we have been able to produce a mapping which is structure-preserving, meaning that every construct of the initial Ada specification can be identified as an entity in the resulting code (in general an IF *process*). This potentially allows traceability between the two representations, with the obvious benefits. Previous applications of model checking to Ada systems, such as the one presented in [9], do not offer this kind of structure preservation (and consequently, they also lend themselves less easily to automation), in general because of the lack of expressivity in the target language (UPPAAL in [9]).

The translation overhead (in terms of verification state space) also proves to be moderate in our case: for example, the case study presented in §5 yields a large set of 34 IF processes, but thanks to the efficient sub-state sharing algorithms used in IF this does not contribute significantly to the size of the state space, only the combinatorial complexity generated by the two tasks does.

The first part of this work was concerned with defining the translation method and with validating it on a prototypical example. In order for these results to be applicable in practice, an implementation in the form of a compiler is needed. For that, we could rely on the GNAT-based open-source implementation of Ravenscar. Resources allowing, these are our plans for the future.

Acknowledgements We wish to thank the ASSERT team from the University of Padua for providing the Ada code of the example as well as the initial motivation for this work. We also thank all other partners of the ASSERT project for fruitful discussions around the topics presented here.

References

[1] ISO/IEC 8652/1995. *Ada 2005 Reference Manual. Language and Standard Libraries*, volume 4348 of *LNCS*. Springer, 2006.

[2] Nabil Abdennadher and Fabrice Kordon, editors. *12th International Conference on Reliable Software Technologies - AdaEurope, Proceedings*, volume 4498 of *LNCS*. Springer, 2007.

[3] K. Altisen, G. Gößler, and J. Sifakis. A methodology for the construction of scheduled systems. In *FTRTFT*, volume 1926 of *LNCS*, pages 106–120. Springer, 2000.

[4] Aonix. ObjectAda Real-Time RAVEN.
http://www.aonix.com/objectada_raven.html.

[5] Matteo Bordin and Tullio Vardanega. Correctness by construction for high-integrity real-time systems: A metamodel-driven approach. In [2], pages 114–127.

[6] Sébastien Bornot and Joseph Sifakis. An algebraic framework for urgency. *Inf. Comput.*, 163(1):172–202, 2000.

[7] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0: A validation environment for component-based real-time systems. In *CAV*, volume 2404 of *LNCS*, pages 343–348. Springer, 2002.

[8] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. *International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Revised Lectures*, volume 3185 of *LNCS*, pages 237–267. Springer, 2004.

[9] A. Burns and A. J. Wellings. How to verify concurrent Ada programs: the application of model checking. *ACM SIGADA Ada Letters*, 19(2):78–83, 1999.

[10] Alan Burns, Brian Dobbing, and Tullio Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. *Ada Lett.*, XXIV(2):1–74, 2004.

[11] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages (Third Edition).* Addison Wesley, 2001.

[12] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications, 2nd ed.*, volume 23 of *Real-Time Systems Series.* Springer, 2005.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[14] Juan Antonio de la Puente, José F. Ruiz, and Juan Zamorano. An open Ravenscar real-time kernel for GNAT. In *Ada-Europe*, volume 1845 of *LNCS*, pages 5–15. Springer, 2000.

[15] Elena Fersman, Leonid Mokrushin, Paul Pettersson, , and Wang Yi. Schedulability analysis using two clocks. In *9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *LNCS*. Springer, 2003.

[16] Object Management Group. Unified modeling language. Available at http://www.omg.org/spec/UML/.

[17] I. Hamid, B. Zalila, E. Najm, and J. Hugues. A generative approach to building a framework for hard real-time applications. In *31st Annual NASA Goddard Software Engineering Workshop*, Baltimore, USA, March 2007.

[18] ITU-T. Languages for telecommunications applications – Specification and Description Language (SDL). ITU-T Revised Recommendation Z.100, 1999.

[19] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[20] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[21] SAE Aerospace. Architecture Analysis & Design Language (AADL). SAE Technical Standard, November 2004.

[22] Bechir Zalila, Irfan Hamid, Jérôme Hugues, and Laurent Pautet. Generating distributed high integrity applications from their architectural description. In [2], pages 155–167.