Towards a Runtime Verification Framework for the Ada Programming Language

André de Matos Pedro[1], David Pereira[1], Luís Miguel Pinho[1], and Jorge Sousa Pinto[2]

[1] CISTER/INESC TEC, ISEP, Polytechnic Institute of Porto, Portugal
{anmap,dmrpe,lmp}@isep.ipp.pt
[2] HASLab/INESC TEC & Universidade do Minho, Portugal
jsp@di.uminho.pt

Abstract. Runtime verification is an emerging discipline that investigates methods and tools to enable the verification of program properties during the execution of the application. The goal is to complement static analysis approaches, in particular when static verification leads to the explosion of states. Non-functional properties, such as the ones present in real-time systems are an ideal target for this kind of verification methodology, as are usually out of the range of the power and expressiveness of classic static analyses. In this paper, we present a framework that allows real-time programs written in Ada to be augmented with runtime verification capabilities. Our framework provides the infrastructures which is needed to instrument the code with runtime monitors. These monitors are responsible for observing the system and reaching verdicts about whether its behavior is compliant with its non-functional properties. We also sketch a contract language to extend the one currently provided by Ada, with the long term goal of having an elegant way in which runtime monitors can be automatically synthesized and instrumented into the target systems. The usefulness of the proposed approach is demonstrated by showing its use for an application scenario.

1 Introduction

Real-time embedded systems are usually large and complex, continuously interacting with the external environment. A single real-time system is usually made of several sub-components concurrently competing for the system's resources. Many, if not all, of these sub-components are not produced in-house, and are assembled together from diverse sources, being these sometimes black-boxes to the system integrator. Some parts may also be *Commercial Off-The-Shelf* (COTS) components that, although being economic, have the drawback of introducing safety concerns, as are usually not accompanied with their source-code and/or complete specification.

Given the critical role of many of the real-time systems developed, their source code is subject to exhaustive testing efforts, which may be extremely expensive. In particular, in what the context of this work relates to, some static

analysis tools have been developed to check the correctness of this class of systems, in some cases applied with success. *Model Checking* [5] is among the most well-known static analysis approaches, but it has the drawback of quickly exploding due to state space search. Other alternative static analysis approaches have strong drawbacks as well [12], therefore in the last years *Runtime Verification* (RV) [1, 12] has emerged in order to complement the existing limitations. RV is concerned with providing theories, languages, and procedures that allow developers to improve programs with specifications of properties to be checked upon execution time. In a nutshell, the idea of RV is to transform the extra specifications and synthesize them into *monitors* which are added to the system. A monitor is a computational element that is responsible for observing (part of) the system and make *verdicts* about its correct execution.

A real-time system is a paradigmatic case to show what RV can bring in terms of safety for software. It is easy to identify, at least, two of the more relevant reasons for such approach: it is very hard to verify during design all the properties that real-time system must exhibit, which requires that some properties are verified only during execution; and non-functional properties such worst-case execution time are extremely hard to *prove* statically. With a runtime approach we can have direct access to the states of the tasks of interest and determine if their conditions hold, thus being able to detect erroneous execution and act accordingly. The introduction of monitors can be handled by a static automatic generation tool, from the properties which are to be checked, and can be time-bounded, since monitors are scheduled as any other task executing in the system.

Most of the RV frameworks developed so far are for Java, and were designed to address software development in that programming language. They are based on formal approaches such as temporal logics [2] and regular expressions [17] and are directed towards ensuring functional correctness. These formal approaches provide expressive languages for writing contracts, and for which their synthesis and implementability are feasible and efficient. Although, ideally, the theory and tools that resulted from this effort should be used for embedded systems (possibly exhibiting real-time characteristics), only a couple of works have addressed this kind of systems. One of these is PathFinder [11], for critical systems written in Java; the other is CoPilot [14, 13], a functional specification language and toolchain for the safe runtime monitoring of ultra-critical software.

In this paper, we present the RMF4Ada framework that aims at complementing the available developments as follows: first, it is an RV framework that intends to use the safety properties and expressiveness of the Ada programming language, which we consider very relevant to implement real-time systems; secondly, we target the specification and synthesis of monitors that are capable of verifying important non-functional properties, such as meeting deadlines, respecting worst-case execution times, among others. The focus of this paper is in the structure of the code that allows synthesizing monitors and the management of events and sequences of events. We also describe an extension to the Ada 2012 specification of contracts so that in the future, it can support contracts

for non-functional properties and that can be verified using frameworks such as this proposal. In order to show the potential of our proposal, we present an Ada implementation of a *mine-drainage controller* enriched with RV behavior using this framework.

2 The RMF4Ada Runtime Verification Framework

In this section we introduce and describe the details of a novel RV framework for the Ada programming language. RMF4Ada combines aspects of *Runtime Monitoring* (RM) (the field that studies ways to define, implement, and control monitors), formal languages, and software architecture methods to provide the infrastructure that is needed to equip an Ada program with RV functionality. The core of RMF4Ada is a set of Ada packages that provide schemas for monitors (possibly executing in different patterns), data structures to represent formal languages and the evaluation of their formulas/terms, and components to represent and manage the events of the system that one might be interested in verifying. It is not simply yet another RM framework, since the properties to be verified or enforced are generated from timed specifications written in the supported formal languages, in a correct-by-construction way.

The architecture of RMF4Ada is depicted in Figure 1. It is divided into two sub-components: an Instrumentor and a Creator. The Instrumentor is a tool which manages the environment for system instrumentation, and that couples monitors that are automatically synthesized *a priori* by some built-in mechanism of the framework, or by some third-party tools; the Creator is a tool that synthesizes monitors based on the specification written in the contracts of the original program; the output is the corresponding Ada executable code. The Creator also contains mechanisms to generate monitors according to different modes of operation. The generic dynamics of RMF4Ada is as follows: the Instrumentor generates an event manager (among other things, responsible for keeping an execution trace of the system with the correct order, and inform monitors of available events for consuming), and in-lines instructions in the original source-code that create the symbolic representation of the events and communicates them to the event manager; afterwards, the Creator generates the monitors and adds them to the source-code already instrumented as a new package, in order to produce the final program with the RV layer working according to the specifications in the original system.

As we have pointed out earlier, the monitors that are added to the target system do not need to follow a single execution behavior. RMF4Ada provides support for two different *modes* of operation for monitors: *time-triggered* monitoring and *event-triggered* monitoring. In the time-triggered mode, monitors run periodically; the time period is calculated beforehand in order to avoid higher detection delays for the events of interest as we propose in [9, 10]. In the event-triggered mode, monitors execute when events occur, are considered sporadic tasks, and consume the available events respecting a given execution pattern.

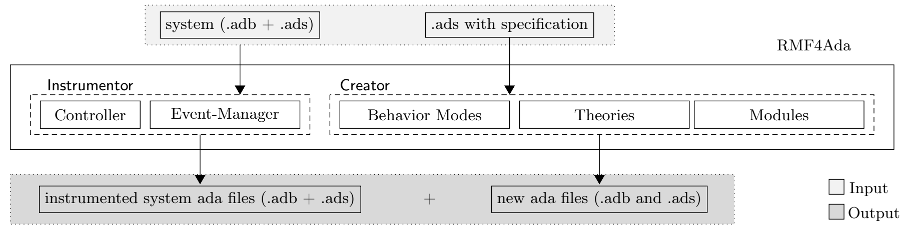

Fig. 1. RMF4Ada Architecture

2.1 The Runtime Monitoring Library

We begin with an overview of the library's structure, and then we introduce library settings such as events, monitor modes, and monitor context-switches.

Structure. The runtime monitoring library (RML) of RMF4Ada is composed of three modules: one that encodes the formal systems used to allow the specification and verification of contracts, other responsible for providing the abstract data types fundamental to construct monitors, and another for building instances of monitors that can be added to the program under consideration. These modules are named *Theories*, *Abstract Data Types*, and *Monitors*.

Theories. This module provides an hierarchy of objects that allow to implement the inductive nature of the sentences of the formal languages that we have adopted to specify contracts as well as to provide the semantical evaluation functions for these sentences, which are the functions responsible for giving verdicts when the target system is executing. Currently, we have considered two formal systems: *Timed Regular Expressions* (TRE) [15], and *Restricted Metric Temporal Logic with Durations* (RMTL-$\int$) [10, 9].

Abstract Data Types. This module provides a set of well-known abstract data types that are fundamental to architect the rest of the framework. Among these are included an array-based *first in first out* (FIFO), a circular FIFO, and an array-based stack. The abstract data structures themselves are not covered in this paper, since they are not interesting from the point of view of runtime monitoring.

Monitors. This module includes the types defining *events* and *event traces*. These are the primitive notions where the activity of the system under consideration are stored. Another building-block provided is the protected *event manager type*, responsible for keeping the events in an internal FIFO. Finally, this library offers the type for monitors, which is specialized according to its operations mode, *i.e.*, its either a task type implementing a periodic (time-triggered) or a sporadic (event-triggered) behavior.

Events. In our framework, all events are typed, thus avoiding miss-specification when instrumentation is performed. Ada discriminant records have been used to statically ensure the release of the correct events. The creation of events is done through several functions. Managing events and traces is the responsibility of the event manager, implemented as an Ada protected type. The event manager allows the system under observation to enqueue several different types of events (as we see later), and to pop relevant events from the global trace into the monitor's local memory. In the code below we give a small example, describing how the release of an event can be performed:

```
-- Generate an event meaning a task_release of Task_A
 Ev_A: Events_Concrete.Event :=
 Events_Concrete.Generate_Event_Task_Release_Structure (
  Name => Task_A,
  Monitor_Identifier_List => (Monitor_FormulaOne_Id => true),
  Time => Time_Unit_Type(TIME_STAMP)
 );
```

In the code above, we use the `Generate_Event_Task_Release_Structure` function to construct a release event. `TIME_STAMP` is a value in accordance with the type `Time_Unit_Type`, and `Monitor_Identifier_List` is a binary map that indicates that the event `Ev_A` could be used by the monitor identified by the value of `Monitor_FormulaOne_Id`. The other monitors, if available, need to be assigned to false, since `Monitor_Identifier_List` is a static list that contains a Boolean assigned for each available monitor. The value of `Task_A` is the identifier of the event task release.

Monitoring Modes. Different monitor modes correspond to different task types. For a monitor of type event-based we use the synchronous task control to suspend the task until the Boolean flag `Event_Based_Is_Sporadic` is true. The code to handle the event and read the event from the event manager with corresponding operations on the memory of the monitor, is the following:

```
-- Use of Timming Events for waiting order
Ada.Synchronous_Task_Control.Suspend_Until_True (
 Event_Based_Is_Sporadic
);

-- Get event from event manager
Event_Manager_For_Monitor.Protected_Event_Manager.readEvent (
 Id => Monitor_Id,
 E => Event_Manager_For_Monitor.Trace_For_Event_Manager.Trace_Elements
     .Event(Tmp_Event)
);

-- Set event to the dedicated trace structure of the monitor
Object.Assign_Event_To_Trace(Event_Manager_For_Monitor.
    Trace_For_Event_Manager.Trace_Element_Type(Tmp_Event));
```

`Tmp_Event` is the event that has been read from the event manager, and `object` the monitor object that contains a trace structure. Note that pushing and pop-

ping event from event manager are protected operations that avoid simultaneous concurrent accesses. We use the protected type `Protected_Event_Manager` to ensure these operations. In time-triggered monitors identified by type `Time_Based_Mode_Type` we use a periodic task with the `delay until` sentence. The code to define this behavior is the following:

```
-- if there are events to consume discard the new ones
if Event_Manager_For_Monitor.Trace_For_Event_Manager.
    Get_Number_Elements(Object.Trace) < 1 then

  -- Get list of events
  Event_Manager_For_Monitor.Protected_Event_Manager.
      readListOfEventsBounded (
    Id => Monitor_Id,
    List => Temporary_Trace
  );

  -- Set list of events to trace structure of the monitor object
  Event_Manager_For_Monitor.Trace_For_Event_Manager.
      Push_ListOfEventsBounded (
    Trace => Object.Trace,
    Trace_Tmp => Temporary_Trace
  );

end if;
```

`Monitor_Id` is the identifier used to get the relevant events from the event-manager, and `Temporary_Trace` is a temporary local variable used in transferring events. The if statement maintains the `Push_ListOfEventsBounded` correct, since it does not issue any array out of bounds access. In the worst-case scenario, the `Temporary_Trace` array list may have the same size, and if one or more elements exist in the local monitor trace, then the same number of events cannot be fitted. The procedure `Monitor_Function` that is recursively defined is called in both modes by the instruction:

```
Object.Monitor_Function;
```

`Object` variable includes all available data structures to store the inputs and outputs of the function. After establishing the monitor modes a piece of code need to be coupled for time-triggered mode after the monitor invocation. The code remaining for this is the following:

```
if Object.Mode = Time_Based_Mode_Type then
  Next_Time := Next_Time + Release_Time;
  delay until Next_Time;
end if;
```

`Next_Time` is a clock value. The above code block intends to establish the behavior of the periodic tasks inducing the system to a sleep state until a certain time is reached to wake-up.

Monitor Context-Switches. Monitors employ high level software context switches based on one stack data structure to be incrementally evaluated. As

we understand high level context switches are the ones that are implemented by software with the aid of memory RAM instead of processor registers. The stack allows us to encode recursive call into a loop, and to control the execution using some control conditions. We introduce this approach as the *subtractive-based abstraction* for runtime monitoring. Our abstraction begins by popping the generic structure that contains the tuple and the verdict, which are the inputs and the outputs of a monitor function, respectively. The procedure that performs a pop operation is invoked using the following code:

```
-- get tuple from stack
Monitor_Stack.Pop(
  Item => Input_Output_Par,
  From => Object.Stack_For_Arguments
);

-- subdivide state to recall monitor function
Tuple := I_O_Par.Tuple;
Verdict := I_O_Par.Verdict;
```

The context-switch restores the previous outcome values from variables `Tuple` and `Verdict` to be able to call the `Procedure_For_Monitor` procedure with a certain trace. The actual tuple represents the last global state of the monitor, and the last verdict that has been saved, respectively. In Ada the monitor is called according to the procedure:

```
Procedure_For_Monitor( Object.Trace, Tuple, Verdict );
```

One cycle of execution or a step is executed by this instruction, which we denote as the `One_Step` restriction. This step may keep the `Trace` structure intact, indicating that no symbols have been consumed. By the way some executions cannot consume any event symbol of the trace and a buffer overrun of the local monitor trace can occur. To tackle this we assume that the execution is progressive and the monitor execution is recursively defined.

To manage the context-switches we consider four conditional statements.

1. Step-based condition – The evalution of the monitor ends when one step of the recursive function `Procedure_For_Monitor` is executed. After each execution step the state is stored for a further resume of the monitor execution. The Ada code for this restriction is as simples as follows:

   ```
   I_O_Par.Tuple := Tuple;
   I_O_Par.Verdict := Verdict;

   Monitor_Stack.Push (
     Item => I_O_Par,
     Onto => Object.Stack_For_Arguments
   );
   ```

2. Symbol-based condition – A symbol consumption is the necessary condition to suspend the monitor execution, and proceed with the context-switch. `One_Step_Until_Symbol_Is_Consuming` is the Boolean variable that activates

this condition in the RML. Any external feedback is required for the procedure `Procedure_For_Monitor` since the re-execution only continues if the trace is unchanged. We also assume that the procedure `Procedure_For_Monitor` is progressive, which indicates that eventually some event is consumed. The Ada code that we use to re-evaluate the monitor procedure is the following one:

```
while not Trace_Has_Been_Changed(Object.Trace) loop
 Procedure_For_Monitor( Object.Trace, Tuple, Verdict );
end loop;

I_O_Par.Tuple := Tuple;
I_O_Par.Verdict := Verdict;

Monitor_Stack.Push (
 Item => I_O_Par,
 Onto => Object.Stack_For_Arguments
);
```

`Procedure_For_Monitor` procedure ends with the push of the `I_O_Par` into the `Monitor_Stack` (accordingly with the restriction `One_Step`), and the function `Trace_Has_Been_Changed` establishes a comparison between two traces (new and old ones). To be an efficient search over both traces labels are used to indicate when a trace is equipped with a new event.

3. Time-bounded condition – A temporal bound for the evaluation of a monitor is used to decide the suspension instant of a context-switch. `One_Step_Until_T_Time_Units` is the Boolean flag used to activate this condition in the RML. The evaluation is made until the time elapsed exceeds the time allowed for the monitor execution. To describe such behavior we use the execution time timers [18]. Such timers allow us to force a monitor to execute in a stipulated constant time by triggering an interruption when the available time expires. In that point the last execution is rejected if a step of the monitor's execution is not completed. The excerpt of Ada code establishing such behavior is the following one:

```
Ada.Execution_Time.Timers.Set_Handler
 ( ETT_Timer,
  Ada.Real_Time.To_Time_Span(CONSTANT_TIME),
  Object.Control.Budget_Expired'Access );

while not Object.Control.Budget_Is_Expired loop
 Procedure_For_Monitor( Object.Trace, Tuple, Verdict );
end loop;

I_O_Par.Tuple := Tuple;
I_O_Par.Verdict := Verdict;

Monitor_Stack.Push (
 Item => I_O_Par,
 Onto => Object.Stack_For_Arguments
);
```

`Object.Control.Budget_Expired` is a protected procedure that waits for the end of the step execution of the monitor, `Object.Control.Budget_Is_Expired` is the function that returns true if the execution time timer expires or false otherwise, and `CONSTANT_TIME` is the constant that indicates the allowed duration for monitor execution.

4. Step-bounded condition – One step of a monitor's execution is performed n times. `One_Step_N_Times` is the condition available in the RML. We tackle this restriction by a simple for loop, as follows:

```
for I in 1..Object.Counter−1 loop
  Procedure_For_Monitor( Object.Trace, Tuple, Verdict );
end loop;

I_O_Par.Tuple := Tuple;
I_O_Par.Verdict := Verdict;

Monitor_Stack.Push (
  Item => I_O_Par,
  Onto => Object.Stack_For_Arguments
);
```

`Object.Counter` defines how many steps the monitor shall execute before a context-switch is performed. The remaining code has the same meaning as described before.

2.2 Library Usage

RML is the support library of RMF4Ada and is available in [7]. Our library encourages developers to make constraints using strong type checking instead of condition-based tests. This abstraction allows the software designers to make less-mistakes when library is used as well as to maintain the integrity of the instrumentation process. In some cases the memory space can be reduced due to the message pass using strings is avoided, the registration of monitors avoids any registration in the event-manager, and the event-manager and monitors discard any filters since the unexpected types of events cannot take place. An overview of library's interconnection is depicted in the Figure 2, including the main blocks, such as, the static instrumentation, the event-manager protected type, and the set of tasks representing the monitors generated automatically by some theory.

A program to be instrumented should be designed following the hierarchy of Ada packages. The main package should be the point where initializations of RML are made, and the remain packages are necessarily extensions of the root package. The instrumentation is made using the push procedures of the event-manager positioned at certain points in program code. We have been included in our library three types of events each one with several sub-types such as

- task release, task begin, task sleep, task resume, and task end;
- pre and post procedures, protected pre and post procedures, pre and post functions, and protected pre and post functions; and

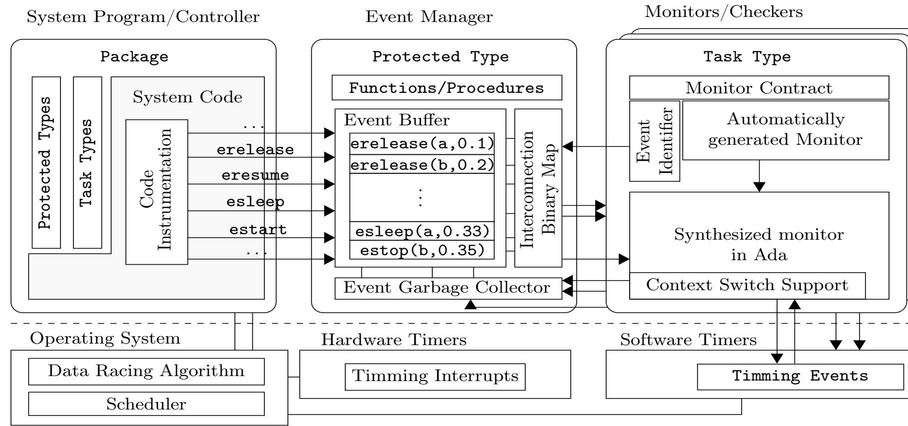

Fig. 2. Illustration of the interconnection of the element blocks provided by the RML

– pre and post assignments, and protected pre and post assignments.

Other manually specified events are discarded in this paper but will be addressed for further phases. The integrity of the event sequences that our framework supplies also should be enforced.

Schedulability analysis can be enforced online using runtime monitoring techniques as well as offline in the development phase. The integrity of the system schedulability analysis requires that events triggered by tasks are correctly pointed and instrumented for each available Ada task type. For instance, this is enough for online schedulability analysis of periodic resource models as proposed in [9]. In the case of the preemptive fixed priority scheduler allowed in Ada by Ravenscar rules, we need to instrument all system tasks in order to provide the events correctly for event monitoring. Our framework should be able to identify if a higher priority task trigger an event task begin, then some time units ago an event task sleep may have occurred. To do it, the framework only needs to select the last event that has occurred in the event trace provided by event-manager. This avoids any instrumentation of events at the operating system level (more precisely at the internals of the scheduler).

We have described a subtractive-based abstraction that leaves us to couple diverse monitoring models in our framework as well as aiding to ensure finite execution segments for each execution cycle. The monitor should be fitted into these abstractions after the synthesis process. A garbage collector also provided by the event-manager allows us to dynamically remove unnecessary events while the system is executing. All events are managed by the event-manager entity, and the interconnection binary map is established to identify the relevant events for each monitor.

3 Contract Language Extension for Runtime Verification

Currently, the newest version of the Ada programming language – Ada 2012 – provides a language of contracts for the dynamic verification of functional properties. Subprograms can be equipped with preconditions, postconditions, dynamic and static predicates, and types can be binded with invariants. However, the contract language is not enough to address the characteristics of RV that are supported by RMF4Ada.

In this section, we propose an extension to the current contract language that is mainly composed of a couple of new syntactic contract constructions such as

1. a construction to establish the type of execution mode of the monitors, and
2. a construction to define the property that the software designer is interested in checking at runtime.

Our extension proposes the usage of a contract of the form `Monitor_Mode =>` $\mathcal{M}$, such that $\mathcal{M}$ is either the keyword `Event_Triggered` or the keyword `Time_Triggered` representing, respectively, an even-triggered monitoring mode or a time-triggered monitoring execution mode. Once the modus operandi of monitors is stated, we will establish formal languages for enforcement of non-functional properties such as temporal order, and durations. Each monitor specification is enforced by the usage of the contract `Monitor_Case =>` $(\mathcal{T}, \mathcal{C})$, such that $\mathcal{T}$ is the formal language of the contract chosen for property specification, and $\mathcal{C}$ is a sentence of this language that specifies some property of interest for runtime enforcement. The pair $(\mathcal{T}, \mathcal{C})$ has one of the following types:

1. a pair (TRE, α) for runtime contracts based on TREs with α being inductively defined by
$$\alpha ::= 0 \mid 1 \mid a \in \Sigma \mid \alpha + \alpha \mid \alpha\alpha \mid \alpha^\star \mid \langle \alpha \rangle_I,$$
where Σ is the set of all events, and I is a time interval of the form $[a..b]$ with $a, b \in \mathbb{R}_0^+$, or
2. a pair $(\mathsf{RMTLD}, \varphi)$ for runtime contracts based on RMTL$-\int$ with φ being inductively defined by
$$\beta ::= c \in \mathbb{R}_0^+ \mid x \in \mathcal{L}v \mid \mathsf{duration}[\beta]\,\varphi,$$
$$\varphi ::= p \in \mathcal{P} \mid \beta\,\mathsf{op}\,\beta \mid \mathsf{not}\,\varphi \mid \varphi\,\mathsf{or}\,\varphi \mid \varphi\,\mathsf{U}_n\,\varphi \mid \varphi\,\mathsf{S}_n\,\varphi \mid \mathsf{Ex}\,x\,\varphi,$$
where $\mathcal{P}$ is the set of propositions for all available events, $\mathsf{op} \in \{<, \leq, >, \geq\}$, $\mathcal{L}v$ is the set of logic variables, and n is a constant time in $\mathbb{R}_0^+$.

Note that each monitor specification uses the pre-defined mode of operation, until a new mode of operation is specified.

An event is defined by the introduction of the attribute `Event` in Ada. We use the shortcut `object'Event(e)` for the meaning of the sentence **`for`** `object'`**`Event`** **`use`** `e`. All events are previously defined, including the event `ANY`, which dictates that any event can occur. Moreover, we also add the attribute `Time` to define

timing settings of the task types dealing with periods and WCET that have been assigned prior to execution.

As we have described, RMF4Ada provides computational support for these formal systems. There are object hierarchies to represent the (mutually) inductive structure of the terms and the formulas of the two formal systems. Moreover, evaluation functions for the statements are also included. In the rest of this paper, we will describe the implementation of a safety-critical real-time system in Ada with the support of RMF4Ada, and we will use the extension of contracts presented in this section. However, the synthesis of the monitors has been obtained manually, because the tool for doing it automatically is not yet available at the present time.

4 Experimental Scenario

In this last section, we describe the implementation of a real-time program that models a mine-drainage scenario in Ada. This scenario has been proposed by Burns *et al.* in [4, 3]. The authors propose a mine drainage controller, and a station for employers to control and monitor the mine state. The system contains a water pump that drains the water into the ground, which needs to be switched off when a critical level of methane is reached. The implementation is available in [8], and employs our idealized contract extension to the RV. The evaluation of RMF4Ada for this particular scenario has been performed as follows: first, we have implemented the mine-drainage scenario guided by the case study presented by the authors [3]; afterwards, we specified the necessary contracts and synthesized them into monitors by hand, due to the lack of the Creator tool. Since the Instrumentor is not currently ready as well, we also performed the instrumentation by hand. After these steps have been completed successfully, we have compared the performance of the original implementation with the monitor-enriched one in order to measure the overhead imposed by the constructions provided by RMF4Ada.

The graphical interface of the simulator is presented in Fig. 3. We can observe in the table depicting the monitoring of tasks in Fig. 3 the minimum and maximum delay values of the computation time of the tasks. The field Count indicates the number of iterations that one task made in the last ten seconds. These values will be used as a reference to compare the original system and the system resulting from its instrumentation and coupling with runtime monitors. The experiments have been performed on an Intel Core i3-3110M at 2.40GHz CPU, and 8 GB RAM running on Fedora 18 x64, in a uniprocessor setting. In the future, we plan to do further experiments where we will use the MaRTE OS [16] which is an operating system fully implemented in Ada. This allows us to speculate that since we got good results in a Linux setting without strict hard real-time constraints on the scheduler side, then we expect better ones in a native real-time operating system as the one just pointed out.

```
                         Mine drainage Simulator
| Environment (Gas)      | Level (units)           || Pump Spec.   | Values        |
|========================|=========================||==============|===============|
| Methane (CH4)          | 771 (ppm)               || State        | OFF           |
| Carbon Dioxide (CO2)   | 1097 (ppm)              || Used Time    | 0.000000000   |
| Air Exaust (CO2)       | 973 (cubic/meters))     || Issue Time   | 83.583503723  |
| Water Reservoir (H2O)  | 9832 (l)                || Life Time    | 202.085708618 |
| Water Pipe Flow (H2O)  | 1 (l/s)                 ||              |               |
|==================================================||==============================|
        Operator Console                              Tasks Monitoring
| System Status: Stable       |   | Name      | Deadline Min,Curr,Max(ms)  | Count |
| Pump Status:   Stable       |   |===========|============================|=======|
| Pump:          OFF          |   | CH4sense  | 0.0024   0.2765   0.5729   | 126   |
| Blower:        ON           |   | CO2sense  | 0.0013   0.1709   0.6140   | 101   |
|======= Monitoring =======|  |   | Airflow   | 0.0010   0.4443   0.6201   | 101   |
| CH4 level:  771             |   | WFLow     | 0.0060   0.4866   1.1356   | 11    |
| CO2 level:  1097            |   | HWLevel   |                            | 0     |
| Air flow:   973             |   | LWLevel   |                            | 0     |
| Water flow: 1.00            |   | Sim       | 0.0054   0.0092   0.5662   | 202   |
|-----------------------------|   |====================================================|
        Time
 10 (seconds)
===================================================================================
 :
```

Fig. 3. Command Line Interface of the Mine Drainage Simulator

Enforcement of Timing properties. Formulas for enforcement of timing properties have been established for task timing analysis, and for a particular execution sequence of the simulation environment. Both formulas will be synthesized, producing two monitors in Ada language that can be coupled manually into the simulator. As a first example of our development, we introduce a contract specification that states that a given task named task `T_Simulation` always has a duration smaller than the pre-defined worst-case execution time. It allows us to monitor a task without using *execution time timers*, which are highly dependent on the operating system API. However, our approach may contain more overhead than one using execution time timers since it has been implemented at a source-level instead of a level closer to the hardware. We are also concerned by the fact that the behaviors supported by execution time timers are stricter than the behaviors supported by our monitors, and the expressiveness is incomparable: on the one hand, the time isolation is a positive point of our framework since we can establish that a set of tasks has a certain budget to execute in a certain period; on the other hand, the overhead is only the big disadvantage when systems have very strict hardware resources. The task specification using our contract language for this duration limited task execution case is the following:

```
task type T_Simulation (period: integer; deadline: integer)
 with
  Monitor_Mode => Event_Triggered,
  Monitor_Case => ( RMTLD ,
     T_Simulation'Event(Task_Release) next implies
     duration[T_Simulation'Time(period)]
     T_Simulation'Event(ANY) < T_Simulation'Time(wcet)
    );
```

The `Monitor_Case` contract defines a RMTL-$\int$ formula to be evaluated in event-triggered mode, `'Event` is an attribute that identifies the event or events to be used, `ANY` identifies all the events assigned to a certain structure, and `a` **next**

implies `b` is a shortcut to the logic formula $\neg a \vee a \,\mathsf{U}_{\leq n}\, b$ with a n greater than the size of the observation as defined in [10]. The synthesis of this monitor can be found in [6]. A second example is a protected type `Protected_Environment` executing a certain timed order. Calls for the protected environment are made beginning with the release event `pre` of the function `read_CH4` and are followed by any combination of events with duration of at most twenty milliseconds. Finally, it ends with the return of the function `read_CO2` value identified by the event *post*. The specification for this second example is the following:

```
protected type Protected_Environment
  with
    Monitor_Mode => Time_Triggered
    Monitor_Case => ( TRE,
      ( Protected_Environment.read_CH4'Event(pre) .
        <(Protected_Environment'Event(ANY))*>[0..20] .
        Protected_Environment.read_CO2'Event(post))*
      ) ,
  is
    function read_CO2 return CO2_Level_State;
    function read_CH4 return CH4_Level_State;
    function read_Air_Flow return Air_Exhaust_State;
    function read_WaterPipe_Flow return WaterPipe_Flow_State;
end;
```

The interval [0..20] assumes the time unit of milliseconds, and `pre` and `post` are the events that occur before and after the execution of a function, respectively. The code resulting from synthesizing both the previous examples can be found in [8], particularly in the specification files `monitor_function_formulaone.ads` and `monitor_function_formulatwo.ads`, respectively.

Time Isolation. It is important to note that time isolation can be ensured using our framework by applying directly the established formalization in [9, 10]. In this work we only need to assume that a task release a certain set of events. Our Instrumentor tool should be capable to correctly instrument these events. The time isolation is a major advantage when several systems should be merged in order to reduce hardware costs, but the level of criticality is ensured and mixed.

4.1 Verdicts

The Instrumentor tool generates four files that are the `monitoring.ads`, the `monitoring.adb`, the `spec.ads`, and the `spec.adb`. The Spec package contains the definition of events used to analyze the *system under observation* (SUO), and the trace type. The Monitoring package instantiates the generic packages provided by RML to be included into the SUO at the instrumentation phase. The package includes the event-manager definition, the both monitor assignments for execution using the monitor collection, and a controller to initialize and finalize the instrumentation before execution begin and end, respectively.

The results are surprisingly positive. The monitor generated by the first formula introduces a maximum overhead of $11\mu s$ for every task iteration of the

task `T_Simulator`. Considering that a wake up from an absolute delay until the operation, including one context switch, in Marte OS is 8.8μs [16] in a Pentium III at 500mhz, the results are satisfactory. We are using one core of an Intel i3 at 2.4GHz. The monitor generated by the first formula has an estimated *worst case execution time* (WCET) of 53μs for one step execution in event-triggered mode. We can conclude that we have maximum overhead of 11+53μs for runtime monitoring. This value has been estimated for 202 iterations of the task `T_Simulator` or the first ten seconds of one execution.

5 Conclusions

In this paper, we have presented a framework for enabling the instrumentation of Ada programs with monitors that enforce RV behavior. The evaluation of our framework shows that the overhead is minimal when compared to the original system, and that it could even be decreased by making our framework constructs more efficient and implementing them in an operation system with actual real-time behavior. Moreover, we have introduced a small extension to the current Ada contract language for enabling the specification of contracts to be checked by monitors. Finally, we provided the general structure of the complete framework, which will include a Creator tool for performing the automatic synthesis of monitors from contracts, and an Instrumentor tool that will automatically instrument a target program with event notification and adequate monitor operation.

Currently, we use two formal systems to support the construction of contracts, namely, TREs and the RMTL-$\int$ timed temporal logic. In the future we plan to explore further formal systems, and to generalize our current implementation in order to allow the integration of such new formal systems in a modular approach, without needing to recompile the framework. Another interesting point to investigate will be the adequacy of RMF4Ada in a multi-core environment, which raises new interesting and hard challenges, such as how to deal with simultaneous events occurring in the different cores, and ways to combine cores solely for monitoring while having the rest of the cores for executing the code of the actual application. Finally, we also want to explore the adequacy of the framework for COTS as internal black-box components, without source-code or rigorous specification for those components.

Acknowledgments The authors would like to thank the anonymous reviewers for their comments that helped improve the manuscript. This work was partially supported by Portuguese National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within projects FCOMP-01-0124-FEDER-037281 (CISTER), FCOMP-01-0124-FEDER-015006 (VIPCORE) and FCOMP-01-0124-FEDER-020486 (AVIACC); and by FCT and EU ARTEMIS JU, within project ARTEMIS/0003/2012, JU grant nr. 333053 (CONCERTO).

References

1. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.
2. P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, March 2000.
3. A. Burns and T. M. Lin. An engineering process for the verification of real-time systems. *Form. Asp. Comput.*, 19(1):111–136, March 2007.
4. A. Burns and A. M. Lister. A framework for building dependable systems. *Comput. J.*, 34(2):173–181, April 1991.
5. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
6. André de Matos Pedro, David Pereira, Luís Miguel Pinho, and Jorge Sousa Pinto. Monitors provided for the Mine Drainage System Simulator. `http://webpages.cister.isep.ipp.pt/~anmap/adaeurope14/examples/mine_drainage/monitors/`. Accessed: 2013-12-15.
7. André de Matos Pedro, David Pereira, Luís Miguel Pinho, and Jorge Sousa Pinto. Runtime Monitoring Library for RMF4Ada. `http://webpages.cister.isep.ipp.pt/~anmap/adaeurope14/`. Accessed: 2013-12-15.
8. André de Matos Pedro, David Pereira, Luís Miguel Pinho, and Jorge Sousa Pinto. The Mine Drainage Simulator Code. `http://webpages.cister.isep.ipp.pt/~anmap/adaeurope14/examples/mine_drainage/system/`. Accessed: 2013-12-15.
9. André de Matos Pedro, David Pereira, Luís Miguel Pinho, and Jorge Sousa Pinto. Logic-based Schedulability Analysis for Compositional Hard Real-Time Embedded Systems. In *Proceedings of the 6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, CRTS '13, 2013.
10. André de Matos Pedro, David Pereira, Luís Miguel Pinho, and Jorge Sousa Pinto. A Compositional Monitoring Framework for Hard Real-Time Systems. In *Proceedings of the 6th NASA Formal Methods Symposium*, NFM '14, 2014. To appear.
11. Klaus Havelund and Grigore Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical Computer Science*, 55(2):200–217, 2001.
12. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
13. Lee Pike, Sebastian Niller, and Nis Wegmann. Runtime verification for ultra-critical systems. In *Proceedings of the Second international conference on Runtime verification*, RV'11, pages 310–324, 2012.
14. Lee Pike, Nis Wegmann, Sebastian Niller, and Alwyn Goodloe. Copilot: Monitoring embedded systems. In *Innovations in Systems and Software Engineering: Special Issue on Software Health Management*, 2012.
15. Riccardo Pucella. On equivalences for a class of timed regular expressions. *Electr. Notes Theor. Comput. Sci.*, 106:315–333, 2004.
16. Mario Aldea Rivas and Michael González Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In *Reliable Software Technologies - Ada-Europe*, volume 2043 of *Lecture Notes in Computer Science*, pages 305–316, 2001.
17. Koushik Sen. Generating optimal monitors for extended regular expressions. In *In Proc. of the 3rd Workshop on Runtime Verification (RV'03), volume 89 of ENTCS*, pages 162–181, 2003.
18. Juan Zamorano, Alejandro Alonso, JoséAntonio Pulido, and JuanAntonio Puente. Implementing execution-time clocks for the ada ravenscar profile. In *Reliable Software Technologies - Ada-Europe 2004*, volume 3063, pages 132–143. 2004.