A Program Logic for Hierarchical and Periodic Real-Time Systems

David Pereira and Luís Miguel Pinho

No Institute Given

Abstract. Compositionality is a key factor in the development of hard real-time systems, essentially because these systems are now large and complex, many time also built from a set of embedded componets which can have distinct scheduling policies.

1 Introduction

Modern hard Real-Time Systems (RTS) are commonly designed and implemented as a a set of embedded components that interact with the external environment is some predefined way. The real-time guarantees of each embedded component may vary depending on the problem that they are expected to solve. In particular, the scheduling policy and resource consumption policies may vary from component to component and the set of all components must satisfy the global system's requirements.

Typically, RTS are designed and developed in a model-driven way: an abstract model of the system is designed and successively refined and test for errors in the specifications. Moreover, implementation is preferably object oriented for better establishing the isolation and dependability among the components under consideration.

2 Hierarchical Scheduling

Hierarchical RTSs are normally subject to analysis in the context of an appropriate *Hierarchical Scheduling Framework* (HSF). In a HSF, a hierarchically designed RTS is seen as a tree of components such that a parent node interacts with its child nodes through an interface $I = (G_S, G_D)$, such that G_S is the resource supply of the parent node to a child node and G_D is the resource demand of the child node. Each node is formally defined as the triple $C = (R_C, S_C, W_C)$, such that R_C is a resource model, S_C is a scheduling algorithm, and W_C is a workload model.

3 Programming Language

In this section we introduce a new and simple programming language specifically crafted to program RTSs and also to facilitate their formal verification.

This new language, which we named IMP-RT, extends the simple imperative programming language IMP [] with hierarchical components, tasks, and statements that interact with execution times. The motivation behind the design of IMP-RT is to provide better programming language support for the implementation of hierarchical hard RTSs, *i.e.*, RTSs where a component is seen as a node in a tree who provides resource access to its child nodes. The child node of a component can either be a sub-component or a task.

Each component is characterized by some real-time properties, namely, it scheduling policy, its period and deadline, and optionally its worst case execution time (WCET). The tasks are characterized similarly, implement some sequential code, and are scheduled according to the policy of the parent component. The syntax of a component is given by the following BNF grammar:

$$C \ ::= \ \mathsf{component}\, name\, \mathsf{with}\, decls\ specs\, \mathsf{is}\, [C_1 \cdots C_n]\ T_1 \cdots T_m\, \mathsf{end}\, name,$$

where *name* is the component's identifier, *decls* is a set of local variable declarations, *specs* is a set of logical and non-functional specifications, C_i are C's sub-components (possibly none), and T_j is a task. Once all the top-level components are completed, we aggregate them to form a RTS. The syntax is the following:

$$Sys \ ::= \ \mathsf{system}\, name\, \mathsf{with}\, specs_{Sys}\, \mathsf{is}\, C_1 \ \cdots \ C_n\, \mathsf{end}\, name,$$

where $spec_{Sys}$ is a set of specifications that include the global scheduling policy with respect to all the top-level components. Note that this way of structuring RTSs is fully compliant with the intuition behind the HSFs prosed in [].

Each component must contain, at least, a task. A task is the entity that is responsible for collecting and computing data, as well as serving as an abstract interface with the external environment (captured by sensor and actuator hardware embedded components). The syntax of tasks is the following:

$$T \ ::= \ \mathsf{task}\, name\, \mathsf{with}\, decls\ specs\, \mathsf{is}\, Stmt\, \mathsf{end}\, name;$$

The non-terminals *decls*, *specs*, and *stms* describe the syntactical structure of variable declarations, specifications, and programming statements, respectively.

component-based, imperative programming language for the programming and specification of RTSs. A RT-IMP program or *system* P is a collection of components C_i, a scheduling algorithm S_P, and optionaly a set of global variables G_P. The BNF grammar rules for the top-level structure of a IMP-RT program is the following:

Each component C_i must specify its deadline d_i and period p_i. Optionally, it can explicitely establish its WCET w_i. The body of any task C_i is made of local declarations and imperative, sequential code. No concurrency is allowed in the scope of component. Syntactically, we write a component as follows:

The non-terminal $decl_{C_i}$ refers to local variable declarations, and $spec_{C_i}$ refers to a collection of specifications. The language of both these constructs of IMP-RT is going to be formally introduced below.

3.1 Types

In RT-IMP we consider a simple type discipline that include Booleans, natural numbers, integers, floating-point, strings, and record values. The kind of scenarios where programs written in RT-IMP are expected to act are highly complex, so basic safety concern

3.2 The Sequential Fragment of RT-IMP

The behavior of a component C_i is implemented as a sequential piece of code. The main difference is that we consider time-consulting and delay statements explicitly in the language. Another difference, is that our sequential fragment considers only terminating loops, and so, the syntactical structure of the loops statement includes an explicit well-founded termination measure.

$$\begin{aligned} stmt ::= {} & \mathsf{skip} \mid x := e \mid \mathsf{time} \mid \mathsf{delay}\, e \mid stmt; stmt \\ \mid {} & \mathsf{if}\, b\, \mathsf{then}\, stmt\, \mathsf{else}\, stmt \mid \mathsf{while}\, b\, \mathsf{var}\, m\, \mathsf{do}\, stmt\, \mathsf{done} \end{aligned}$$

The non-terminals e and b refer to, respectively, arithmetic expressions and Boolean expressions, and are defined as usual. The non-terminal m refers to a computable function that evaluates to natural numbers, and that must return a smaller value at each subsequent iteration of the loop. The statements delay and time interact with the time clock of the complete program. The former delays the execution of the code until the evaluated value of the expression e is reached, whereas the latter simply returns the current point-in-time of the system's execution.

4 Program Logic

The assertion language fragment of RT-IMP includes the usual notions of preconditions, postconditions, and loop-invariants. On top of these, it the language considers constructions that refer to non-functional primitives, such as the program's execution time, and also allows to express standard real-time scheduling parameters such as a task's deadline, period, and worst-execution time.

The base of the language is the classic Hoare logic for sequential programs, that is, a first-order language. We adopt $x, y, x_0, y_0, \ldots$ to refer to first-order variables, and the special variable *now* to denote the system's global clock. The constants c_i denote the i-th component of the system, while d_i, p_i, and w_i denote, respectivelym i-th task's deadline, period, and worst-case execution time. The interpretation of variables is performed over natural numbers.

4.1 Running Example

As a running example to exhibit the characteristics and expressive power of RT-IMP we choose Burns *et. al.*'s Pump Control System (PCS) []. Each building-block of the PCS will be encoded into a RT-IMP component, with the entities

responsible for interacting with the external environment abstracted to special functions. These include the $wl(t)$ and $ml(t)$ denoting the water and methane levels of the mine at t time. The functions pump_on and pump_off denote turning on and off the PCS's water pump, respectively, and the functions is_on and is_off determine the current state of the pump. Furthermore, we use *hwl*, *hll* to denote the water's highest level and lowest levels allowed for proper execution of the system, and we also use *critical*to refer to the methane level from where danger becomes eminent.

We start by defining the environment monitoring component. It essentially consists in reading water and methane levels at some predefined constant rate.

```
component Env_Monitor with
is
 task Read_Levels with
  period => 60 ;
  wcet => 5
 is
  wl := read_water_level;
  ml := read_methane_level;
 end Read_Levels;
end Env_Monitor;
```

The monitor's code establishes a rate of five time units to execute, and it is expected to compute in at most two time units. The terms `read_water_level` and `read_methane_level` represent statements that interact with the external environment, doing what their names suggest. Their WCET is expected to be explicitly defined as globals of the program.

Next, follows the pump's controller code.

```
component Pump
is
 task Controller with
  deadline => 1;
 is
  if(wl > whl and ml < critical and is_off) then
   pump_on;
  elif(wl ≤ wll and is_on) then
   pump_off;
  elif

 end Controller;
```

5 Proof System and Verification Conditions

6 $\mathcal{L}$ in Practice

7 Conclusions and Future Work

In this paper we introduced a rich language for the specification and programming of hard real-time systems in a time- and resource-aware hierarquical and compositional way. Programs specified and written in our language are composed of components, and each of those components is made of a set of tasks governed by some prescribed hard real-time scheduling policy. Moreover, there might by different scheduling policies associated to components.