An Executable Semantics for Synchronous Task Graphs: From SDRT to Ada

Morteza Mohaqeqi(✉), Jakaria Abdullah, and Wang Yi

Uppsala University, Uppsala, Sweden
{morteza.mohaqeqi,jakaria.abdullah,yi}@it.uu.se

Abstract. We study a graph-based real-time task model in which inter-task synchronization can be specified through a rendezvous mechanism. Previously, efficient methods have been proposed for timing analysis of the corresponding task sets. In this paper, we first formally specify an operational semantics for the model. Next, we describe a method for Ada code generation for a set of such task graphs. We also specify extensions of the approach to cover a notion of broadcasting, as well as global inter-release separation time of real-time jobs. We have implemented the proposed method in a graphical tool which facilitates a model-based design and implementation of real-time software.

Keywords: Automated code generation · Ada programming language · The synchronous digraph real-time task model · Schedulability analysis

1 Introduction

Safe, accurate, and efficient timing analysis of real-time applications is an important requirement in safety-critical embedded systems design. To achieve this goal, having formal models which can specify the structure and behavior of the software in an expressive way is essential. At the same time, the models utilized must be of a suitable level of abstraction, through avoiding unnecessary technical details, such that the analysis can be carried out in a reasonable time.

In the past, several models have been proposed to specify real-time workloads, ranging from the periodic and sporadic task models [7] to more complex graph-based ones [4,9,11]. These models are used to describe the computational workload, and accordingly, to perform timing analysis of the software application. While many studies concern theoretical methods for analyzing the task sets specified by these models, less attention has been paid to implementation issues. However, in practice, a designer needs to have a clear definition of the relation between modeling components and the corresponding implementation building blocks. Having such a knowledge, which helps in (automatically) generating executable programs from a set of formal models, is specially important in the model-based development paradigm [6].

In this work, we consider one of the most expressive real-time task models, i.e., Synchronous Digraph Real-Time (SDRT) [9]. SDRT extends the Digraph

© Springer International Publishing AG 2017
J. Blieberger and M. Bader (Eds.): Ada-Europe 2017, LNCS 10300, pp. 137–152, 2017.
DOI: 10.1007/978-3-319-60588-3_9

Real-Time (DRT) task model [11] by introducing inter-task synchronization through a rendezvous mechanism. Efficient analysis methods for dynamic- and fixed-priority scheduling of DRT tasks, and also for fixed-priority scheduling of SDRT tasks, have been previously proposed [9,12]. In this work, we employ a slightly extended version of SDRT and study automatic Ada code generation for the model. We opt for the Ada programming language [8] since the language primitives, specially the provided notions of task and synchronization, match very well with the SDRT task semantics.

As it will be demonstrated, the SDRT task model allows non-deterministic behavior. We attempt to resolve the non-determinism by confining the possible behaviors of an SDRT task. The goal is then to produce source code implementing the behavior such that the timing analyses (performed on (S)DRT task sets [9,11]) remain valid. In summary, the key contributions include:

- Defining a formal operational semantics for SDRT;
- Proposing a code generation approach to implement the specified semantics;
- Showing how to model global inter-release time constraints using SDRT.

In the rest of the paper, we first review related work. The syntax, as well as operational semantics, of SDRT is formally defined in Sect. 3. We present our approach for implementation of the SDRT behavior using the Ada programming language in Sects. 4 and 5. Some extensions of the method are demonstrated in Sect. 6. Concluding remarks and future work are presented in Sect. 7.

2 Related Work

Implementation of real-time tasks using the Ada programming language has been recently studied by Real et al. [10] with an emphasis on preserving release jitter constraints. For this goal, it is proposed to implement jitter-sensitive tasks in a time-triggered manner, running in the highest level of priority, combined with a number of priority-scheduled jitter-tolerant tasks. A given time-triggered plan is managed/scheduled by a *protected type* with the highest priority, which plays the role of a scheduler. Time triggered tasks synchronize with this scheduler via an entry call. In comparison, our approach can also be used to implement the structure of a time-triggered plan with SDRT. Meanwhile, the SDRT model provides more flexibility in the design of a real-time application, through, for instance, allowing to model branches and inter-task synchronizations.

One of the most relevant models to SDRT is task automaton [5] for which a code generation method is proposed in [2]. Compared to task automata, an important feature of SDRT is that the job release times criteria is separated from the application code logic. In terms of the operational semantics, unlike timed automata, SDRT tasks are not allowed to manipulate the clock variables that determine eligibility of a next job for release. In this way, minimum inter-release times are decoupled from the functionality of the jobs. This is a crucial difference which makes the schedulability analysis problem for the (S)DRT model feasible,

in contrast to that of task automata which can be even undecidable in the general case [5]. The code synthesis algorithm provided for task automata in [2] suggests to manage synchronizations and scheduling events by the generated application code. In addition, the implementation of the method (integrated in the TIMES tool [3]) is platform dependent. In contrast, we leverage Ada's primitives, including the synchronization mechanism, which inherently match with the SDRT semantics. This leads to simpler and more intuitive codes. Furthermore, the generated code is hardware independent.

A first attempt to generate Ada code from SDRT models has been carried out by Abdullah et al. [1]. Compared to that work, in this paper we provide a formal operational semantics for the model, and also cover a complete semantics of SDRT including conditional branching. Moreover, we present a technique to model/implement end-to-end inter-release separation times using the SDRT synchronization mechanism.

3 Real-Time Task Graphs with Synchronization

In this work, we focus on the Synchronous Digraph Real-Time (SDRT) [9] task model, which is a graph-based model extended with inter-task synchronizations. Informally, an SDRT task is specified by a directed graph where each path of the graph represents a possible execution path of the task. By means of this model, a task which releases different types of jobs, i.e., with a variable behavior, can be modeled. In what follows, we first present a number of definitions and notations that are used throughout the paper. Then, the syntax and semantics of the SDRT task model are formally defined.

3.1 Notations

We use Σ to denote a set of action labels used to specify inter-task synchronizations. Σ is assumed to contain a *null* action, denoted by $\perp$, which shows the absence of a synchronization. Let Y denote a set of variables. A *valuation* over Y is a function which maps each variable in Y to a value from its domain. Any logical condition over the variables in Y is called a *guard*; the set of all guards is denoted by G. For a given valuation σ and a guard g, both defined over a variable set Y, we write $\sigma \models g$ to denote that σ satisfies g (i.e., the guard is evaluated to True). We also use $\mathbb{N}$ to denote the set of all non-negative integers.

3.2 Syntax

The syntax of an SDRT task is specified using a directed graph. More specifically, considering a set of actions Σ, a set of variables Y, and a set of guards G defined over Y, we define an SDRT task as follows.

Definition 1 (SDRT Task). *An SDRT task is defined as a tuple* (V, v_0, E)*, where*

– *V is a set of vertices,*
– *$v_0 \in V$ is the initial vertex,*
– *$E \subseteq V \times \mathbb{N} \times G \times \Sigma \times V$ is a set of edges.*

Each vertex $v \in V$ represents a *job type* and is associated with a non-negative integer, $d(v)$, as its relative deadline. Each instance of a job type is called a job. A task releases a (possibly infinite) sequence of jobs according to the constraints specified by edges. Intuitively, an edge $(v_i, p, g, a, v_j) \in E$ indicates that if the latest job of the task has been released at time t_0 and is of type v_i, and also the guard g is satisfied after the completion of the job, then the task can synchronize on the action a at any time $t \geq t_0 + p$, releasing a new job of type v_j. Based on this meaning, p is called the *minimum inter-release* time. The precise semantics of an SDRT task is presented in the next subsection.

Here, we assume that exactly two tasks are involved in each synchronization, that is, there is no action $a \in \Sigma$ appearing on the edges of more than two tasks. We later relax this restriction in Sect. 6. In addition, throughout this paper, we assume constrained deadlines. This means that, for any arbitrary vertex $v \in V$, it holds that $d(v) \leq p$ for all p for which $\exists(v, p, g, a, u) \in E$.

It is worth noting that the syntax of an SDRT task has been originally defined, in [9], in a more abstract level. In this work, as we are dealing with code generation, we consider a more concrete definition. Particularly, in the current work, the task syntax is supposed to specify an initial vertex, as well as guards on edges. As this specification only restricts the behavior of a task, the existing timing analysis methods still provide a safe (although maybe a pessimistic) result.

3.3 Operational Semantics

We first make a set of assumptions based on which the SDRT semantics will be defined.

Assumption 1 (Local Access to the Variables). *Each task's variables can be accessed and updated only by the task itself (and by none of the other tasks). As a result, between the finish time of a job and the start of the next one, the value of the guards are not changed.*

We also assume that the functionality of a job of type v is specified by a function $F_v(.)$ which manipulates the task's variables. More specifically, given a current valuation σ of the task variables, $F_v(\sigma)$ denotes the valuation of the variables immediately after the execution of the job. Further, given a set of n tasks, we assume that the first job of the i-th task, for $1 \leq i \leq n$, initializes task's variables to a valuation $\sigma_{0,i}$.

The operational semantics of the SDRT task model is defined using a labeled transition system. Let $\{(V_1, v_{0,1}, E_1), \ldots, (V_n, v_{0,n}, E_n)\}$ denote a set of n SDRT tasks. A semantic state of the system is then defined as a triple $(\bar{v}, \bar{\sigma}, \bar{c})$, where

– $\bar{v} = \langle v_1, \ldots, v_n \rangle$, with $v_i \in V_i$, for $1 \leq i \leq n$, is a vector of vertices (job types), which keeps track of the type of the latest released jobs,

- $\bar{\sigma} = \langle \sigma_1, \ldots, \sigma_n \rangle$ is a vector of valuations, where σ_i denotes a valuation over the variables of the i-th task,
- $\bar{c} = \langle c_1, \ldots, c_n \rangle$ denotes a vector of n non-negative integers, referred to as *clock* variables. The value of c_i shows the time which has passed after the release of the last job of the i-th task.

Before defining the transition rules, we introduce a number of notations. Take an arbitrary vector of job types $\bar{v}$. By $\bar{v}[v_i/v_i']$, we denote the vector of job types obtained by replacing v_i with v_i' in $\bar{v}$, while the other entries of $\bar{v}$ remain unchanged. Additionally, for a vector of clocks $\bar{c}$ and a set of clock variables r, $\bar{c}[r \mapsto 0]$ denotes the vector derived from $\bar{c}$ after resetting those clock variables of $\bar{c}$ that are in r to 0. Also, for a clock vector $\bar{c} = \langle c_1, \ldots, c_n \rangle$, we define $\bar{c} + 1$ as $\bar{c}$ after incrementing each entry by one, that is $\bar{c} + 1 \doteq \langle c_1 + 1, \ldots, c_n + 1 \rangle$. Additionally, for a valuation $\bar{\sigma} = \langle \sigma_1, \ldots, \sigma_n \rangle$ and a job type $v_i \in V_i$, we define $F_{v_i}(\bar{\sigma}) \doteq \langle \sigma_1', \ldots, \sigma_n' \rangle$, where $\sigma_i' = F_{v_i}(\sigma_i)$, and $\sigma_j' = \sigma_j$ for $j \neq i$. Using these definitions, we now present the SDRT semantics.

Definition 2 (SDRT Operational Semantics). *Consider a set of SDRT tasks $\tau = \{(V_1, v_{0,1}, E_1), \ldots, (V_n, v_{0,n}, E_n)\}$. Also, define $\bar{v}_0 = \langle v_{0,1}, \ldots, v_{0,n} \rangle$, $\bar{\sigma}_0 = \langle \sigma_{0,1}, \ldots, \sigma_{0,n} \rangle$, and $\bar{c}_0 = \langle 0, \ldots, 0 \rangle$. The operational semantics of τ is defined by a labeled transition system with an initial state of $(\bar{v}_0, \bar{\sigma}_0, \bar{c}_0)$, and two types of transitions:*

1. *Delay transitions, denoted by $(\bar{v}, \bar{\sigma}, \bar{c}) \xrightarrow{\delta} (\bar{v}, \bar{\sigma}, \bar{c} + 1)$, which represent the progress of time;*
2. *Release transitions, which are associated with the release of new jobs, and include two types:*
   - $(\bar{v}, \bar{\sigma}, \bar{c}) \xrightarrow{\perp} \big(\bar{v}[v_i/v_i'], F_{v_i'}(\bar{\sigma}), \bar{c}[\{c_i\} \mapsto 0]\big)$ *if* $\exists (v_i, p, g, \perp, v_i') \in E_i$ *such that* $p \leq c_i$ *and* $\sigma_i \models g$,[1]
   - $(\bar{v}, \bar{\sigma}, \bar{c}) \xrightarrow{a} \Big(\bar{v}[v_i/v_i'][v_j/v_j'], F_{v_j'}(F_{v_i'}(\bar{\sigma})), \bar{c}[\{c_i, c_j\} \mapsto 0]\Big)$ *if there exist edges* $(v_i, p_1, g_1, a, v_i') \in E_i$ *and* $(v_j, p_2, g_2, a, v_j') \in E_j$ *with* $a \neq \perp$ *and* $i \neq j$ *such that* $p_1 \leq c_i$, $p_2 \leq c_j$, $\sigma_i \models g_1$, *and* $\sigma_j \models g_2$.

In this definition, the release transition rules are written assuming that a job takes its effect on the task variables immediately after its release (by applying the function $F_v()$ with no delay), while in practice, it would take some duration to execute the job. Nonetheless, this does not compromise the correctness of the semantics. The reason is that, as we consider constrained deadlines, the execution of a job is always finished before the corresponding minimum inter-release times are passed, given that the job meets its deadline. As a result, the guard conditions, which may depend on the task variables, are evaluated only after the job is completed, and its influence on the variables have taken place. Therefore, the variables are not used before the completion of the job, and thus, it does not matter when they are updated (i.e., at the beginning, or at any time

[1] Recall that c_i and σ_i denote the i-th entries of $\bar{c}$ and $\bar{\sigma}$, respectively.

during the execution of the job). Based on this, we can also argue that, the defined initial state corresponds to the instant exactly after the first job of each task has been released and also taken its effect.

We point out that our focus is on the job release pattern of an SDRT task set. Hence, in the system state, we do not keep track of the amount of the executed workload of a job. Nevertheless, the defined semantics truly reflects the behavior of the task set, from a release time perspective, as long as no deadline is missed.

Based on the original definition of SDRT [9] (and also DRT [11]), an edge can be taken, and the corresponding job can be released, at any time after the specified minimum inter-release time is passed (given that the other conditions are met). This entails a non-deterministic release-time, while for the implementation, we need to determine release times deterministically. We resolve this issue using the so-called *maximal progress* assumption [2]. According to this, a job is supposed to be released as soon as possible. In terms of the specified transition system, this assumption is expressed as follows.

Assumption 2 (Maximal Progress). *In the specified transition system in Definition 2, whenever there are both delay transition and release transition(s) doable, the system takes the release transition(s) first.*

The presented operational semantics provides a basis for converting an SDRT task set to an executable code. As code generation for the branching structures plays a major role in implementing an SDRT task, we treat it separately in Sect. 4. Next, in Sect. 5, we present our implementation approach for the whole task graph.

4 Code Generation for Branching Structures

A branching structure can be specified in SDRT by a vertex with multiple outgoing edges. To decide which edge must be taken, the program needs to consider the respective minimum inter-release times, guards, and also the synchronization actions. In this section, after reviewing a number of assumptions, we demonstrate our approach to implement the guard, minimum inter-release time, and synchronization criteria of a set of edges comprising a branch. We exploit the *rendezvous* mechanism of Ada for this goal. Then, we present an algorithm for implementing the complete semantics of such structures.

4.1 Assumptions

In order to follow the semantics of the Ada rendezvous, which is used for inter-task synchronization, we assume that the set of synchronization actions Σ contains two types of actions: any action a is either a *sending* action, denoted by $a!$, or a *receiving* action, denoted by $a?$. As will be seen, when generating source code for a task, sending actions are mapped to (implemented by) an **entry call**, while receiving actions are mapped to the **accept** statement of the Ada rendezvous.

While Ada provides a mechanism for a conditional *accept* (within a `select` block), there is no analogous structure for conditional *entry calls*. Hence, we need to slightly change the semantics of SDRT to comply with this restriction. For this purpose, when the guard of an edge with a sending action is satisfied and the associated minimum inter-release time is also passed, we will choose that edge to be taken (although not immediately if the receiving task is not ready at the moment), without checking the other edges any more. To formalize this, consider an arbitrary edge $e = (v, p, g, a, u)$, and an edge $e' = (v, p', g', b!, u')$ with a sending action. Edge e' is said to be *enabled* before e if $p' < p$ and g' is satisfied (irrespective of whether the rendezvous on b can be done at the moment). Given this definition, the release transition rules in Definition 2 are rewritten as:

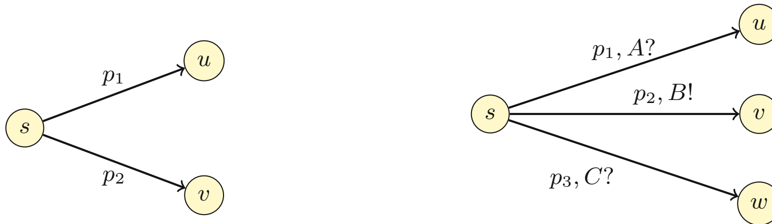
- $(\bar{v}, \bar{\sigma}, \bar{c}) \xrightarrow{\perp} \left(\bar{v}[v_i/v'_i], F_{v'_i}(\bar{\sigma}), \bar{c}[\{c_i\} \mapsto 0]\right)$ if $\exists e = (v_i, p, g, \perp, v'_i) \in E_i$ such that $p \leq c_i$ and $\sigma_i \models g$, and there exists no edge outgoing from v_i in E_i with a sending action which is enabled before e;
- $(\bar{v}, \bar{\sigma}, \bar{c}) \xrightarrow{a} \left(\bar{v}[v_i/v'_i][v_j/v'_j], F_{v'_j}(F_{v'_i}(\bar{\sigma})), \bar{c}[\{c_i, c_j\} \mapsto 0]\right)$ if there exist edges $e_i = (v_i, p_1, g_1, a?, v'_i) \in E_i$ and $e_j = (v_j, p_2, g_2, a!, v'_j) \in E_j$ with $a \neq \perp$ and $i \neq j$ such that $p_1 \leq c_i$, $p_2 \leq c_j$, $\sigma_i \models g_1$, and $\sigma_j \models g_2$, and there exists no edge with a sending action outgoing from v_i in E_i enabled before e_i and also no such an edge from v_j in E_j enabled before e_j.

4.2 Realizing Basic Blocks

In order to conform with the maximal progress assumption (Assumption 2), the implemented task needs to be notified as soon as a *release* transition becomes eligible. According to the specified semantics, release transitions depend on the corresponding guards, minimum inter-release times, and synchronizations. In the following, we specify that how each of these criteria can be checked at runtime to trigger a release transition.

Guards. In edges with no synchronization, or with a sending action, the guard condition can be checked by an `if-then-else` structure. If evaluated to True, the transition will be chosen to take. However, if an edge is related to a receiving action, we will use the "conditional accept" structure of Ada to restrict the synchronization to be done only if the guard is satisfied and the edge with the corresponding sending action is also ready to be fired. This case is elaborated shortly.

Minimum Inter-release Times. To respect a minimum inter-release time between two jobs, we use the `delay until` statement of Ada, which provides a way to wait until a (absolute) time instant. As an example, consider the branching structure shown in Fig. 1a, where p_1 and p_2 denote the minimum inter-release times assuming $p_1 < p_2$. Further, assume g_1 and g_2 to denote the corresponding



(a) A choice with two edges (b) A choice with three edges with synchronization

Fig. 1. Sample branching structures in SDRT.

guards. The Ada code generated for this part of the model is seen in Listing 1.[2] In this example, the release time of the current job, which is of type s, has been assumed to be 0.

```
1    -- After completion of the last released job
2    delay until p1;
3    if g1 then
4        next_state := u;
5        goto loop_start; -- Skipping the rest
6    end if;
7    delay until p2;
8    if g2 then
9        next_state := v;
10       goto loop_start;
11   end if;
```

Listing 1. Implementing the branching structure shown in Fig. 1a

Synchronization. An edge with a receiving action can be fired only if the task sending that action is ready to synchronize. If there are multiple such edges having the required minimum inter-release time passed, the program needs to wait until one of the synchronizations becomes doable. We use the *selective accept* structure to implement this semantics. For example, consider the branch structure shown in Fig. 1b, where p_1, p_2, and p_3 denote the minimum inter-release times, with $p_1 < p_2 < p_3$. Further, let g_1, g_2, and g_3 denote the corresponding guards. The code presented in Listing 2 implements this structure. As seen in Lines 1 to 9, when p_1 expires, the program attempts to evaluate guard g_1, and if satisfied, synchronize on action A. If such a synchronization cannot be accomplished until p_2, then the guard g_2 is checked. If it is satisfied, the program takes the second edge (Lines 10 to 14). Otherwise, synchronization on A is tried again until p_3. If it is not performed by that time, then both the first edge and the third edge are eligible, which are tried using a selective accept block (Lines 20 to 27).

[2] We use `goto` to avoid lengthy and redundant codes. The same logic can be easily implemented without this statement.

```
delay until p1;
select
    when g1 =>
    accept A;
    next_state := u;
    goto loop_start;
or
    delay until p2;
end select;
if g2 then
    Task_2.B;         -- Entry call to Task_2
    next_state := v;
    goto loop_start;
end if;
select
    -- Repetition of the code appeared in Lines 3 to 6
or
    delay until p3;
end select;
select      -- A selective accept
    -- Repetition of the code appeared in  Lines 3 to 6
or
    when g3 =>
    accept C;
    next_state := w;
    goto loop_start;
end select;
```

Listing 2. Ada implementation of the branching structure shown in Fig. 1b.

4.3 Implementation Algorithm for Branching Structures

Our method for generating Ada code for the semantics of a branching structure is shown in Algorithm 1.

In Algorithm 1, the input E is the list of all outgoing edges from a certain vertex, where $E[i]$ denotes the i-th entry of E. Also, $E[i].p$ and $E[i].a$ denote the associated minimum inter-release time and synchronization action, respectively. For simplicity and without loss of generality, in the presented pseudo-code, it is assumed that the latest job has been released at time zero.

The algorithm iterates over the set of edges E. If an edge is not marked with a receiving action, then the decision for taking that edge will be made only based on the guard through the code printed by Lines 7 to 9. Otherwise, the edge is added to the set R. As a result, R contains all edges with a receiving action whose minimum inter-release time has been passed. After examining the edge, if R is empty, then the program needs to just wait until the minimum inter-release time of the next edge (if any) is passed; see Lines 13 to 16. Besides, if R is not empty, i.e., if there are pending receiving actions, the selective accept structure of Ada is used (as shown in Algorithm 2, which is called in Line 18 of Algorithm 1). In this case, the program waits for the first entry call to one of the pending accept statements, until a new edge becomes eligible, if any.

Algorithm 1. Generating Ada code for a branching structure

```
Input: E: List of edges sorted by inter-release times ascendingly.
 1: procedure BranchCode(E)
 2:     n ← |E|                                         ▷ Number of entries in E
 3:     R ← {}                         ▷ Used to keep edges which have a receiving action
 4:     print("delay until " + E[1].p + ";")
 5:     for i ← 1 to n do
 6:         if E[i] is not labeled with a receiving action then
 7:             print("if " + E[i].g + " then ")
 8:             print code for taking edge E[i]
 9:             print("end if;")
10:         else
11:             R ← R ∪ {E[i]}
12:         end if
13:         if R = {} then
14:             if i ≠ n then
15:                 print("delay until " + E[i + 1].p + ";")
16:             end if
17:         else
18:             SelectiveAccept(E, R, i, n);
19:         end if
20:     end for
21: end procedure
```

Algorithm 2. Generating the selective accept code

```
 1: procedure SelectiveAccept(E, R, i, n)
 2:     print("select ")
 3:     print("when " + R[1].g + " => ")
 4:     print("accept " + R[1].a + ";")
 5:     for k ← 2 to |R| do
 6:         print("or ")
 7:         print("when " + R[k].g + " => ")
 8:         print("accept " + R[k].a + ";")
 9:     end for
10:     if i < n then
11:         print("or ")
12:         print("delay until " + E[i + 1].p + ";")
13:     end if
14:     print("end select; ")
15: end procedure
```

5 Implementation of a Task Graph

Each SDRT task graph is implemented as a `task` in Ada, running an infinite loop. Inside the loop, the graph structure is implemented by keeping track of the latest released job, and realizing the branching structures. We demonstrate it through a sample task graph shown in Fig. 2. Minimum inter-release times are assumed

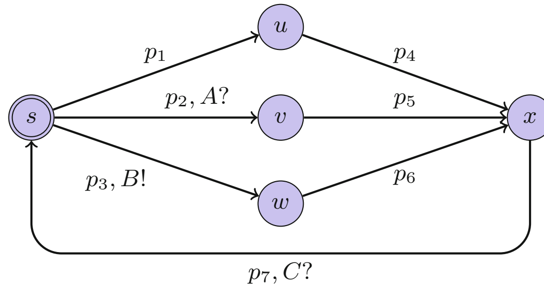


Fig. 2. A sample SDRT task `T1`.

as $p_1 = 100\,\text{ms}$, $p_2 = 200\,\text{ms}$, $p_3 = 500$ ms, and $p_4 = p_5 = p_6 = p_7 = 100\,\text{ms}$. Further, let g_1, g_2, and g_3 denote the guards on edges from s to u, v, and w, respectively. The guard of the other edges is assumed to be always True.

The Ada code realizing this task model is seen in Listing 3. In the task body, first, a type `State` is defined which includes a distinct value for each vertex (Line 14). The variable `Current_State` is defined of this type to store the latest released job of the task. Also, the variable `Last_Release` is defined to keep the release time of the latest released job. Additionally, minimum inter-release times are declared as constants (Lines 17 to 20). The functionality of each job type is also implemented as a procedure (Lines 22 to 31). As seen, the task priority is dynamically changed before and after execution of the job code. We will talk about priority assignment shortly.

```
1   -- Context clauses and pragmas omitted
2  procedure Taskset_1 is
3      ----  Task declaration ----
4      task T1 is          -- A singleton task
5          pragma Priority(System.Priority'Last);
6          entry A;
7          entry C;
8      end T1;
9
10     ----  task body ----
11     task body T1 is
12         ---------- Variable declaration -------
13         T1_prio : System.Any_Priority := 20;  -- Task priority
14         type State is (s, v, w, u, x);
15         Current_State : State := s;   -- The first job
16         Last_Release  : Ada.Real_Time.Time;
17         p1 : constant Time_Span := Milliseconds(100);
18         p2 : constant Time_Span := Milliseconds(200);
19         p3 : constant Time_Span := Milliseconds(500);
20         -- p4, p5, p6, p7 are defined similarly
21
22         -- Procedures for the job types of T1:
23         procedure s_code is
24         begin
25             Ada.Dynamic_Priorities.Set_Priority(T1_prio);
26             -- The code for job type s goes here
27             Ada.Dynamic_Priorities.Set_Priority(System.Priority'Last);
28         end s_code;
29
30         -- Procedures for v, u, w, and x are specified as well
31             ...
```

```
32
33          --------------- Task logic --------------
34          begin
35              Last_Release := Clock;
36              loop
37                  <<T1_loop>>
38                  case Current_State is
39                      when s =>
40                          s_code;
41                          delay until Last_Release + p1;
42                          if g1 then
43                              Current_State := u;
44                              Last_Release := Last_Release + p1;
45                              goto T1_loop;
46                          end if;
47                          delay until Last_Release + p2;
48                          select
49                              when g2 =>
50                              accept A;
51                              Last_Release := Clock;
52                              Current_State := v;
53                              goto T1_loop;
54                          or
55                              delay until Last_Release + p3;
56                          end select;
57                          if g3 then
58                              T2.B;           -- Entry call to task T2
59                              Last_Release := Clock;
60                              Current_State := w;
61                              goto T1_loop;
62                          end if;
63                          select
64                              -- Repetition of the code in Lines 49 to 53
65                          end select;
66                      when u =>
67                          u_code;
68                          delay until Last_Release + p4;
69                          if True then
70                              Current_State := x;
71                              Last_Release := Last_Release + p4;
72                              goto T1_loop;
73                          end if;
74                     -- Similar code is generated for v and w
75                         ...
76                      when x =>
77                          x_code;
78                          delay until Last_Release + p7;
79                          select
80                              when True  =>
81                              accept C;
82                              Last_Release := Clock;
83                              Current_State := s;
84                              goto T1_loop;
85                          end select;
86                  end case;
87              end loop;
88      end T1;
89      --------------------------------------------
90  begin
91      null;
92  end Taskset_1;
```

Listing 3. Ada implementation of the task shown in Fig. 2

In the task implementation, the task first initializes `Last_Release` by the current time (in Line 35), considered as the release time of the first job. Inside the loop, `Current_State` is examined, through a **case** statement, to find the

type of the latest released job. For each job type, Algorithm 1 is employed to implement the respective behavior. For instance, when the job type is s, a branch with three edges must be treated. After the minimum inter-release time p_1 is passed, if g_1 is True, u is selected as the next job. In addition, the current time, which would be equal to `Last_Release` plus p_1, is assigned as its release time (see Lines 41 to 44). If g_1 is not satisfied, the next edge must be tried. For this, after waiting until p_2 is passed, a select statement is executed. If g_2 is evaluated to True and the rendezvous on entry `A` can be done before p_3 is passed, the next job will be of type v (Line 52). In this case, the release time of the job is not calculated by adding p_2 to the previous release time. Instead, it is obtained by reading the current clock value (Line 51). The reason is that, in such a case, the job is released when the synchronization is done, which is determined by the other task involving in the rendezvous. The respective code for other situations is similarly generated, as seen in Lines 57 to 85.

Priority Assignment: An important step in realizing each task is determining the respective priority. For this, we note that, in our implementation, a task consists of two different types of code: codes for controlling the release timings of the jobs, and codes implementing the actual functionality of the jobs as defined by the application. An essential requirement is that the release semantics of a task must not be influenced by the execution of the jobs from other tasks. To respect this, we opt to run the logic controlling release instants of the jobs in the highest priority level. For this purpose, the initial priority of all tasks is set to the highest priority level; see Line 5 in of Listing 3. The priority of a task is then adjusted to its actual (user-defined) priority whenever it wants to execute the functionality of a job. One such dynamic priority adjustment is seen in Lines 25 and 27.

6 Extensions

This section extends our approach to cover a broadcast semantics. Additionally, we describe how an end-to-end inter-release separation time can be modeled by SDRT tasks.

6.1 Broadcasting

Up to now, we have assumed that a synchronization involves no more than two tasks. One can extend the model to include a broadcast semantics as well. In a broadcast synchronization, there may be several tasks with the same receiving action, while there is one task with the corresponding sending action. Whenever the task with the sending action wants to take the respective edge, it will try all the relevant tasks, but in a non-blocking way. For instance, consider a broadcasting on an action A, where two tasks Task1 and Task2 contain the respective receiving action. Then, the task associated with the sending action will execute the following code:

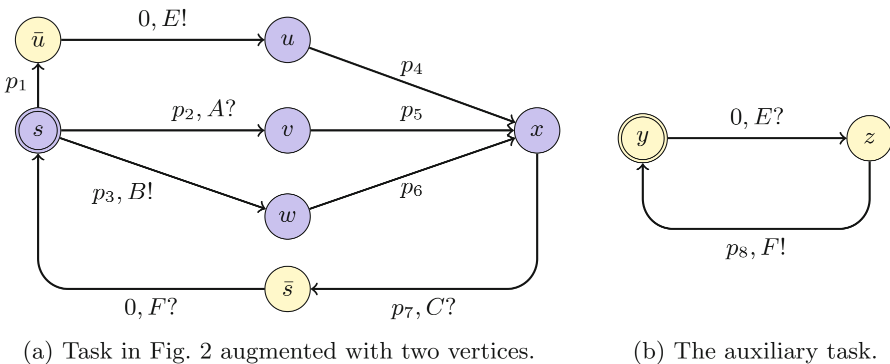
```
select
    Task1.A;
else
    null;
end select;
select
    Task2.A;
else
    null;
end select;
```

The `else` part lets the task continue its progress with no blocking if the other task is not accepting the entry at the moment.

6.2 End-to-End Inter-Release Times

Basically, in an SDRT task, the minimum inter-release time constraint can be specified only between two *consecutive* jobs. However, sometimes it is needed to respect a minimum separation time between the release of two jobs which are not necessarily released successively. As an example, in the task shown in Fig. 2, we may need to add a minimum separation time constraint between any job of type u and any subsequent job of type s. Such a constraint is called an *end-to-end* minimum inter-release separation time. In [12], a method has been proposed to transform a DRT graph with such a constraint to an ordinary DRT task. However, the obtained DRT may contain pseudo-polynomially many number of vertices compared to the original one. Instead, one can use the synchronization mechanism of SDRT to allow putting this constraint with less effort (although the computational complexity of the respective analyses may ultimately be the same).

For instance, in the mentioned example, to preserve a minimum separation time of p_8 between the jobs of type u and subsequent jobs of type s, we can



(a) Task in Fig. 2 augmented with two vertices. (b) The auxiliary task.

Fig. 3. Modifying the task in Fig. 2 to respect an end to end inter-release time separation constraint.

add an auxiliary task with two vertices as seen in Fig. 3b. Also, we augment the DRT task in Fig. 2 with two vertices, namely $\bar{s}$ and $\bar{u}$, seen in Fig. 3a. Whenever a job of type u is released, the task sends a signal, through the action $E!$, to the auxiliary task. On the other side, in order for a job of type s to be released, the task synchronizes on the signal F. According to Fig. 3b, this can be done not earlier than p_8 time units after the release of u's instance. In this way, a job of type s may be released only if the intended delay after the last release of u is observed.

7 Conclusion and Future Work

In this paper we defined an operational semantics for the SDRT task model and provided a method for generating Ada code for this semantics. The method has been implemented in a graphical tool.[3] Also, we discussed extensions of the approach to cover a broadcast synchronization, as well as global and end-to-end inter-release time constraints.

As a future work, we want to formally prove that the provided implementation conforms to the model, i.e., it does not generate a behavior not specified by the SDRT semantics (when neglecting scheduling overheads). Another direction of extending this work is to tackle the model non-determinism. The semantics provided in this work does not specify a deterministic choice in the release of new jobs when more than one are possible at the same time; the actual behavior of the implemented program depends on the Ada run-time system. But, it may be possible to assign a priority to the transitions, and then, utilizing existing mechanisms in Ada, such as `pragma Queuing_Policy`, to preserve orderings enforced by such priorities.

References

1. Abdullah., J., Mohaqeqi, M., Yi, W.: Synthesis of Ada code from graph-based task models. In: 32nd Symposium on Applied Computing, pp. 1466–1471 (2017)
2. Amnell, T., Fersman, E., Pettersson, P., Yi, W., Sun, H.: Code synthesis for timed automata. Nordic J. Comput. **9**(4), 269–300 (2002)
3. Amnell, T., Fersman, E., Mokrushin, P., Pettersson, Yi, W.: TIMES - a tool for modelling and implementation of embedded systems. In: 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 460–464 (2002)
4. Sanjoy, K.B.: The non-cyclic recurring real-time task model. In: Real-Time Systems Symposium (RTSS), pp. 173–182 (2010)
5. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: schedulability, decidability and undecidability. J. Inf. Comput. **205**(8), 1149–1172 (2007)
6. Kim, B., Feng, L., Sokolsky, O., Lee, I.: Platform-specific code generation from platform-independent timed models. In: Real-Time Systems Symposium (RTSS), pp. 75–86 (2015)

[3] The tool is not publicly released at the moment of writing this work. A primary version is available at http://user.it.uu.se/~mormo492/TimesPro.zip.

7. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
8. McCormick, J.W., Singhoff, F., Hugues, J.: Building Parallel, Embedded, and Real-Time Applications with Ada. Cambridge University Press, Cambridge (2011)
9. Mohaqeqi, M., Abdullah, J., Guan, N., Yi, W.: Schedulability analysis of synchronous digraph real-time tasks. In: 28th Euromicro Conference on Real-Time Systems (ECRTS), pp. 176–186 (2016)
10. Real, J., Sáez, S., Crespo, A.: Combining time-triggered plans with priority scheduled task sets. In: 21st Ada-Europe International Conference on Reliable Software Technologies, pp. 195–212 (2016)
11. Stigge, M., Ekberg, P., Guan, N., Yi, W.: The digraph real-time task model. In: 17th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 71–80 (2011)
12. Stigge, M.: Real-time workload models: expressiveness vs. analysis efficiency. Ph.D. thesis, Uppsala University (2014)