Synthesis of Ada Code from Graph-Based Task Models

Jakaria Abdullah
Uppsala University
P.O. Box 337
751 05 Uppsala, Sweden
jakaria.abdullah@it.uu.se

Morteza Mohaqeqi
Uppsala University
P.O. Box 337
751 05 Uppsala, Sweden
morteza.mohaqeqi@it.uu.se

Wang Yi
Uppsala University
P.O. Box 337
751 05 Uppsala, Sweden
wang.yi@it.uu.se

ABSTRACT

Software for safety-critical applications must provide high-confidence behavior through predictable timely executions. The Synchronous Digraph Real-Time (SDRT) task model is a graph-based model for safety-critical software, for which efficient timing analysis techniques exist. In this work, we present a software synthesis method to generate Ada source code from SDRT models verified by timing analysis. We also explore how the expressiveness of SDRT can be utilized in synthesizing real-time simulation code of systems with complex behavior through a heart/pacemaker case study.

CCS Concepts

•Computer systems organization → Embedded software; Real-time system specification;

Keywords

Embedded software synthesis; Graph-based Task model;

1. INTRODUCTION

1.1 Motivation

The development of software for safety-critical applications is a complex process due to the tight requirement of predictable execution beahvior. To ensure it, software designers express timing requirements of the software components in high-level models (often visual) and use timing analysis tools such as MAST [14]. Implementation of such timing analysed components is currently limited to either time-triggered (TT) [13] or event-triggered (ET) [22] real-time tasks with single job type.

Graph-based task models [21] such as SDRT [16] offer features like multiple job types and job release patterns based on a graph structure. We may require several simple TT or ET tasks to implement the complex execution behavior that a graph-based task can exhibit. [21] shows that a variant of SDRT without synchronization feature is one of the most expressive task models for independent real-time tasks. Timing analysis of SDRT tasks is also shown to be efficiently doable for large task sets [16]. However, there is no tool that can generate software from graph-based task models such as SDRT. In this paper, we present a software synthesis technique to generate executable source code from SDRT task models. Our synthesis technique generates source code in Ada [3], which is a well-established programming language for developing safety-critical software.

1.2 Related Work

Exisitng state of the art code generators like Simulink embedded coder [15], SCADE KCG [12] and QGen [4] provide periodic and sporadic task synthesis from Synchronous Finite State Machine (FSM) blocks. Each FSM block has a set of trigger events that, when active, may result in the execution of a set of actions. Each of these events is associated with a period, and it is common for an FSM to have multiple activation events with different periods. To make sure it will not miss any trigger event, the task is executed at the greatest common divisor (GCD) of the periods of its trigger events. This idea is optimized in [18, 11] by generating multiple tasks according to a partitioning of the FSM based on the periods of the trigger events to improve the system schedulability. MAST [14] has an Ada code generator for UML Marte profile [19] that can utilize schedulability analysis of periodic tasks with static and dynamic offsets. In contrast, our focus is not in transforming functional models into simplified real-time tasks but rather to consider expressive graph-based task models for code synthesis that are not periodic.

Timed automata [6] is one of the most expressive task model which is considered for code generation and implementation [2, 5, 7, 23]. In [7], C code is synthesized from a deterministic semantics of timed automata. In [20], timed automata model of a pacemaker-heart system is translated into Simulink Stateflow model and C/C++ code is generated from it using Simulink Coder [15]. In our work, we are generating Ada code from SDRT which is a less expressive workload model compared to timed automata but is amenable to more efficient timing analysis.

The rest of the paper is organized as such, first we briefly introduce the syntax and semantics of the graph-based task model that we use in this paper. Next, we discuss the important aspects of this model in code synthesis followed by the choice of programming language. Then we describe the details of our synthesis algorithm. Finally, we present a case study involving a pacemaker and a random heart model.

SAC 2017, April 03-07, 2017, Marrakech, Morocco
© 2017 ACM. ISBN 978-1-4503-4486-9/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3019612.3019681

2. TASK MODEL

This section specifies the syntax, semantics and timing analysis methods of the Synchronous Digraph Real-Time task model used for code synthesis in this paper.

2.1 Syntax

We consider a real-time application to be composed of a task set $\tau = \{T_1, \ldots, T_N\}$ with N Synchronous Digraph Real-Time (SDRT) [16] tasks. An SDRT task T is specified by a directed graph $G(T) = (V(T), E(T))$, where $V(T)$ and $E(T)$ denote the set of vertices and edges of G. Each vertex $v \in V(T)$ represents a type of real-time job that T can release. Here a real-time job is a piece of recurrent sequential code. A vertex v is labeled with an ordered pair $\langle e(v), d(v)\rangle$ which denotes the worst-case execution-time $e(v)$ and relative deadline $d(v)$ of the corresponding job. Both values are assumed to be positive integers.

The graph structure of $G(T)$ determines the order in which jobs generated by T are released. Each edge (u, v) is labeled with a positive integer $p(u, v)$ denoting the minimum inter-release time between two consecutive job releases. Additionally an edge (u, v) in SDRT may be labeled with an action which is denoted by $a(u, v)$. The actions are used to denote a synchronization between the tasks. Two SDRT tasks T_i and T_j are said to have a synchronization on action s if there exist some edges $(u, v) \in G(T_i)$ and $(u', v') \in G(T_j)$ such that $a(u, v) = s$ and $a(u', v') = s$. To model these actions as rendezvous synchronization primitives we define two types of valid actions. We use an action $s?$ ending with ? to indicate it as a get/accept action in a pairwise rendezvous. At the same time, we use $s!$ ending with ! symbol to denote the corresponding send/call action of $s?$. We use the notation $a(u, v) = []$ to show that an edge (u, v) is not associated with any synchronization. Finally, we assume a job deadline $d(u)$ is bounded by $p(u, v)$ for all outgoing edges (u, v).

2.2 Semantics

We define the semantics of SDRT based on the set of job releases which can be generated by the respective tasks. Job releases of an SDRT task are specified by a *job sequence*. A job sequence together with the scheduling policy and job priorities determine the execution order of the software code modeled by the SDRT tasks.

Definition 2.1 (Job Sequence [16]). *Let (R_i, e_i, v_i) represent release of a job instance of task T, in which R_i, e_i, and v_i denote the job release time, job execution time, and the job type, respectively. A job sequence $\sigma = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$ is generated by T, if $(v_0, v_1, \ldots)$ is a path in the task's graph $G(T)$, and for all $i \geq 0$:*

- $R_{i+1} - R_i \geq p(v_i, v_{i+1})$, *and*
- $e_i \leq e(v_i)$,

A job sequence may be finite or infinite.

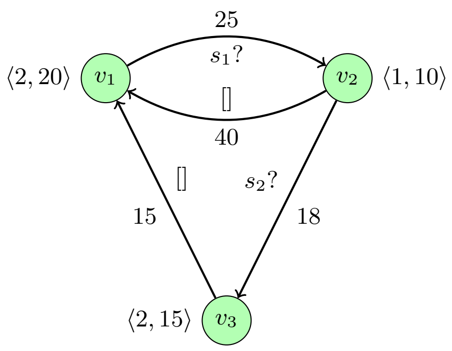
The jobs of two SDRT tasks which are associated to a common synchronization action must be released at the same time. If one of the synchronizing jobs is ready to be released while the other one is not, it will be blocked until the other job becomes ready. This synchronization behavior forces SDRT tasks to generate only *Synchronous Job Sequences.*

Definition 2.2 (Synchronous Job Sequences [16]). *Two job sequences $\sigma = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$ and $\sigma' = [(R'_0, e'_0, v'_0), (R'_1, e'_1, v'_1), \ldots]$ generated by two arbitrary SDRT tasks T_p and T_q are said to be synchronous if for all $i \geq 0, j \geq 0$: $R_i = R'_j$ if $a(v_{i-1}, v_i) = a(v'_{j-1}, v'_j)$.*

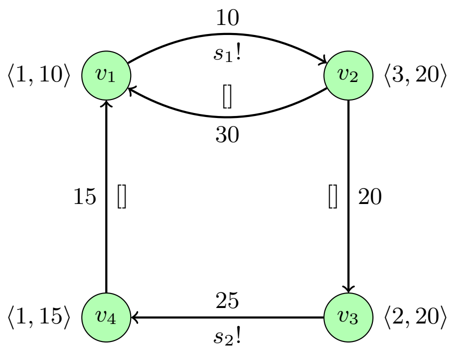
Consider an SDRT task and an arbitrary path $\pi = (v_0, v_1, \ldots, v_l)$ in the respective graph. Then, the *most dense* job sequence generated via π is defined as $\sigma_\pi = [(R_0, e_0, v_0), \ldots, (R_l, e_l, v_l)]$, where

- $R_0 = 0$,
- $R_i = \sum_{j=0}^{i-1} p(v_j, v_{j+1})$, for $0 < i \leq l$,
- $e_i = e(v_i)$, for $0 \leq i \leq l$.

While the most dense job sequence of a path is unique, infinite number of job sequences can be associated to each path. Figure 1 shows two SDRT tasks which have two synchronizations on actions s_1 and s_2. Here, the release of job v_3 of task T_1 is synchronized with the release of job v_4 of task T_2 on action s_2. As a result, v_3 and v_4 must be released at the same time after satisfying their respective minimum job inter-release separation times. The jobs that have no synchronization on release such as v_1 of task T_1, can be released without considering release of any jobs of T_2.



(a) SDRT task T_1



(b) SDRT task T_2

Figure 1: Two SDRT tasks with two synchronizations on actions s_1 and s_2.

2.3 Timing Analysis

A Static Priority (SP) schedulability analysis for SDRT tasks in uniprocessor is presented in [16]. The algorithm uses two abstraction refinement techniques based on over-approximation and under-approximation of the workload to deal with combinatorial explosion of path combinations in SDRT.

3. DESIGN DECISIONS

The goal of code synthesis is to transform a model to an executable code while preserving the execution behavior of the model. As we mentioned in the model semantics, a path in an SDRT task can generate infinite number of job sequences which can be different from each other. In the next subsections we describe challenging model properties of SDRT and our design decisions regarding the implementation of SDRT task using a programming language.

3.1 Non-determinism in the Model

There are two main sources of non-determinism in SDRT models:

3.1.1 Minimum inter-release time

An SDRT task can release a job as soon as the minimum inter-release time after the release of its previous job expires. The idea of minimum inter-release time introduced in Sporadic tasks [17] is to model real-time jobs/interrupts generated by external events. It is not possible to guarantee schedulability of jobs released to handle interrupts that occur arbitrarily frequently. However, when a minimum interval between successive invocations is guaranteed by the environment, schedulability becomes possible [8]. Clearly, the concept of minimum inter-release time is a property of SDRT which is important for timing analysis and should be implemented by the execution of the code generated from this model. An SDRT job can be released immediately after the expiration of the relevant minimum inter-release time to execute like a time-triggered job. To execute like an event-triggered job, an SDRT job can be released by an event any time after the expiration of the minimum inter-release time. This event can be generated by another task of the system or a hardware interrupt.

3.1.2 Branching behavior in job release

The second source of non-determinism in SDRT execution is rooted in the feature of branching in job release. Branching in job release can happen in SDRT task in two ways:

Branching based on synchronization: In this case, the next job to be released is determined based on satisfaction of both timing and synchronization constraints. Here we observe two sub-cases, namely (a) releases of all branch jobs have both timing and synchronization constraints and (b) some of the branch jobs have release constrained by synchronization constraints but not all. An example of situation (b) is depicted for job v_2 in Figure 1(a). Here v_2 has two successor jobs namely v_1 and v_3. The release of v_3 has to be synchronized with respect to action s_2 while v_1 can be released upon expiration of minimum inter-release delay.

Branching without synchronization: The next job to be released can be determined based on the satisfaction of some conditions. These conditions only depend on local variables of the task. As a result, between finish time of a job and start of the next one the value of the conditions is not changed. Therefore, if we evaluate the conditions after a job is finished, that evaluation remains valid until the next job is released. It is assumed that the conditions are checked in an if-then-else structure which, at runtime, determines the path that the program should follow.

```
loop
   --- wait for release event
   Event_Object.Wait(D);
   --- update delay upon release
   Next := Clock + Inter_Rel;
   -- execute job code --
   ...........
   delay until Next;
   -- ensures inter release time
end loop;
```

Figure 2: Ada implementation of a sporadic task

3.2 Choice of Programming Language

From the discussion on SDRT model features it is evident that the programming language to be used for synthesizing SDRT code should support features such as the notion of tasking and absolute delay, rendezvous based inter-task communication and selective branching based on synchronization.

Ada is a programming language that embeds the timing requirements within the program syntax as well as provides library routines to the programmers to express the needs of timing requirement. Ada provides all SDRT features listed above and most of the existing theories of real-time scheduling can be applied to it [9]. Ada uses a runtime system for real-time tasks that can be easily ported to different platforms. However, state of the art implementation of the Ada programs are limited by timing analysis of periodic/sporadic tasks [10] which we intend to extend through our current work. Figure 2 is a sample implementation of a sporadic task in Ada.

In SDRT, release of jobs from two different tasks can be synchronized based on an action. In Ada, the rendezvous is a similar mechanism for controlled synchronization between two tasks. Ada's rendezvous is based on a *client-server* model. A client task requests a rendezvous with a server task by making *entry calls* just as if the server is a protected object. Server tasks indicate willingness to accept a rendezvous on an entry by executing an *accept* statement. For the rendezvous to take place, both the server and the client task must have issued their requests. A task issuing a rendezvous request is blocked until the rendezvous happens. As described earlier, we defined two types of actions valid in SDRT. Among these actions, the send/call action $s!$ directly maps to the Ada rendezvous entry calls. Similarly, the get/accept action $s?$ of SDRT matches the Ada rendezvous *accept* statement. However, in SDRT semantics, a rendezvous is only allowed during a job release. This can be implemented in Ada code by first waiting for a delay and then executing the respective rendezvous operation. For the purpose of simplicity, we only use simple rendezvous (without exchange of parameters) of Ada to implement this behavior. As we see next, rendezvous behaviors can be combined with branching in job releases.

SDRT allows branching based on synchronization. To implement this behavior, we use the *selective accept* feature of Ada. As we mentioned earlier we allow a synchronization action to be either a call or an accept action. In *selective accept* of Ada, branching of code is only allowed using accept action of rendezvous. For the case of entry call synchronization action, we assume it to be executed once the timing requirement of the job is satisfied.

```
loop
  case Current_Job is
     when Job_Name1 =>
         Job_Name1_procedure;
         -- release logic --
     when Job_Name2 =>
         Job_Name2_procedure;
         -- release logic --
         ...............
  end case;
  <<end_label>>
end loop;
```

Figure 3: Ada implementation of an SDRT task

4. CODE SYNTHESIS

4.1 Synthesis algorithm

In our current synthesis algorithm we assume that if an SDRT job release has both timing and synchronization constraints then first it needs to satisfy the timing one. For the case of job release without synchronization we release the job in time-triggered manner and do not allow branching without synchronization. Branching based on user specified conditions will be integrated in our future work. Additionally we fix one of the jobs of each task as an initial job of that task. We use the following steps during Ada code synthesis from an SDRT task:

1. We sort all outgoing edges from a job type according to their inter-release times in a non-decreasing order.
2. At the beginning, the smallest inter-release time is inserted in the code as delay. Subsequently, differences between the inter-relase times are inserted as additional delays.
3. After initial delay if a valid edge contains an accept action, we insert a selective accept (means that now we can accept a synchronization action and release that branch) block with a delay alternative where the delay is the time when it is also possible to release the next branch.
4. These selective accept blocks are iteratively generated until there is a branch which can be immediately released. The immediate release of a job satisfying both its timing and synchronization constraints is similar to the urgent transition of timed automata.
5. For a sending action, a task entry call is added immediately after the code selecting the next job.

We implemented this synthesis algorithm as code generation feature of our tool MASI.

4.2 Code Structure

The structure of the code is very simple. We have an Ada implementation file (adb extension) with a main procedure containing all the SDRT tasks. A task declaration block includes entry definitions related to synchronization. The task body is divided into two parts: declaration and main body. In the declaration part of the task body a local procedure is implemented for each of the job types. The main body of the task contains an infinite loop and the graph structure capturing the release pattern which is encoded in a switch-case statement. Each case of this switch-case statement is related a job of the corresponding task. The execution of a job involves a call to the local procedure containing the job code followed by the code for releasing the next job. A skeleton Ada code of an SDRT task is presented in Figure 3. Figure 4 shows an example implementation of SDRT release logic in Ada generated by our synthesis approach.

```
when Job1 =>
   Job1_Procedure;
   -- Next uses absolute time value
   -- Inter_rel1 is the smallest
   Next := Next + Inter_rel1;
   delay until Next;
  -- Inter_rel2 >= Inter_rel1
   Next := Next + Inter_rel2 -
        Inter_rel1;
   select
         accept Signal1;
         Next_Job := Job1;
         goto end_label;
   or
         delay until Next;
   end select;
   -- Inter_rel3 >= Inter_rel2
   Next := Next + Inter_rel3 -
        Inter_rel2;
   select
         accept Signal1;
         Next_Job := Job1;
         goto end_label;
   or
         accept Signal2;
         Next_Job := Job2;
         goto end_label;
   or
         delay until Next;
   end select;
   -- When there is a valid transition
   -- without accept action
   Next_Job := Job_Name;
   -- entry call for send action
   Callee_Task_Name.entry_name;
```

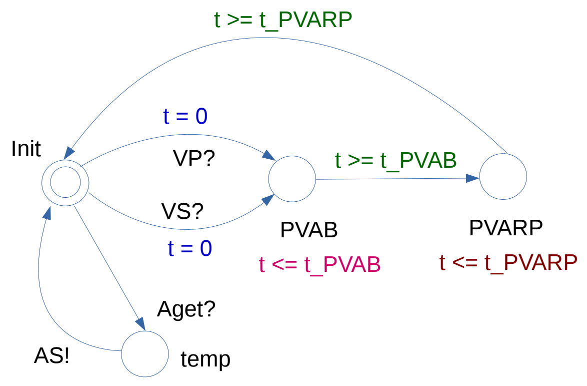
Figure 4: Example of release logic implementation for an Ada SDRT task

5. CASE STUDY

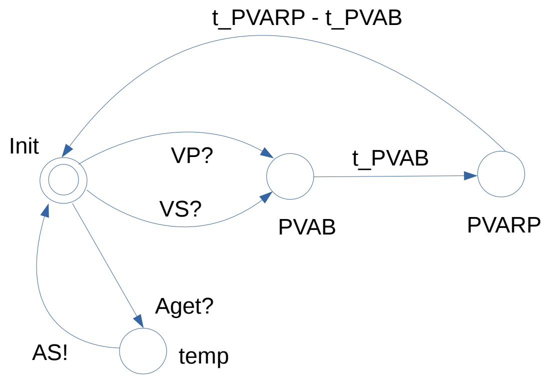
5.1 Pacemaker-Heart Model

We use timed automata representation of a dual chamber DDD pacemaker with a simplified random heart model [20] as a representative case study. The primary function of a pacemaker is to maintain an adequate heart rate. The pacemaker paces the heart when the desired heart rate is below a lower rate limit but it does not pace it above an upper rate limit. In DDD mode, the pacemaker monitors the Atrial and Ventricular events in the heart and generates required pacing actions based on the state of the heart.

We assume a closed-loop system with two main components which are models of a pacemaker and a human heart. The pacemaker receives Aget (Vget) Atrial (Ventricular) events generated by the heart. These are internally recognized as the signals AS (Atrial Sense) and VS (Ventricular Sense) which are used to synchronize different states of the different components of the pacemaker. The pacemaker generates AP (Atrial pacing) and VP (Ventricular pacing) action signals on the heart model.



(a) PVARP Automata



(b) PVARP SDRT

Figure 5: Timed Automata and SDRT models of PVARP Component of the pacemaker

The pacemaker has five software components capturing different timing requirements based on input from the heart. Timed automata model of each of these components can be found in [20]. These automata communicate with each other using broadcast channels and shared variables. Here we briefly describe each of these components.

LRI (Lower Rate Interval) keeps the heart rate above the desired minimum value. If no AS is received after `t_TLRI - t_TAVI` time period following a Ventricular event then AP is delivered by LRI.

AVI (Atrio-Ventricular Interval) maintains the delay between the Atrial and Ventricular activations. If no VS has been sensed within `t_TAVI` after an Atrial event (AS, AP), then AVI will generate VP. The task should maintain an interval of `t_TURI` between two Ventricular events (VP, VS).

PVARP (Post Ventricular Atrial Refractory Period) receives Atrial events (Aget) and detects them as AS for synchronizing other components. With each Ventricular event (VP or VS) there will be a period of `t_PVAB + t_PVARP` when Agets are not recognized as AS. During the period of `t_PVAB` all Agets will be ignored.

VRP (Ventricular Refractory Period) receives Vget events from the heart and recognizes them as VS. After each Ventricular event (VS or VP) the task should wait for a period of `t_TVRP` to generate the next Ventricular event.

URI (Upper Rate Interval) works as a timer to limit Ventricular pacing events. Two consecutive VPs should be separated by an interval of `t_TURI`.

It is possible to validate the correct behavior of the pacemaker by using a random heart model with two software components, one for generating Atrial events and another for generating Ventricular events. Both the components can randomly generate a heart event within a range of valid intervals.

5.2 SDRT Task Models

Timed automata is a modeling formalism that allows nondeterminism and it is the most expressive model for realtime workload [21]. As our case study problem is originally encoded in timed automata, we translate each of these automata to an SDRT task. However, this is not the general procedure for translating timed automata model to SDRT. A close inspection of automata used for modeling pacemaker gives following major differences with the SDRT definitions:

Firstly, each of the automaton in timed automata model uses separate clocks where all the SDRT tasks use a global clock for timing. However, in the pacemaker timed automata models, the state from which a timing constraint is checked is always preceded by the transitions that reset its local clock. This makes our translation to SDRT easy as it uses timing constraints relative to job (or state) release. The only exception is a pair of overlapping timing requirements in the component PVARP where clock reset is absent in a preceding transition. We resolved it by modifying the timing requirement relative to the last job released with a clock reset. Figure 5 shows PVARP automata and its corresponding SDRT task.

Secondly, pacemaker automata communicate with each other using broadcast synchronization while SDRT is only defined for blocking pairwise rendezvous. To implement this we made send part of a broadcast synchronization nonblocking by using the timed entry call feature of Ada. An Ada timed entry call lets a sender specify a maximum delay before achieving rendezvous, failing which the attempted entry call is withdrawn and an alternative sequence of statements is executed. As a result the task which blocks with a receiving action from broadcast will complete rendezvous when it is ready before the sending action. We use 0.1 millisecond delay for the timed entry calls in broadcasting.

Finally, pacemaker automata use guard conditions in states to make urgent transitions. In SDRT, we merge such *clock* <= *contraint* (guard in state) and *clock* >= *constraint* (guard in transition) conditions as a minimum inter-release time *constraint* relative to the release of the job. In code synthesis, it results in an immediate transition to the next job when this timing requirement is true.

As the states in timed automata model of pacemaker have no execution times, jobs of our translated SDRT tasks are also initialized with zero execution times. Many transitions of the timed automata model has only the synchronization constraint but no timing constraint. In those cases, we assume the corresponding SDRT task edge to have a zero minimum inter-release time.

5.3 Pacemaker Code Synthesis

5.3.1 Synthesized code

We synthesized Ada code from a SDRT taskset comprising 7 tasks (5 of them modeling pacemaker components and 2 tasks generating random heart events). The code is gener-

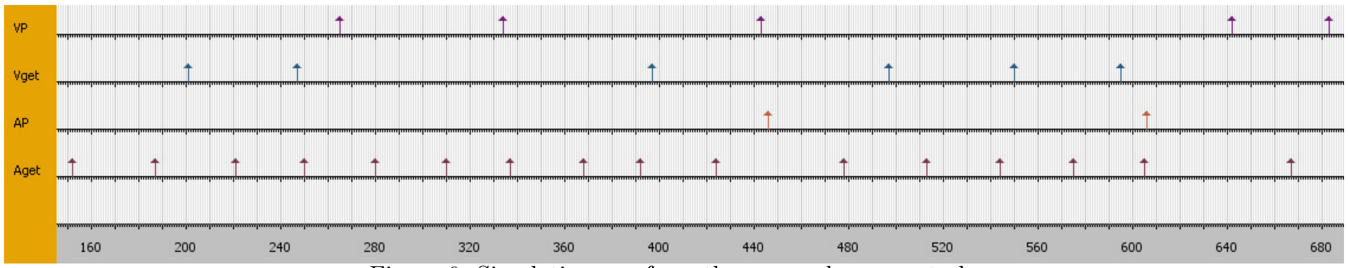


Figure 6: Simulation run from the pacemaker case study

ated for general Ada runtime system which can be compiled to run over any Linux or Windows operating system. We wrote a visualizer in Java to check the heart signals and corresponding response signals of the pacemaker. The visualizer uses a TCP/IP socket to receive information about the events from the running Ada executable. The generated Ada program is instrumented to send release events of different tasks to the visualizer. The executables for this case study is downlodable from [1].

5.3.2 Simulation

For simulation we use timing constraint values `t_TAVI=150`, `t_TLRI=1000`, `t_TPVARP=100`, `t_TVRP=150`, `t_TPVAB=50` and `t_URI=400` used in previous case study. Each of these values are in milliseconds. For generating heart events using a random heart model we use millisecond range $[248, 348]$ and $[435, 535]$ for Atrial and Ventricular events respectively. Finally, we did some simulation experiments using a 2.8 GHz core i7 processor running 64 bit Windows 7 operating system. In the sample runs pacemaker tasks are correctly generating pacing events when signals are missing from the random heart tasks. In Figure 6 we present screenshot of such a run.

6. CONCLUSION

In this paper, we presented a software synthesis algorithm for generating Ada code from the SDRT task model. The synthesis process is demonstrated with a pacemaker-heart case study. In future, we want to extend this synthesis algorithm in several directions. Firstly, branching without synchronization in job release can be integrated with the currently supported release logic. We also plan to develop a formal proof of the correctness for our synthesis algorithm and execute synthesized code on RTOS and bare metal platform to verify the timing analysis of the model. Another future work is to evaluate synthesized code with respect to different code generation metrics such as code footprint.

7. REFERENCES

[1] Experiment package. https://www.dropbox.com/s/snmko28mh1evx9s/PM_ADA.tar.gz?dl=0.
[2] T. Abdellatif, J.Combaz, and J.Sifakis. Model-based implementation of real-time applications. In *Proc. of EMSOFT*, pages 229–238, 2010.
[3] Ada language. http://www.adacore.com/.
[4] QGen. http://www.adacore.com/qgen/.
[5] K. Altisen and S. Tripakis. Implementation of timed automata: An issue of semantics or modeling? In *Proc. of FORMATS*, pages 273–288, 2005.
[6] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
[7] T. Amnell, E. Fersman, P. Pettersson, H. Sun, and W. Yi. Code synthesis for timed automata. *Nordic Journal of Computing*, pages 269–300, 2002.
[8] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. of RTSS*, pages 182–190, 1990.
[9] A. Burns. Why the expressive power of programming languages such as ada is needed for future cyber physical systems. In *Proc. of Ada-Europe*, pages 3–11, 2016.
[10] A. Burns, B. Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proc. of Ada-Europe*, pages 263–275, 1998.
[11] P. Deng, Q. Zhu, M. D. Natale, and H. Zeng. Task synthesis for latency-sensitive synchronous block diagram. In *Proc. of SIES*, pages 112–121, 2014.
[12] KCG. http://www.esterel-technologies.com.
[13] H. Kopetz. *Real-Time Systems-Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
[14] MAST toolset. http://mast.unican.es/.
[15] Simulink embedded coder. https://www.mathworks.com/products/embedded-coder/.
[16] M. Mohaqeqi, J. Abdullah, N. Guan, and W. Yi. Schedulability analysis of synchronous digraph real-time task. In *Proc. of ECRTS*, pages 176–186, 2016.
[17] A. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. MIT Laboratory for Computer Science, 1983.
[18] M. D. Natale and H. Zeng. Task implementation of synchronous finite state machines. In *Proc. of DATE*, pages 206–211, 2012.
[19] MARTE profile. http://www.omg.org/spec/MARTE/.
[20] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. From verification to implementation: A model translation tool and a pacemaker case study. In *Proc. of RTAS*, pages 173–184, 2012.
[21] M. Stigge and W. Yi. Graph-based models for real-time workload: a survey. *Real-Time Systems*, 51(5):602–636, 2015.
[22] P. Tabuada. Event-triggered real-time scheduling of stabilizing control tasks. *IEEE Transactions on Automatic Control*, 52(9):1680–1685, 2007.
[23] M. D. Wulf, L. Doyen, and J.-F. Raskin. Almost asap semantics: From timed models to timed implementations. In *Proc. of HSCC*, pages 296–310, 2004.