Towards a Tool: TIMES-Pro for Modeling, Analysis, Simulation and Implementation of Cyber-Physical Systems

Jakaria Abdullah[1(✉)], Gaoyang Dai[1], Nan Guan[2], Morteza Mohaqeqi[1], and Wang Yi[1]

[1] Uppsala University, Uppsala, Sweden
{jakaria.abdullah,gaoyang.dai,morteza.mohaqeqi,yi}@it.uu.se
[2] Northeastern University, Shenyang, China

Abstract. We consider a Cyber-Physical System (CPS) as a network of components that are either physical plants with continuous behaviors or discrete controllers. To build CPS's in a systematic manner, the TIMES-Pro tool is designed to support modeling, analysis and code generation for real-time simulation and final deployment. In this paper, we present our decisions in designing the modeling language, the tool architecture and features of TIMES-Pro, and also a case study to demonstrate its applicability.

Keywords: Cyber-Physical System · Timing analysis · Real-time simulation · Automated code generation

1 Introduction

Cyber-Physical Systems are systems that contain both discrete components such as digital controllers that generate and react to discrete events according to control laws and continuous components such as physical plants whose behaviors change continuously according to natural laws. Existing design tools for designing such hybrid systems such as Simulink [1] and Modelica [2] have inherent limitation due to the lack of expressiveness in their underlying modeling language and ability for analysis. In this paper, we present an integrated system design tool TIMES-Pro which adopts an expressive yet analytically tractable modeling language based on the Digraph Real-Time (DRT) task model [3–5] to model discrete components and conditional differential equations to model continuous physical components (differential equations with mode switches). For analysis, the continuous components of a system will be abstracted according to a set of predicates of interests, controlling the interaction with the discrete components of the system. DRT models will be used to approximate the continuous components for automated analysis. Our goal is to develop a toolbox supporting modeling and abstraction of both discrete and continuous components, timing analysis and code generation for real-time simulation as well as final deployment on a given execution platform for the discrete components. The rest of the paper is organized as such, first we present different design decisions related to our system design tool. Next, we briefly introduce our modeling language and its existing analysis and code generation supports. Then we present the status of our tool implementation. Finally, we present an intended case study involving a pacemaker and a random heart model.

2 Design Decisions

In this section we summarize design decisions concerning mainly the design of the modeling language as well as the architecture and the features of TIMES-Pro.

Trade-Off Between Expressiveness and Analysis Efficiency. Ideally, the modeling language of a tool should be as expressive as possible to enable faithful modeling of complex system behaviors such as dynamic branching and looping. As the expressiveness of models grows, so grows the complexity of their analysis. For example, timed automata have been found to be the most expressive model for real-time workload [6], but its analysis suffers from state-space explosion problem which makes it impractical to be used in large system design. To study the trade-off, different real-time models have been developed to compromise the expressiveness and analysis efficiency [7].

The DRT task model [3] is a rather expressive model allowing large flexibility to express release patterns accurately by representing each computation task as a directed graph. It generalizes most existing models in real-time scheduling theory [7]. It is shown that the feasibility problem of DRT can be solved in pseudo-polynomial time [3]. Additionally, efficient techniques of exact response-time analysis for DRT task models, for both static-priority and EDF scheduling have been developed using over-approximation of workload abstraction and refinement methods [5]. Finally, DRT model is extended to support rendezvous-style synchronizations with efficient analysis using over-approximation and under-approximation of workload abstractions [8]. Based on availability of these efficient analysis methods we choose DRT as the core modeling language of our design tool.

Separation of Communication and Computation Concerns. The two major aspects of computer systems embedded in a CPS are computation and communication. Computational elements of a system should be independently designed without adherence to any specific communication mechanism. This allows not only, separation of concerns, in system design but also efficient analysis. From our previous work on task automata and scheduling analysis [9], it is known that many decision problems are computationally hard (even undecidable) for systems where feedback is allowed. Allowing communication to occur during the execution of a computation task may easily bring the feedback effects and change the workload of the system dynamically [9]. Obviously, making communication independent of computation may also allow modularity in system design, and flexible and portable design.

We impose this principle by allowing communication to occur only on the release of computational jobs. This means that the released computation job involves no blocking or non-blocking communication primitives and thus communication can not happen during the execution of the job, which makes it easier to analyse the timing properties of a computation job and also the global timing properties of a system when scheduling is involved.

Functional Correctness Independent of Non-functional Behavior. The functional correctness of a system should be maintained during design-space exploration for satisfying the non-functional requirements. For example, changing the execution time of a task should not change its output or its logical correctness. This sounds a simple principle to implement if only the functional correctness of a task is concerned. On the system level, this is a challenging problem. For example, a functionality of a system is implemented by the execution of a number of tasks. The system designer should make sure that the execution order of tasks by the scheduler will not change the functionality. Technically, this requires that the scheduling policies adopted by the scheduler should ensure the functional correctness implied by system-level global invariants.

System Development in a Simulated Environment. A popular engineering technique to validate or certify CPS is emulation. Emulation, popularly known as hardware-in-the-loop simulation, is used to validate controllers by running them in closed-loop with the actual plant. However, in many cases the actual plant is not available for emulation. Firstly, the plant may be a hardware which is developed at the same time. Secondly, actual plant may be too sensitive and can not tolerate an error during simulation (such as human organs). Finally, construction of the actual plant may be too expensive for the test of a prototype of concept. To encounter these deficiencies, we decide that our system design tool should provide simulated environment using realistic but approximated model of the actual plant.

Here the challenge lies in modeling the continuous semantics of a physical process using discrete software so that all important plant behavior necessary for the simulation can be generated. Current practice of using numerical solvers for evaluating continuous state is either too slow or too complex for any real-time simulation. To counter this, we tend to explore computationally efficient approximation techniques for solving differential equations which we can model using only software components. Our final goal in this regard is to generate code of plant to allow real-time co-simulation.

Analysis Based on Abstraction Refinement Techniques. Analysis of complex system behavior is computationally challenging as possible combinations of concurrent component behaviors grow exponentially with the number of active system components. However, for analysis of many properties such as non-functional properties (e.g. schedulability) partial orders may be derived for the search space, preserving the properties of interests. Thus a hierarchy of abstractions may be generated and systematically evaluated using different levels of the abstractions until an acceptable solution is found. This is the fundamental behind combinatorial abstraction refinement approach [5] which is generalized in [10,11]. We intend to use this abstraction refinement framework for different analysis problems of our design tool.

3 Modeling Language

This section introduces the modeling language of our tool. In particular, we describe the model used to represent independent real-time tasks and its extension with inter-task synchronization.

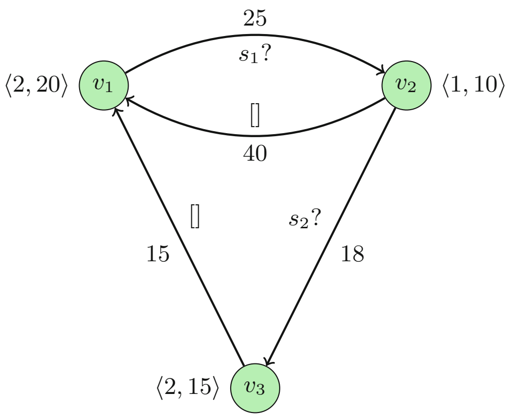
3.1 Task Model

The core of our modeling language is the Digraph Real-Time (DRT) task model [3]. A DRT task T is represented by a *directed graph* $G(T)$ with vertex and edge labels. Each vertex $v \in G(T)$ 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 worst-case execution time $e(v)$ and relative deadline $d(v)$ of the corresponding job. Both values are assumed to be positive integers.

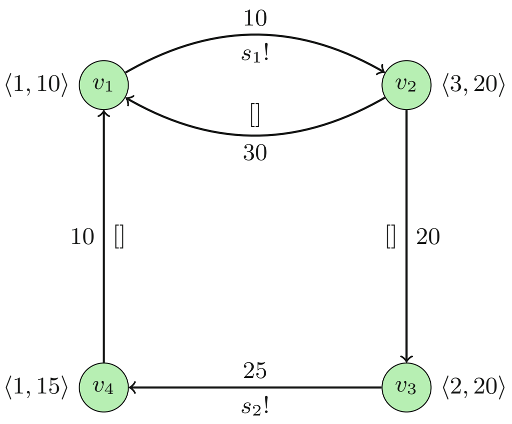
The graph structure of $G(T)$ denotes the order in which jobs generated by T is released. Each edge (u, v) is labeled with a positive integer $p(u, v)$ denoting the minimum job inter-release separation time. We assume a job deadline $d(u)$ is bounded by the minimal of $p(u, v)$ for all outgoing edges (u, v). Finally, we describe a system with a DRT task set $\tau = \{T_1, \ldots, T_N\}$.

We assume the execution of each DRT task to be independent of each other. While DRT tasks can generate independent real-time jobs of a system, in reality many systems contain jobs with inter-task dependencies. To support such inter-task synchronization requirements we extend DRT task model to Synchronous Digraph Real-Time (SDRT) task model [8]. An SDRT task has the same syntax as a DRT task, except that an edge (u, v) may be labeled with an action $a(u, v)$. The actions are used to model synchronization among tasks. Two SDRT tasks T_1 and T_2 are said to have a synchronization on action s if there exist some edges $(u, v) \in G(T_1)$ and $(u', v') \in G(T_2)$ such that $a(u, v) = s$ and $a(u', v') = s$. To model these actions as rendezvous synchronization primitives of programming language we define two types of valid actions. We use $s?$ ending with ? to represent a get/accept action in a pairwise rendezvous. At the same time, we use $s!$ ending with ! to denote the corresponding send/call action of $s?$. $a(u, v) = []$ means that an edge (u, v) is not associated with any synchronization.

In a synchronous execution, the jobs of two SDRT tasks 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, the former one will be blocked until the latter one becomes ready. This synchronization behavior is a special case of rendezvous synchronization where synchronization only happens when two synchronous jobs release together.



(a) SDRT task T1



(b) SDRT task T2

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

Example 1. 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 $T1$ is synchronized with the release of job v_4 of task $T2$ 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 such as v_1 of task $T1$ can be released without considering release of any jobs of $T2$.

3.2 System Model

A CPS system model contains components which model software, hardware and the surrounding physical environment. A software or hardware component may be modeled with a set of discrete states and corresponding transitions among them. Continuous states of a physical component is usually expressed using differential equations. A simple way to model such physical component is to adopt a discrete time-step based approach where the continuous behavior is sampled every time-step of length δ. The granularity of such time-step δ is chosen according to the nature of modeled system and the differential equations involved.

A major challenge in CPS simulation is to integrate hardware, software, and physical components so that their combined system behavior conforms to the reality. The heterogeneity of their behaviors makes this integration difficult as a discrete-time component may need to communicate with a continuous-time component. A way to tackle this problem is to use assertions in component interfaces that confirm certain component behaviors. These types of assertions establish a clear interface of the component with precise obligations for caller, callee and environment. This idea is similar to Design-by-contract [12] which is a software engineering technique that exploits runtime assertions to define precise verifiable interface specifications with so-called invariants and pre- and post-conditions.

A system model S in TIMES-Pro is a set of interacting components $C_1, C_2, \ldots, C_N$. Each component C_i is described by SDRT tasks with timing and synchronization constraints. Tasks in two different components can be connected/glued by a common synchronization action. Direction of communication between two connected components is defined based on the type of inter-component synchronization actions. This direction is from the component with "send" action to the components with "accept" actions. We allow two types of inter-component connections using synchronization in TIMES-Pro. In a *conditional* (or *branching*) connection, a component connects with several other components using the same synchronization action. As a result, communication can only happen with one component at a time, among those are ready to communicate. This means different components can share a communication channel to receive from/send to the same component. In *multi-way* connection, a component connects with different components using different synchronization actions. As a result, different components use different communication channels to receive from/send to the same component.

For the purpose of analysis and simulation, a CPS system model needs all types of components (software, hardware and physical environment). However, only software components are required for code generation.

4 Analysis and Synthesis

In TIMES-Pro, a system design is represented by a DRT or SDRT task set. Currently the tool offers two main functions: timing analysis of design models and generation of executable code from the models. In this section, first we describe the analysis techniques implemented in TIMES-Pro. Then we show the code generation approach used in the tool to generate Ada code.

4.1 Analysis

DRT/SDRT is supported by a rich theoretical foundation for timing analysis. Efficient feasibility and schedulability analysis algorithms have been developed, even for those problems that are generally intractable (from the computational complexity point of view). The following analysis algorithms are currently implemented as Python scripts in our tool:

1. Feasibility analysis (or EDF schedulability analysis) of DRT tasks in uniprocessor. It is based on an iterative graph exploration procedure based on a novel path abstraction technique [3].
2. Static priority (SP) schedulability analysis of DRT/SDRT tasks using combinatorial abstraction refinement techniques [5,8].
3. Exact response-time calculation of DRT tasks under SP and EDF scheduling. It is also based on the combinatorial abstraction refinement framework to achieve exact results from initial overapproximations [13].
4. DRT workload partitioning on multiprocessors for Partitioned SP and EDF scheduling algorithm. The partitioning algorithms are based on bin-packing algorithms for sporadic task in multiprocessors [14] but extended to support DRT tasks. These algorithms must determine two criteria:

   Task ordering criteria: Different measures can be used to determine the order by which the tasks are selected to be assigned to a core. These ordering criteria can be considered as either "increasing" or "decreasing". Currently, two ordering metrics have been used in TIMES-Pro:

   Utilization: The utilization of each cycle in the DRT graph is defined as the ratio between WCET sum of nodes in that cycle, and sum of their inter-release times. The utilization of a task is defined as the maximum utilization among all the (simple) cycles in the graph.

   Density: The density of a job is obtained by dividing its execution time by its relative deadline. Intuitively, the density of a job shows that how stringent the deadline of a job is with respect to its execution demand. The density of a task is the maximum density among all of its jobs.

   Core selection criteria: Different bin-packing heuristics can be used to select a core to test for the possibility to accommodate the selected task. Currently TIMES-Pro supports two bin-packing heuristics Best-Fit and Worst-Fit [14]. In Best-Fit packing, cores are sorted according to the decreasing utilization order while in Worst-Fit cores are sorted in the increasing utilization order. Uniprocessor schedulability tests of DRT for SP and EDF are used to decide where the selected task can be assigned to that core. In case of SP, the priorities are assumed to be unspecified. Thus, the partitioning algorithm is free to assign suitable priorities to the tasks for better schedulability.

4.2 Code Generation

The goal of code generation is to transform a design model to executable code while preserving the execution behavior of the model. We use the Ada programming language [15] for code generation, as it provides a run-time system suitable for executing real-time tasks. The following important behaviors of the DRT/SDRT task model need to be handled carefully for code generation:

Synchronization in Job Release: In SDRT, release of jobs from two different tasks can be synchronized based on an action. In Ada, 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 in SDRT. The send/call action $s!$ directly maps to 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. If an SDRT job release has both timing and synchronization constraints, then the timing one must be satisfied first. 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.

Branching in Job Release: A DRT/SDRT task can release jobs sporadically, i.e., after the release of a job, the next job can only arrive after waiting for the minimal inter-release separation time. This sporadic behavior may be combined with the branching of jobs in the sense that different types of jobs can be released if their respective inter-release times after the predecessor job are satisfied. To generate code for this behavior, we have two options.

In *branching based on condition*, the next job to be released is determined according to the satisfaction of some conditions. It is assumed that the conditions are checked in an if-then-else structure which, at run-time, deterministically determines which path the program should follow. For example, in Fig. 1(b) the job v_2 has two successors v_1 and v_3. We illustrate corresponding Ada branching code in Listing 1.1. However, this interpretation of branching can not handle job release constrained by a synchronization action where blocking is needed. Therefore this type of branching is only preferable in DRT tasks.

```
case Current_job is
     when v2 =>
       v2_code;
       if Branch_condition then
            Current_job := v1;
            Next := Next + v2_v1_del;
            delay until Next;
       else
            Current_job := v3;
            Next := Next + v2_v3_del;
            delay until Next;
       end if;
      .........
```

Listing 1.1. Branching Ada code for job v_2 of Fig. 1(b).

In *branching based on synchronization*, the next job to be released is decided non-deterministically based on satisfaction of both timing and synchronization constraints. Here we observe three cases: (a) releases of all branch jobs that have both timing and synchronization constraints, (b) some of the branch jobs have release constrained by synchronization constraints but not the rest and (c) releases of branch jobs that are only constrained by timing constraints. Case (c) can be implemented using branching based on condition as described earlier. An example of situation (b) is depicted for job v_2 in Fig. 1(a). Here v_2 has two successors v_1 and v_3. The release of v_3 has to be synchronized with action s_2 while v_1 can be released upon expiration of minimum inter-release separation.

To implement this behavior, we use *selective accept* feature of Ada. As 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. This behavior is implemented in the following steps: first, we sort all outgoing transitions or edges from a job in increasing order of their inter-release times. After observing the smallest possible delay, we insert a selective accept (which means now we can accept an synchronization action and release the branch with the smallest inter-release time) with a delay alternative until the time point when it is also possible to release the next branch. 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 an urgent transition in timed automata. We illustrate Ada branching code for job v_2 in Fig. 1(a) using selective accept in the code segment in Listing 1.2.

```ada
case Current_job is
     when v2 =>
          v2_code;
          Next := Next + v2_v3_del;
          delay until Next;
          Next := Next + v2_v1_del;
          select
               accept s2;
               Current_job := v3;
               Next := Clock;
               goto end_of_case;
          or
             delay until Next;
          end select;
          ..............
```

Listing 1.2. Branching Ada code for job v_2 of Fig. 1(a).

5 Tool Overview

In this section, we present the main features of TIMES-Pro, the tool architecture and the main components in the implementation. Architecture of our tool is shown in Fig. 2.

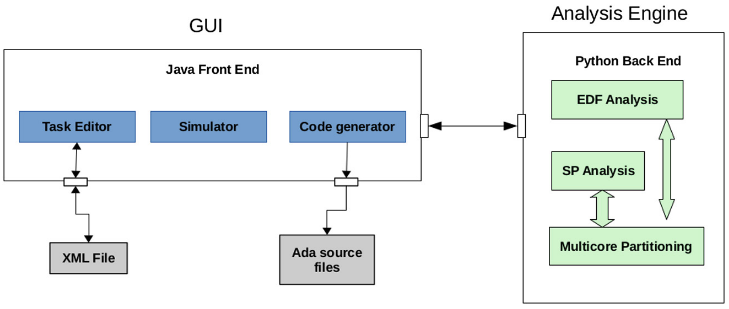
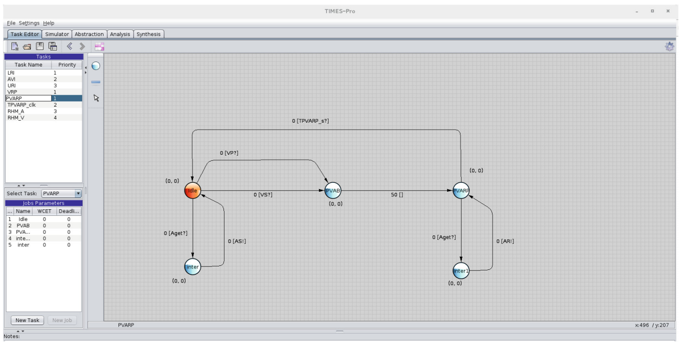


Fig. 2. Tool architecture of TIMES-Pro.

5.1 Features

– **Editor** (see Fig. 3) to graphically model a system and its associated timing, execution resource and synchronization requirements. A system description consists of either a DRT or SDRT task set. The list of all tasks with their assigned priorities is shown in the left side of the main graphical editor. Timing properties of the jobs of a selected task is presented in a table below the task set. All the properties (including the names) of both the task set and jobs properties table are editable.
In the main graphical editor, a task is described by its directed cyclic graph structure. User can define a jobtype by assigning its WCET, relative deadline and associated execution code segment. Different job types are connected by edges where the user can specify the minimum inter-release time between the two jobs. As an incoming edge denotes release constraints of the job, a synchronization action relevant to this job is also specified as the edge property. In the first option, the system designer explicitly states the branching condition variable together with the job code. Branching conditions are also allowed inside a job.



Tasks

Task Name	Priority
LRI	1
AVI	2
URI	3
VRP	1
PVARP	1
TPVARP_clk	2
RHM_A	3
RHM_V	4

Jobs Parameters

...	Name	WCET	Deadli...
1	Idle	0	0
2	PVAB	0	0
3	PVA...	0	0
4	inte...	0	0
5	inter	0	0

Fig. 3. System modeling using SDRT tasks in TIMES-Pro editor.

– **Simulator** (see Fig. 4) to dynamically visualise the execution behavior and the resource utilization of a system model. The simulator generates possible execution traces with zero or random initial phase. This trace is displayed either stepwise or continuously up to the first deadline miss. It is possible to configure the speed of visual simulation within a scale of 1 to 10. System utilization is dynamically displayed below the main simulation. Currently the simulator supports fixed priority and EDF scheduling simulation on a uniprocessor.

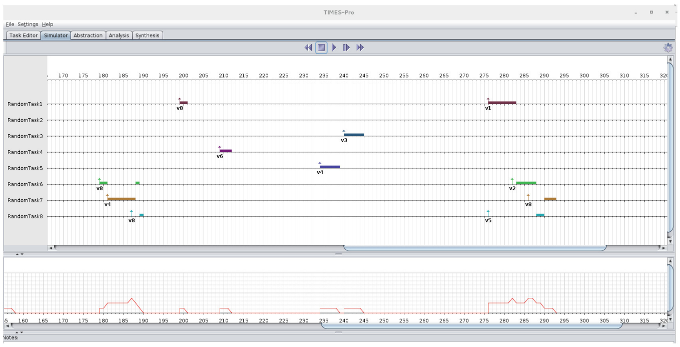


Fig. 4. Visualization of job execution simulation in TIMES-Pro simulator.

– **Analyzer** to check that the tasks associated to a system model satisfy their timing requirements. The analysis suite includes schedulability analysis of tasks under Fixed Priority and EDF scheduling, computation of worst-case response times of tasks and partitioning of workload into multiprocessors. To help testing the algorithms, analyzer has a configurable random task generator which can generate task sets of different size and utilization. Additionally, analyzer provides visualization data of different abstractions used for analysis like request functions.
– **Code Generator** to generate executable Ada code from task sets. The code generator realises a subset of the behavior specified in the DRT/SDRT task model and assumes Ada runtime system will ensure proper execution of the generated code.

5.2 Implementation

Current implementation of the tool is logically divided into three parts:

– **Graphical User Interface** consists of the editor, simulator, visualization of analysis and code generator. It also includes an abstraction visualization tab to visualize workload abstractions to be used in analysis (see Fig. 5). It is possible to check syntax of the model before analysis and the whole system model can be load from or save to an XML file. Currently, the complete GUI has been implemented using JAVA.

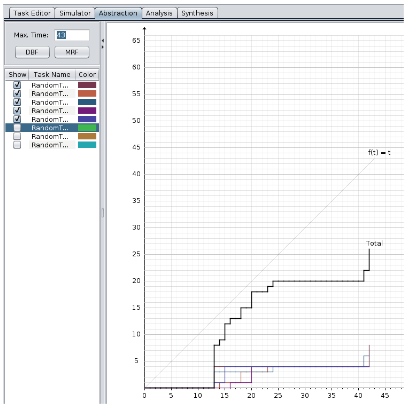
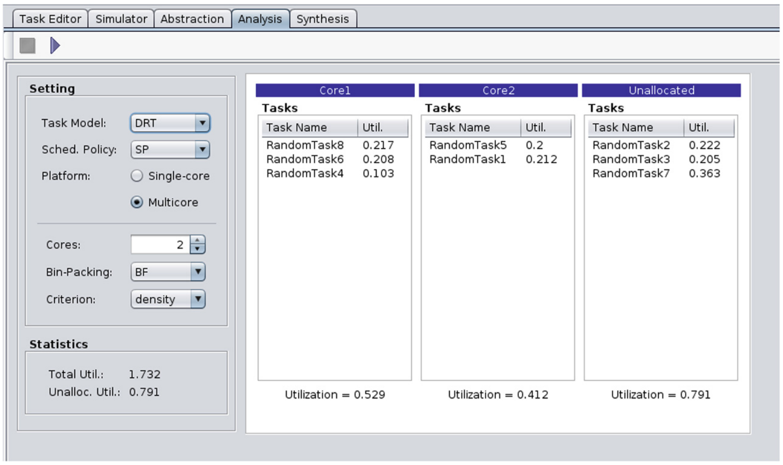


Fig. 5. Workload abstraction visualization in TIMES-Pro abstraction tab.

- **Analysis Engine** implements analysis algorithms using Python scripts. Schedulability analysis and WCRT calculation algorithms for DRT tasks are included in a Python library called *libdrt*. All these algorithms are available for both preemptive Fixed Priority and EDF scheduling algorithms. Analysis engine has a set of workload partitioning algorithms for multiprocessors. Currently implemented algorithms include Best-Fit and Worst-Fit bin-packing algorithms [14] using density or utilization criterion for both partitioned Fixed priority and EDF scheduling. Figure 6 shows different options available in current implementation of the analyzer integrated with GUI. A random task set generator is implemented for creating task sets with different utilizations, size and timing constraints (see Fig. 7).



Setting

Task Model: DRT
Sched. Policy: SP
Platform: Single-core [✓ Multicore]
Cores: 2
Bin-Packing: BF
Criterion: density

Statistics

Total Util.: 1.732
Unalloc. Util.: 0.791

Core1 – Tasks

Task Name	Util.
RandomTask8	0.217
RandomTask6	0.208
RandomTask4	0.103

Utilization = 0.529

Core2 – Tasks

Task Name	Util.
RandomTask5	0.2
RandomTask1	0.212

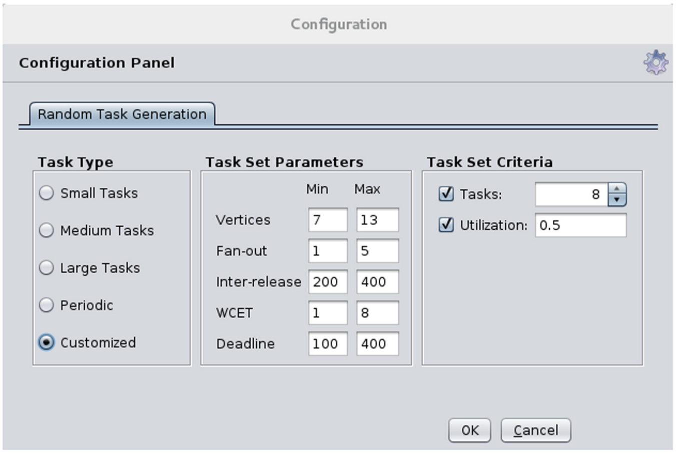
Utilization = 0.412

Unallocated – Tasks

Task Name	Util.
RandomTask2	0.222
RandomTask3	0.205
RandomTask7	0.363

Utilization = 0.791

Fig. 6. A sample analysis scenario in TIMES-Pro analyzer using Best-Fit bin-packing in dual core multiprocessor.


Configuration

Configuration Panel

Random Task Generation

Task Type
- Small Tasks
- Medium Tasks
- Large Tasks
- Periodic
- [✓ Customized]

Task Set Parameters

	Min	Max
Vertices	7	13
Fan-out	1	5
Inter-release	200	400
WCET	1	8
Deadline	100	400

Task Set Criteria
- [✓] Tasks: 8
- [✓] Utilization: 0.5

Fig. 7. Configurations for random task generation in TIMES-Pro.

– **Code Generation** is currently implemented with a separate editor. The code generator translates the graphical model of a task set loaded in the editor into a single Ada implementation file (with .adb extension). It allows editing and syntax checking of auto-generated Ada code. The generated code can be compiled to run on top of generic Ada runtime system.

5.3 Ongoing and Future Extensions

Currently we are working on following feature extensions for TIMES-Pro:

– We are developing novel timing analysis techniques to precisely model the controller software driven by physical system behavior. As a first step, we study an engine control application and present an exact timing analysis by partitioning the state space of the engine behaviors [17]. In future, we intend to generalize this result for any control software driven by physical process and integrate the method to TIMES-Pro.
– We are extending the Code Generator for C code generation that can run using the FreeRTOS [18] real-time operating system. In future, we will generate executable code for multicore platforms based on partitioned multiprocessor scheduling [14].

6 Case Study

We use the heart and dual chamber DDD pacemaker model used in [16] as a case study to illustrate CPS system modeling with our tool.

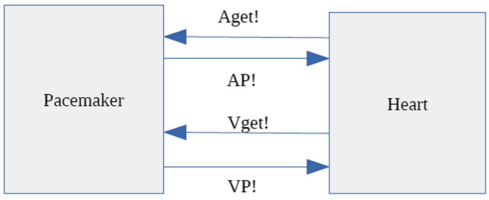


Fig. 8. System-level view of the heart and pacemaker.

6.1 System Modeling

We deal with a closed-loop system with two main components, a pacemaker and a human heart. A pacemaker monitors the Atrial and Ventricular events in the heart and generates required pacing actions based on the state of the heart. The system model is shown in Fig. 8.

The pacemaker receives Aget and Vget events from 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 tasks of the pacemaker. There is another internal signal called AR (Atrial refractory) which is used for the monitoring purpose. The pacemaker generates AP (Atrial pacing) and VP (Ventricular pacing) action signals to the heart model.

6.2 Component Modeling

The pacemaker has five main tasks capturing different timing requirements based on inputs from the heart. Here we describe each of these tasks using SDRT models:

- PVARP: Post Ventricular Atrial Refractory Period (PVARP) task receives Atrial events (Agets) and detects them as AS for synchronizing the other tasks. 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. However during the period of `t_PVARP` the incoming Agets are recorded as AR signals.
- VRP (Ventricular Refractory Period): This task receives Vget events from the heart and recognize them as VS. After each Ventricular event (VS or VP) the task should wait for a period of `t_TVRP` to generate next Ventricular event.
- LRI (Lower Rate Interval): This task keeps the heart rate above desired minimum value. If no AS is received after `t_TLRI` − `t_TAVI` time period following a Ventricular event then AP is delivered.
- AVI (Atrio-Ventricular Interval): This task maintains the delay between the Atrial and the Ventricular activations. If no VS has been sensed within `t_TAVI` after an Atrial event (AS, AP), the task will generate VP. The task should maintain an interval of `t_TURI` between two Ventricular events (VP, VS).

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

The simple version of the random heart model has two tasks, one for generating the Atrial events and another for generating the Ventricular events. Both of these components can randomly generate an intrinsic heart event within a range of valid intervals.

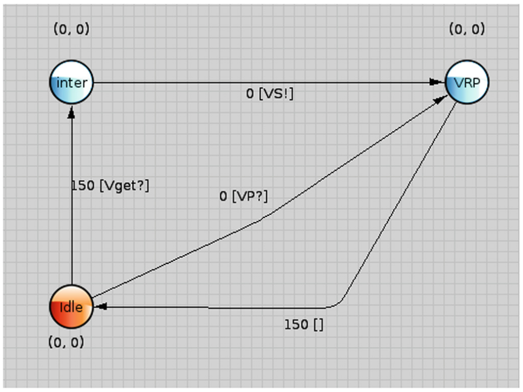


Fig. 9. SDRT model of the VRP component in TIMES-Pro.

```
when Init =>
    Init_code;
    Next := Next + Init_to_temp_delay;
    delay until Next;
    Next := Next + Init_to_VRP_delay;
  select
    accept Vget;
    Current_State := temp;
    Next := Clock;
    goto end_of_case;
  or
    delay until Next;
  end select;
  select
    accept Vget;
    Current_State := temp;
    Next := Clock;
    goto end_of_case;
  or
    accept VP;
    Current_State := VRP;
    Next := Clock;
    goto end_of_case;
  end select;
      .........
```

Listing 1.3. Partial view of the code generated for the component VRP by TIMES-Pro.

Currently we have modeled these pacemaker and heart components in our tool using SDRT tasks. We generated Ada executable code from the model. For example, Fig. 9 shows the VRP component of pacemaker in TIMES-Pro and Listing 1.3 partially shows the generated code. In future we would like to use more complex heart models and visualize the simulation.

7 Conclusions and Future Work

This paper presents an integrated system design tool TIMES-Pro for the design and implementation of CPS. Different design decisions are explained and motivated; the tool architecture and the current state of implementation are presented. As future work, we will further develop the modeling language to support continuous components of CPS, and abstraction techniques for the analysis of combined behaviors by both types of components and generation of executable code to simulate the behaviors in real-time.

References

1. Simulink. http://www.mathworks.com/products/simulink/
2. Modelica. http://modelica.org
3. Stigge, M., Ekberg, P., Guan, N., Yi, W.: The digraph real-time task model. In: Proceedings of RTAS, pp. 71–80. IEEE Press, New York (2011)
4. Stigge, M., Yi, W.: Hardness results for static priority real-time scheduling. In: Proceedings of ECRTS, pp. 189–198 (2012)
5. Stigge, M., Yi, W.: Combinatorial abstraction refinement for feasibility analysis. In: Proceedings of RTSS, pp. 340–349. IEEE Press, New York (2013)
6. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES — a tool for modelling and implementation of embedded systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 460–464. Springer, Heidelberg (2002). doi:10.1007/3-540-46002-0_32
7. Stigge, M., Yi, W.: Models of real-time workload: a survey. In: Audsley, N., Baruah, S. (eds.) Real-Time Systems: The Past, the Present, and the Future, pp. 133–160 (2013)
8. Mohaqeqi, M., Abdullah, J., Guan, N., Yi, W.: Schedulability analysis of synchronous digraph real-time task. In: Proceedings of ECRTS 2016, pp. 176–186 (2016)
9. Fersman, E., Krcal, P., Pettersson, P., Yi, W.: Task automata: schedulability, decidability and undecidability. Inf. Comput. **205**(8), 1149–1172 (2007)
10. Stigge, M.: Real-time workload models: expressiveness vs. analysis efficiency. Ph.D. dissertation, Uppsala University (2014)
11. Guan, N., Tang, Y., Abdullah, J., Stigge, M., Yi, W.: Scalable timing analysis with refinement. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 3–18. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46681-0_1
12. Meyer, B.: Applying "design by contract". Computer **25**(10), 40–51 (1992). http://dx.doi.org/10.1109/2.161279
13. Stigge, M., Guan, N., Yi, W.: Refinement-based exact response-time analysis. In: Proceedings of ECRTS, pp. 143–152 (2014)

14. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011)
15. Ada programming language. http://www.adacore.com/
16. Jiang, Z., Pajic, M., Moarref, S., Alur, R., Mangharam, R.: Modeling and verification of a dual chamber implantable pacemaker. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 188–203. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28756-5_14
17. Mohaqeqi, M., Abdullah, J., Ekberg, P., Yi, W.: Refinement of workload models for engine controllers by state space partitioning. In: Proceedings of ECRTS (2017, to appear)
18. FreeRTOS Real-time Operating System. http://www.freertos.org