Received: 1 March 2019 | Revised: 23 January 2020 | Accepted: 9 March 2020

DOI: 10.1002/spe.2831

EXTENDED CONFERENCE PAPER

WILEY

Designing real-time systems using imprecise discrete-event system specifications

Gabriel Wainer[1] | **Mohammad Moallemi**[2]

[1]Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada

[2]ERAU NEAR Laboratory, Embry-Riddle Aeronautical University, Daytona Beach, Florida

Correspondence
Gabriel Wainer, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario K1S 5B6, Canada.
Email: gwainer@sce.carleton.ca

Funding information
Natural Sciences and Engineering Research Council of Canada

Summary

Real-time (RT) systems include hardware and software components interacting in a tight fashion. Although formal methods for RT systems development have advanced, they are sometimes difficult to apply in practical applications, and scalability is compromised as the complexity of the system scales up. Instead, using modeling and simulation (M&S) methods and tools has showed to be useful for verification of practical aspects of RT systems (and having the advantage to be able to including models of the physical environment they interact with). Although several efforts exist in M&S of RT systems, none of them has considered problems of transient overloading in the RT systems specifications. Here, we introduce a new theoretical framework called I-DEVS (imprecise discrete event systems specification) with the goal of guaranteeing responses to inputs within specified time constraints under such transient overloading conditions. The solution presented here has the advantages of a formal specification and the practicality of an M&S-based approach. We also discuss how to define hierarchical models running in RT, and we present a set of tools that can be applied to develop RT-embedded applications, and RT simulations.

1 | INTRODUCTION

Real-time (RT) systems are usually built as software components embedded in specialized hardware interacting in a tight fashion with their environment and satisfying "hard" timing constraints (with deadlines at millisecond scales). Decisions taken by such RT systems can lead to catastrophic consequences (either financially or in human lives); thus, their correctness and timeliness are critical. In recent years, RT systems have faced growth both in number and their complexity. The ever-increasing demand for new features and quality (in terms of safety, dependability, fault-tolerance, power consumption, etc.) combined with decreasing budget and time-to-market pose new challenges to RT systems designers. Various systems ranging from autopilots, power plant control, transportation, and telecommunication, to customer electronics, medical equipment, and intelligent and automated systems are examples of such hard RT systems.

A RT system is defined by Liu[28] as "a system that is required to complete its work and deliver its services on a timely basis." Many of these systems are deployed in embedded microprocessors working in hardware computing platforms with special configurations and interfaces. Reference 34 describes an embedded system as "a system designed to perform a dedicated function, typically with tight RT constraints, limited dimensions, and low cost and low-power requirements." The architecture of these systems usually integrates different types of hardware components such as processors, analog and digital components, as well as mechanical (eg, sensors and actuators) and visual components, which demands increasingly challenging multidisciplinary design and development efforts. In particular, when multiple tasks run in a single

processor, we might end up not being able to execute all the tasks before their corresponding deadlines, producing what is called an *overrun condition* in which we will miss deadlines until the condition is resolved (in some cases the overruns are caused by triggering of emergency tasks that use extra CPU cycles, and when the emergency ends, the overrun condition is resolved).

Different formal methods have been proposed to design and develop these systems, since they provide a provable hence reliable framework that facilitates the construction of such critical applications. These methods are special cases of mathematical-based techniques for design, development, and verification of software and hardware systems.[33] They allow for appropriate mathematical specification and analysis of the designs, which can contribute to the reliability of the final system, yet they add to the complexity of the design and increase the cost of the development. Therefore, they are mostly appropriate for systems with critical applications where safety and robustness are a foremost aspect. Unfortunately, these methods do not scale up well, as most formal proving mechanisms cannot provide formal proofs of correctness when the complexity of the system grows.[1,14]

Instead, modeling and simulation (M&S) provides a practical solution in solving the above-mentioned difficulties in the design of RT and embedded systems, caused by formal methods. Computer-based M&S is a useful tool for efficient analysis, design, verification, and optimization of general dynamic systems. The use of M&S in software engineering reduces costs and risks and allows for exploring different aspects of the system.

Formal M&S is a branch of M&S, in which the simulation models are defined using a formal approach. This technique has shown promising results in making multidisciplinary system development tasks manageable.[49] It provides a hierarchical design scheme in which higher abstract levels are branched into levels that contain more details. The system specifications are expressed using mathematical notations in which the details of the behavior of the system are accurately modeled. Other advantages of formal M&S are applying formal model checking techniques at design time,[37,41] incremental refinement of the initial simulation models, simulation-based validation, reuse of the existing models, risk free testing of critical RT applications.

We present new techniques to overcome overrun conditions in hard-RT systems designed using a M&S-driven approach[31,32] based on a formal specification: the DEVS (Discrete-Event System Specification) formalism.[49] DEVS is an increasingly accepted framework that provides an abstract and intuitive way of modeling, independent of underlying simulators, hardware, and middleware. DEVS supports hierarchical and modular construction of models, which fits our needs (models at different levels of abstraction can be defined independently, and later integrated into a hierarchy). DEVS decouples model, experiments, and execution engines (allowing for portability and interoperability).

One critical aspect of a hard RT system is the production of outputs before the specified deadline. For instance, the deadline to trigger the request to deploy an airbag after a crash is 10 ms, in order to meet the mandatory 55 ms deadline for the airbag to be deployed. If that deadline is missed, the passengers can die. As we can see, late outputs in such systems not only degrade the system performance but also produces catastrophic results (loss of lives and expensive assets). However, in circumstances with system overloads, it might be impossible to meet the deadlines. Since RT systems are not deterministic, tasks may enter the system at any time hence, there is no prior knowledge of their occurrence times.

The imprecise computation (IC) technique[25] helps to overcome these high computation peaks by discarding unnecessary computations in overload conditions. The main idea is to separate the computations into mandatory and optional chunks (the mandatory computation affects the correctness of the result and the optional one affects its quality). This research aims at introducing a flexible RT task execution paradigm for DEVS by incorporating IC technique into the DEVS task model (note: although we use the DEVS acronym throughout the paper, the research is based on P-DEVS with ports). Based on the requirements in a hard RT system, it is safer for a task to produce less accurate results on time, rather than producing the accurate result, late. The motive is to employ IC with the proposed RT DEVS approach[32] in order to have a formal platform for designing hard RT systems. The objective is to address the above challenges in the proposed DEVS-based RT design scheme, without complicating the formalism or adding extra processing burden and maintaining the backward compatibility in order to reuse previous models. The contributions of this paper can be summarized as follows:

- We discuss in detail the use of Imprecise DEVS (I-DEVS), which provides flexibility to the user by separating the behavior of the system to mandatory and optional, to achieve a more reliable RT task scheduling from the processor.[31] The imprecise computing approach allows early detection of system overloading, which, in turn, can improve fault tolerance which is crucial in the field of embedded systems.

- A detailed (model-independent) task model of DEVS formalism is presented. The idea is that, starting from an I-DEVS model, we can construct a standard scheduling task model (which can be used for schedulability analysis), in which we identifies processing tasks from the I-DEVS specification, and we define a method to convert in schedulable CPU cycles to be executed in a RT DEVS-based system. This proposal is then used to develop RT IC-based scheduling algorithms.
- An early reaction algorithm to the overrun conditions in DEVS-based hard RT systems is proposed, in which the system can act early enough to save the critical tasks from lateness.
- The I-DEVS M&S framework is implemented on E-CD++ M&S software,[44] providing a development platform for imprecise modeling and execution using DEVS formalism.

The I-DEVS "flavor" is provides a few advantages over standard DEVS. As it has been formally proven that DEVS is the most generic Discrete-Event System Specification,[49] I-DEVS can be considered as an instance of DEVS. Nevertheless, I-DEVS simplify the task of the modeler, who only needs to identify if a task is mandatory or optional, and this provides various advantages, summarized here:

- Identifying the mandatory and optional tasks at the model level, allows one to define, study, verify and properly implement independent simulation/real-time engines, so that modelers can focus on the imprecise models themselves, instead of needing to implement the imprecise computing methods as parts of their own models, making the modeling task simpler.
- The semantics of I-DEVS is different, as it will be discussed in detail in the following sections: differently from DEVS, the optional tasks do not execute under transient overloads. A formal specification of the operational semantics of I-DEVS (which is outside of the scope of this article), could be used to define and study properties of the models defined, allowing early detection of errors in the model specification.
- Having a mandatory/optional task definition mechanism allows the modeler to conduct model design with that mindset and makes accessible the use of the imprecise computing method, which, at present, is not being used by designers as they are not aware of it. By providing the methods and tools to use imprecise computation improves the modeling approach from the design phase.
- Using this approach, we can rely on the various schedulability tests that exist for guaranteeing timeliness of hard real-time tasks using imprecise computing. Although this research is outside the scope of this paper, we have explored these methods[29] showing how, from a I-DEVS specification, we can formally analyze the schedulability of a set of tasks defined as I-DEVS models.

The rest of the paper is organized as follows: section 2 introduces related work; section 3 defines the task model derived from RT-DEVS. Section 4 introduces the I-DEVS methodology, after which, section 5 introduces and discusses a case study showing implementation results.

2 | BACKGROUND

In this section we introduce related work and background information for this research. It includes two sub-sections, the first introducing RT systems with imprecise computations, and the second introducing DEVS and RT DEVS.

2.1 | RT systems and imprecise computations

The imprecise computations (IC) technique[25] is a useful approach for handling RT scheduling issues under transient overloads. It introduces a methodology for separating the critical (mandatory) part of a task from its uncritical (optional) part, thus making it possible for a RT system to accept more tasks to the system while trying to run as many optional subtasks as possible. This allows the system to be dynamically configured to accept more tasks when the system's processing traffic is high, by producing less accurate results, and on the other hand, when the system burden is low, execute tasks completely to produce accurate results.

Within the IC definition, a monotone task is defined as a task in which "the quality of its intermediate result does not decrease as it executes longer."[24] Monotone tasks can be found in almost all RT applications and their flexibility in terms

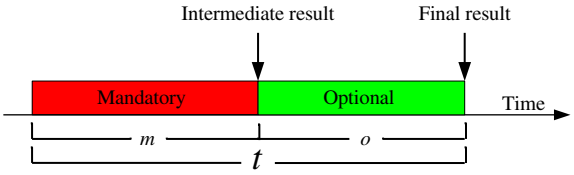


FIGURE 1 A monotone task divided to mandatory and optional parts[25] [Color figure can be viewed at wileyonlinelibrary.com]

of duration of computation helps designers to implement the IC technique. A solution to handle high processing peaks in an RT system is to divide the monotone tasks into two versions of a computation: the primary, which executes longer and produces more accurate result and the secondary version which executes shorter but produces less accurate result. Whenever the deadline is short, the secondary version can be processed to meet the deadline while having an acceptable result.

Contrary to monotone tasks, tasks with 0/1 constraints must be executed to completion or not executed at all. Scheduling 0/1 constraint tasks is more difficult.[24]

The following definitions are used in scheduling algorithms for imprecise computations: Considering a task set $T = (T_1, T_2, \dots, T_n)$ of preemptable tasks, the following parameters are defined regarding each task T_i: r_i is the time at which task T_i is released; di is the deadline at which task T_i must be completed; t_i is the processing time required for task T_i; and w_i is the weight of the task T_i which is the relative importance.

Every task T_i is composed of two subtasks: *Mandatory* and *Optional*. The mandatory subtask M_i needs processing time m_i and the optional subtask O_i needs processing time o_i. Then $m_i + o_i = t_i$. Figure 1 illustrates these definitions.

In a schedulable RT system, each task T_i is referred to as "being executed" when at least all its mandatory subtasks included in the associated jobs are executed (jobs are instances of tasks occurring during the execution). The optional subtasks of task T_i are available for execution only if the mandatory subtasks of T_i are executed properly. The scheduler can terminate an optional subtask at any time during its execution. Based on this definition, a perfect hard RT system is a system in which all the tasks are composed of mandatory subtasks and a perfect soft RT is a system in which all the subtasks are optional.[25]

IC has been well investigated and many RT scheduling algorithms have been introduced (some of these algorithms are presented in section 3). Imprecise scheduling algorithms benefit from the separation of mandatory and optional subtasks, in which they apply appropriate scheduling procedures for each processing type. The priority of the jobs is also considered by some of the algorithms and there are algorithms for periodic jobs, too. Moreover, the concept of error is defined based on the portion of the optional subtask that has not been executed in the context of a full task.

A common problem in hard RT systems is the occurrence of overrun situations when the system does not have enough processing resources to fulfill all the requesting processes. This issue poses critical risks to the machinery under control, and it may cause catastrophic results. IC technique offers an effective way of resource utilization in such circumstances. We show how to use this technique for the proposed RT DEVS framework[32] by introducing the Imprecise-DEVS (I-DEVS) methodology. This approach combines the dynamic advantages of the IC technique with the rigor of a formal modeling methodology.

Imprecise computation has been used for minimizing error in RT task scheduling.[5] The error is calculated as a function of the amount of discarded optional processing as a result of overrun situation happening in the system. Many off-line task scheduling algorithms have been proposed in the past, based on IC technique (refer to References 3,27,and 38). There is no optimal algorithm that minimizes total error in on-line RT scheduling systems, when a feasible schedule exists, because of the lack of a-priori knowledge of the occurrence time of the jobs.[39]

The mandatory first algorithm assigns processing times to mandatory tasks first, based on statistics to reduce the total error (refer to References 4 and 12).

The NORA (No-Off-line tasks and on-line tasks Ready upon Arrival) algorithm[40] is based on EDF (Earliest Deadline First) algorithm and is mainly designed for online task systems, in which each task is ready upon arrival. Each task is assigned a reserved interval based on reverse scheduling algorithm. Each task's mandatory subtask is assigned a reserved execution time starting from its deadline equal to the amount of processing time required for its mandatory subtask, based on the EDF algorithm. As long as the mandatory task set is feasible a reserved interval set can be found.

The DOT_Sched algorithm[10] is an extension to the NORA algorithm for online tasks that are ready upon arrival. It uses three reservation lists: R(M) for mandatory tasks, R(O) for optional tasks, and R(M + O) for both of them. Like NORA algorithm each task is assigned a reserved interval in R(O) or R(M) and R(M + O) lists, starting from its deadline way back equal to its processing time.

RT-Frontier[22] is a RT operating system that presents an imprecise computation framework for constructing RT applications. It decomposes computations to mandatory, optional, and wind up parts. The wind-up part works as a termination function after the termination of an optional part, reducing the termination cost and increasing portability. A novel scheduling algorithm called Slack Stealer for Optional Parts (SS-OP) is used in RT-Frontier which is based on the above mentioned three segment imprecise computation model and imposes small overhead to the system, while applying dynamic load balancing scheme.

Except the work presented in Reference 22, which only applies IC in a specific operating system, no research aims at integrating this technique with a formal methodology to be used in RT and embedded system design and construction. The proposed I-DEVS approach, allows the model designer to deploy this technique at the design time, specifying the optional and mandatory behaviors of the target system.

IC has been applied to different fields, including RT and embedded systems,[22,26,46] multimedia processing,[9,13,20] planning and artificial intelligence,[15,35] and databases.[18,2] Despite all of these efforts, IC is not yet widely used in industrial embedded applications. In recent years it has also found its way into other related fields, including scheduling in fog computing,[7] energy-aware systems,[50] Operating Systems design,[11] between others. The reason might be related to "strict theoretical assumptions and the lack of cost-effective support method that can be easily implemented in embedded systems."[22] Here, we want to address these issues in the context of RT DEVS-based systems.

2.2 | The DEVS formalism

DEVS is a formal M&S methodology, based on generic dynamic systems, including well-defined coupling of components, hierarchical, modular construction, support for discrete event approximation of continuous systems and support for repository reuse.[8] A real system modeled with DEVS is described as a composite of sub-models, each of them being behavioral (atomic, see Figure 2) or structural (coupled). The complexity of these systems has encouraged the use of model-based approaches for designing purposes. M&S techniques provide solutions for efficient analysis, design, verification, and optimization of these systems reducing cost and risk. Related works based on DEVS for RT are presented in Reference 16.

A DEVS model is described as a set of basic atomic and coupled models. Atomic models are still the most basic constructions, which can be combined with other models into coupled models. The DEVS atomic model has the following structure:

$$AM =< X_M, Y_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >,$$

where $X_M = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of input ports and values. $Y_M = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values. S is the set of sequential states. $\delta_{ext}\ Q \times {X_M}^b \to S$ is the external state transition function. $\delta_{int}\ S \to S$ is the internal state transition function. $\delta_{con}\ Q \times {X_M}^b \to S$ is the confluent transition function. $\lambda\ S \to {Y_M}^b$ is the output function. $ta\ S \to R^+_{0,\infty}$ is the time advance function; with, $Q: = \{(s, e) \mid s \in S, 0 \le e \le ta(s)\}$ the set of total states.

The semantics of the DEVS definition are as follows. At any given time, a basic model is in a state *s*. and in the absence of external events, it will remain in that state for a period of time as defined by **ta**(s). When an internal transition takes place, the system outputs the value λ(s), and changes to state δ_{int}(s). If one or more external events $E = \{x_1 \ldots x_n / x \in X_M\}$ occurs before ta(s) expires, that is, when the system is in the state (s, e) with $e \le$ **ta**(s), the new state will be given by

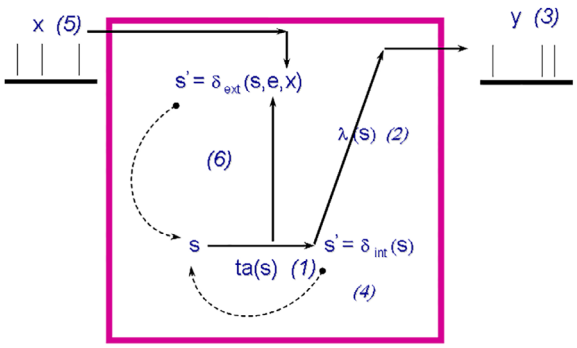


FIGURE 2 DEVS atomic component state transition sequence [Color figure can be viewed at wileyonlinelibrary.com]

δ_{ext}(s, e, E). Suppose that an external and an internal transition collide, that is, an external event E arrives when e = **ta**(s), the new system's state could either be given by $\delta_{ext}(\delta_{int}(s), e, E)$ or $\delta_{int}(\delta_{ext}(s, e, E))$. The modeler can define the most appropriate behavior with the δ_{con} function. As a result, the new system's state will be the one defined by $\delta_{con}(s, E)$.

A coupled model connects the basic models together in order to form a new model. This model can itself be employed as a component in a larger coupled model, thereby allowing the hierarchical construction of complex models. The coupled model is defined as:

$$CM =< X, Y, D, EIC, EOC, IC >$$

X is the set of input ports and values, Y is the set of output ports and values, D is the set of the component names, EIC (external input couplings) connects the input events of the coupled model itself to one or more of the input events of its components, EOC (external output couplings) connects the output events of the components to the output events of the coupled model itself, IC (internal coupling) connects the output events of the components to the input events of other components.

The DEVS formalism was originally defined to allow the definition of discrete-event models and their simulation. In order to extend this formalism to RT simulation and application development, we proposed the RT DEVS as a RT extension to DEVS[32] to provide RT simulation capability as well as a formal methodology for simulation-driven development of RT and embedded applications. Unlike RT-DEVS,[19] this formalism applies minor modifications to DEVS, allowing for easy reuse of the previous models.

The most critical characteristic of a RT system is the availability of output within the specified deadline. RT DEVS assigns a deadline to each output in the atomic component, and it verifies the deadline when the associated output is produced. Hence, the concept of deadline is embedded in the formalism and it is implemented in the abstract simulation mechanism.

The atomic component of RT DEVS is formally defined by

$$AMRT =< X, S, Y, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta, d >, \quad (1)$$

where X, S, Y, δ_{ext}, δ_{int}, δ_{con} and λ are the same as DEVS.

ta: $S \rightarrow R^{+}_{0,\infty}$, time advance function (which is tied to physical clock of the system).

d: $S \rightarrow R^{+}_{0,\infty}$ is the relative deadline of each state for output production. The deadline starts at the end of the associated state when the output function is invoked to produce an output (ie, considered the release time of the output task). The deadline is allocated to each output generated by the output function. Management of the deadline is done by the time-advance function.

The engines that execute atomic and coupled components provide an abstract simulation mechanism, referred to as *simulators* and *coordinators*, respectively. Thus, the control hierarchy is composed of coordinators as middle nodes and simulators as the leaves. The top-most coordinator, referred to as *Root Coordinator* (RC) initiates each simulation phase by sending the following messages to the simulators: (q, t) representing an *input message* that carries an input value from external environment and the time stamp of the message. The *collect message* (@, t) gives an instruction to generate an output. The *internal message* (*, t) is for an imminent simulator. The *output message* (y, t) is produced in response to a collect message.

A coordinator is responsible for converting output messages to input messages in case of an internal coupling. It is also responsible for sending the smallest time of internal event (also referred to as *next change* (t_N)) among its components. The *last change* time (t_L) is the relative time from the last activity in a component to the current time. The following snippet represents collect message handling algorithm in a P-DEVS simulator.

```
1 when receive (@, t):
2 if (t = t_N) then
3       y = λ (s)
4       send (y, t) to the parent coordinator
5       send (done, t) to the parent coordinator
6 else
7       error
```

The following snippet represents input message handling in P-DEVS.

```
1  when receive (q, t):
2        Add event q to the bag
3        send (done, t) to the parent coordinator
```

The receipt of input message does not trigger the external transition function. Instead, the input is inserted in the bag, allowing for processing simultaneous inputs stored in the bag, when an internal message is received. The following snippet represents the internal message handling.

```
1  when receive (*, t):
2  if (t_L ≤ t < t_N) and bag is not empty
3        e = t − t_L
4        s = δ_ext(s, e, bag)
5        empty bag
6        t_L = t
7        t_N = t_L + ta(s)
8  else if (t = t_N) and bag is empty
9        s = δ_int(s)
10       t_L = t
11       t_N = t_L + ta(s)
12 else if (t = t_N) and bag is not empty
13       s = δcon(s, bag)
14       empty bag
15       t_L = t
16       t_N = t_L + ta(s)
17 else if (t > t_N or t < t_L)
18       error
19 send (done, t_N) to parent coordinator
```

The internal message will produce three different executions: line 2 is the case when the internal message is received before the end of current state's lifetime and the input bag is not empty. This means there are inputs to be serviced; hence the external transition is invoked. Line 8 is when the internal message is received at the end of the lifetime of the current state, while the input bag is empty, indicating an internal transition. Line 12 shows when internal message is received at the end of the state and there are inputs to be served. Therefore, confluent function must be called to handle the collision of external and internal transitions.

3 | RT DEVS TASK MODEL

The main computations in a RT DEVS runtime engine occur in state transition functions (ie, the functions defined by the modeler in the atomic models) as well as in scheduling and message transfers (described in the coordinator/simulator subsystem in section 2). Assuming that the message transfer has overhead associated to the context switching between the standard DEVS tasks in the system, the set of tasks in a RT DEVS system (in theory) is composed of transitions and output functions. This information is used to map the DEVS functions (δ_{ext}, δ_{int}, δ_{conf}, and λ) run by the RT simulator into an RT kernel and scheduler, providing a platform where IC can be applied. The set of tasks in a RT DEVS system (in theory) is composed of transitions and output functions (δ_{ext}, δ_{int}, and λ) run by the RT simulator. Figure 3 shows the processing tasks in such a RT DEVS atomic component during a state transition (note that in this example, used to show the process, we assume that each transition function takes 1 time unit, and the context switch between them is negligible; the figure shows how to map the DEVS transition functions into a scheduling timeline).

As we can see, the external transition (δ_{ext}) is mapped into a task named "T_X" that initiates the state S. The task's release-time is equal to the arrival of the input at the system. No deadline is assumed for the T_X task (as it is an input and

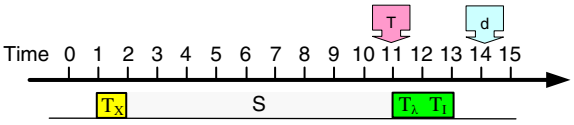


FIGURE 3 Processing carried for a state transition [Color figure can be viewed at wileyonlinelibrary.com]

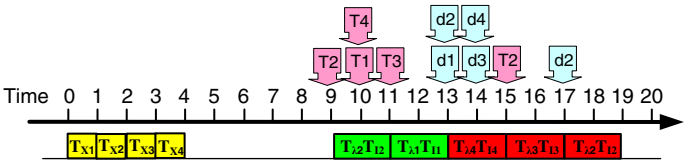


FIGURE 4 Overload scenario [Color figure can be viewed at wileyonlinelibrary.com]

it can arrive at any time). The output task "T_λ" and internal transition (δ_{int}) task "T_I" are considered to execute together forming $T_\lambda T_I$ task (as outputs in DEVS are always followed by an internal transition). The release time of task $T_\lambda T_I$ is equal to the end of the state S and specified by *ta(s)* (indicated as **T** in Figure 3). Its deadline is specified by the *d(s)* function in Equation (1) (and indicated as **d** in Figure 3).

This RT DEVS definition provides an M&S-based design scheme method for engineering RT systems, in which system components can be simulated (and formally analyzed) before the actual development and deployment.

Let us assume a scenario where the system receives a large number of inputs. In that case, the system needs to execute many transitions and produce the corresponding outputs, making the output tasks delayed (and even exceeding their deadlines). Figure 4 shows such an overload scenario when four inputs are injected, starting external transitions on different atomic components.

As discussed in Figure 3, we can see the release time of the tasks triggered by the time advance function *ta*(*s*) for each task, depicted as Ti in Figure 4. Also, we show the associated deadlines for each of the tasks. There is only one processor that executes the transition functions (showed in yellow/green/red under the timeline). As can be seen, Tλ1, Tλ2, and Tλ4 meet their deadlines; however, Tλ3 and Tλ2 (the second instance produced by an internal transition at time 18) do not. The internal transition task TI2 produces a new state with ta(s) of 4 time units, which exceeds its deadline at time 17. This situation can be avoided if multiple internal transitions tasks happen sequentially. However, it could have been prevented, if the system would have detected the overload conditions early enough to apply IC-based scheduling. In order to avoid this overload scenario, we divide the tasks into mandatory and optional parts. The states of the atomic model are categorized as mandatory and optional. A mandatory state will lead to a mandatory output function (represented as output task) and an optional state will lead to an optional output task. This abstraction in the definition of mandatory and optional tasks in the level of state machine, allows the system designer to define imprecise components without being involved in the details of the lower level tasking system. Assuming input T_X tasks are always mandatory (to avoid missing any inputs) non-critical output represented by T_λ tasks can be optional. The T_λ subtask of an optional $T_\lambda T_I$ task can be terminated under transient overloads. In other words, during overrun situation, the system skips optional outputs to save time and resources for the mandatory ones. For instance, an aircraft or an automobile control system during a critical stormy situation with a bulk of reconnaissance and control tasks can discard infotainment tasks to respond to the critical control commands. A similar scenario can occur in any RT system where a sequence of non-critical outputs can be skipped to alleviate the overload situation by keeping the necessary outputs produced on time.

A RT system can be prone to overload conditions at any time during execution, posing risks and reducing the reliability of the system in hard RT conditions. Nevertheless, the overrun situation can be transient, happening at very random occasions. The nondeterministic nature and lack of a-priory knowledge of the occurrence times of the tasks, adds to this issue severing the risk. Based on these ideas, we propose using an IC-based approach in the context of RT DEVS modeling. Defining a clear theoretical tasking system of RT DEVS provides a reliable starting point to employ various available techniques and algorithms for reliable scheduling and also schedulability analysis of these systems. The proposed I-DEVS approach with minimal modification to DEVS, allows for well-organized integration of IC technique with DEVS.

4 | IMPRECISE DEVS

As mentioned earlier, we want to provide an IC framework for applications where the job arrival times are not known a-priori. The approach tries to balance the computation when the system is busy and on the other hand not reducing

its performance, while keeping the run-time overhead of the implementation as low as possible. The method combines theoretical concepts in IC and DEVS modeling, while having the power to be used in practical projects based on this technique.

I-DEVS uses RT DEVS and adds a mandatory or optional condition for each state, as follows:

$$AM =< X, S, Y, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta, d >$$

where X, Y, δ_{ext}, δ_{int}, δ_{conf}, λ, ta and d are the same as in RT DEVS, S {(s, c) | s$\in Z^+_0$ and c $\in$ {mandatory | optional}}.

The states of the atomic model are categorized as mandatory and optional. A mandatory state will have a mandatory output function (represented as an output task) and an optional state will produce an optional output task. This abstraction in the definition of mandatory and optional tasks in the level of state machine allows the modeler to define imprecise models without being involved in the details of the lower level tasking system. This artifact allows implementing a real-time engine that executes the model, making the execution mechanisms (the "simulator" if running in simulation mode, or the "real-time engine" when running in real-time) independent of the model specification, making easier the implementation of those execution artifacts and related schedulability tests.

The main runtime algorithm performed in the Root Coordinator (RC) (the top coordinator in the DEVS abstract runtime hierarchy introduced earlier[49]) is unchanged. It is started first by waiting for an external or internal event. RC routes external input through an external message (q) to the destination atomic model (which triggers the δ_{ext} function). Otherwise, it waits for the closest internal event (λ1), and then sends collect (@) and internal messages (*) to the target atomic model to trigger λ and δ_{int} functions, respectively. The collect message executes the λ function on the atomic model and the internal message executes δ_{int}. The atomic model responds to the @ message by executing the λ function and returning the output value through an output (y) message. The atomic model also executes δ_{int} in response to a * message and returns its next internal event time by a done message. The time advance is driven by the RT clock.

Whenever there is more than one output task to be serviced in an overrun situation (ie, when tasks drift later than their release times), the mandatory ones have priority over the optional ones. If an optional output task is to be serviced later than its release time plus a grace period, it will be discarded. The grace period depends on various factors and it defines a threshold for tolerating lateness in processing optional tasks. In the case studies presented later in the paper, we use a grace period of zero-time units, the most stringent scenario. In order to detect transient overload conditions, we check to see if the tasks are drifted late from their release-times. When there is a drift from the release time, this might be an indication of an overloaded system which, in the near future, could trigger new tasks to react to these conditions (using an early reaction strategy). Using a grace period, we can decide on a threshold used to decide when the system starts dropping optional tasks. The grace period is handled by the runtime engine, and it can be a function of the processing resources, level of criticality of the optional tasks, or other function towards reacting to such conditions. The grace period can be used to tune the system to obtain desirable tradeoffs between losing accuracy and meeting hard deadlines. Systems with hard RT deadlines can have a shorter grace period in order to save time for mandatory tasks by sacrificing optional ones and gaining higher reliability, while soft RT systems could tolerate delays sporadically in order to achieve higher precision and quality. The grace period could also be modified dynamically by the runtime system, using learning algorithms to adapt to changing conditions. On the other hand, these dynamic conditions can be explored by simulating the I-DEVS model, allowing the modeler to conduct experiments in a risk-free environment with various test scenarios before it is deployed on the target hardware.

The early reaction strategy helps the system to save time for later mandatory tasks that have not been released yet. Whenever a sequence of optional events in a system is delayed, the system starts discarding the optional tasks.

The early reaction strategy helps the system to save time for later mandatory events that have not been released yet. Whenever a sequence of optional events in an atomic model is delayed, the atomic model starts discarding the output functions. Therefore, the modified execution algorithm in an atomic model, when receiving a collect message is shown below.

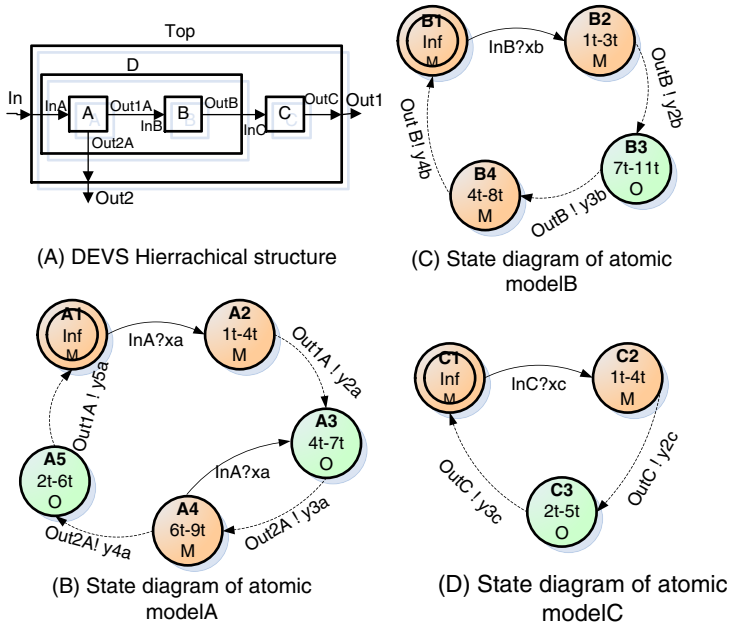


FIGURE 5 Example I-DEVS model [Color figure can be viewed at wileyonlinelibrary.com]

```
1  Receive (@, t)
2  if (s is optional AND t_L + ta(s) + t_g < t_now)
3        raise error //optional tasks dropped
4  else if (t_now > t_L + d(s))
5        raise error //deadline missed
6  else if (t_now ≤ t_L + d(s))
7        y = λ (s)
8        send (y, t) to the parent coordinator
9  send (done, t) to the parent coordinator
```

Line 2 verifies if the optional output is going to be executed later than its release time ($t_L + ta(s)$) plus the grace period (t_g). The idea is to add the grace period discussed earlier to the deadline of the optional output. Line 4 verifies the deadline condition and finally line 6 is the case when the output function is qualified for execution. (t_{now} is the current simulation time and t_L is the time of the last event in the system).

4.1 | Execution scenario

Figure 5A shows a sample RT system design hierarchy (designed in I-DEVS) where two atomic components *A* and *B* are coupled into *D*, which is itself coupled with atomic component *C*. Various input/output ports connect these components together, as shown in the figure. Figure 5B shows the state diagram of component *A* using DEVS-Graph. The state diagram summarizes the behavior of a DEVS atomic component by presenting the states (circles), transitions (arrows), inputs, outputs, and state durations, graphically. Note that continuous arrows indicate external transitions.

External transitions are triggered by external events; in this case, DEVS Graphs use a notation used in the CCS formal language, meaning that there is an input (represented by the question mark "?") coming on a given port (to the left of the "?" sign), with a specific value (specified to the right of the "?" sign). When that input occurs, a transition is triggered. As it can be seen, model A is initially in state *A1* (with life-time = infinity, representing a passive model waiting for an input), until an input *xa* is received on port *InA*. In that case, the external transition causes a state-change to *A2*. The system stays in this state for 1t while its deadline is 4t (written inside the state circle).

Internal transitions, represented by dotted lines, are triggered when the lifetime of a given state is consumed. As this is a DEVS model, the transition is preceded by an output. Using CCS notation, here we use a question mark "!" to represent

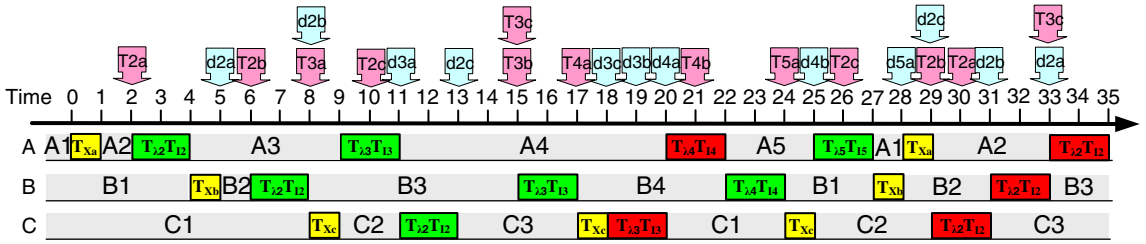


FIGURE 6 Example transient overload scenario [Color figure can be viewed at wileyonlinelibrary.com]

the output, that is sent through a given port (to the left of the "!" sign), with a specific value (specified to the right of the "!" sign). In our example, when the lifetime of the *A2* is consumed, it produces the output *y2*a and transits to state *A3* (internal transition). A similar scenario can be seen in states *A3*, *A4*, and *A5*, with outputs *y3a*, *y4a*, and *y5a* produced, respectively. Figure 5C,D shows the DEVS-Graphs for atomic components *B* and *C*, respectively. To clarify the details of the model, the DEVS-Graph of atomic model C (shown in Figure 5D) can be formalized as a DEVS specifications as follows:

$$\mathrm{C} = < \mathrm{X}, \mathrm{S}, \mathrm{Y}, \delta_{\mathrm{ext}}, \delta_{\mathrm{int}}, \delta_{\mathrm{conf}}, \lambda, \mathrm{ta}, \mathrm{d} >,$$

where X = {(InC, xc)}, S = {(C1, mandatory), (C2, mandatory), (C3, optional)}, and S_0 = C1, Y = {(OutC, y2c), (OutC, y3c)}, δ_{ext} (C1, e, <InC, xc>) = C2, δ_{int} (C2) = C3, δ_{int}(C3) = C1, $\delta_{\mathrm{conf}} = \delta_{\mathrm{ext}}$ has priority over δ_{int}, λ(C2) = <OutC, y2c>, λ(C3) = <OutC, y3c>, ta(C1) = ∞, ta(C2) = 1t, ta(C3) = 2t, d(C1) = ∞, d(C2) = 4t, d(C3) = 5t.

The specifications of atomic models A and B are similar and straightforward.

For simplicity reasons, the state durations are considered small; however, in reality they are longer, compared to the execution time of the T_X, T_λ, and T_I tasks. In a RT system with large number of atomic components, overload conditions can happen at different points of time (when multiple T_X, T_λ, and T_I tasks from different atomic components collide, or are scheduled very close to each other), causing a drift in the execution of the tasks. An input *xa* enters the system from input port *In* at time zero, producing task T_{xa} (with 1t processing time), at time 1 (ie, 1t) the atomic component *A* transits from the initial state *A1* to *A2*. The life-time of state *A2* is 1t, thus at time 2, we run tasks $T_{\lambda 2}T_{I2}$, producing the output *y2a* (for simplicity reasons the outputs are not shown on the diagram in Figure 6) and causes an internal transition from *A2* to *A3*, (as specified in Figure 5B). The output produced by the atomic component *A* (*y2a*) via port *Out1A* is translated into an input for *B* (see port connections in Figure 5A). Thus, task T_{Xb} is executed right after $T_{\lambda 2}T_{I2}$, causing the atomic component *B* to change state from *B1* to *B2*. The rest of the system advances according to the specifications (in Figure 5). At t = 18, tasks $T_{\lambda 4}T_{I4}$ of *A*, $T_{\lambda 3}T_{I3}$ of *C*, and $T_{\lambda 2}T_{I2}$ of A, *B*, and *C* (shown in red in Figure 6) miss their deadlines because of the overrun condition in the system, causing these tasks to drift later than their deadlines.

In Figure 5B-D, the mandatory and optional states are marked with an *M* or an *O*, respectively. By applying the proposed early-reaction scheduling and considering a zero grace period (a perfect hard RT system), $T_{\lambda 3}$ of *A* is skipped (see Figure 7), because state *A3* is optional. Due to later than "release-time (*T3a*) plus zero grace period" execution of $T_{\lambda 3}T_{I3}$, the system detected the overrun situation, therefore it is able to save other mandatory tasks by shifting $T_{\lambda 4}T_{I4}$ to time 16. As discussed earlier, we use a zero grace period as this is the most stringent scenario, and we want to show how to deal with scheduling of the real time tasks for a hard RT application (ie, a system without a grace period in which we must guarantee every deadline). The time conserved due to discarding $T_{\lambda 3}$, also caused Tasks $T_{\lambda 3}T_{I3}$ of *B* and *C* as well as T_{xc} of C to be released after $T_{\lambda 4}T_{I4}$, hence shifting the execution of later tasks to earlier times, saving other mandatory output tasks. The same conditions happen for $T_{\lambda 3}$ of *B* and *C* at times 18 and 31. As a result, by discarding three optional T_λ tasks, the other three mandatory output tasks (tasks $T_{\lambda 4}T_{I4}$ of *A*, $T_{\lambda 3}T_{I3}$ of *C*, and $T_{\lambda 2}T_{I2}$ of *B*) are saved from lateness.

5 | CASE STUDY

The proposed I-DEVS formalism was implemented on E-CD++ an implementation of RT DEVS on the Xenomai RT framework. In this implementation, T_X tasks are made user configurable (ie, *periodic* or *aperiodic*), and their main job is to run user-defined input driver programs as soon as they are spawned. The main RT task implements the RT DEVS run-time abstract algorithm and takes care of T_λ and T_I tasks. This task is also responsible to implement and verify the

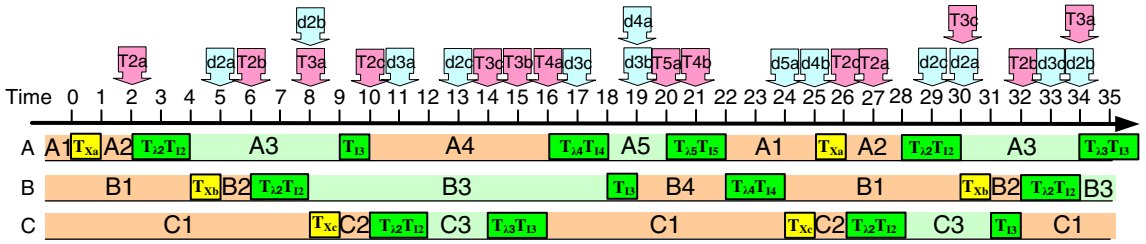
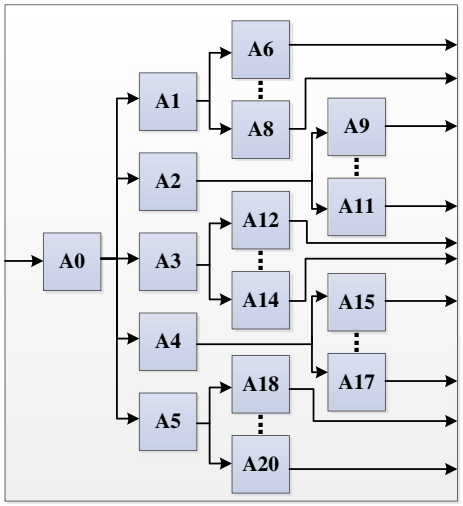
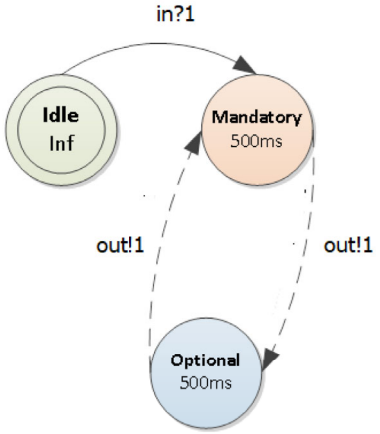


FIGURE 7 Applying imprecise computation to the sample scenario [Color figure can be viewed at wileyonlinelibrary.com]



(A) DEVS model hierarchy



(B) DEVS Graph atomic model behavior

FIGURE 8 Case study model architecture. A, DEVS model hierarchy; B, DEVS graph atomic model behavior [Color figure can be viewed at wileyonlinelibrary.com]

I-DEVS formalism and its execution. The implementation of the imprecise computation on E-CD++ is seamless and backward compatible (ie, the previous models also can be executed and are considered precise models).

Following, we discuss different scenarios we used to test the implementation of I-DEVS on E-CD++. As a proof of concept, we built a synthetic RT model with 20 atomic components each of them equipped with and input port and an output port. The synthetic model's design is represented in Figure 8A, where atomic components are connected to propagate the input to other components in order to create frequent flocks of T_λ and T_I tasks. This scenario causes overrun conditions due to the propagation of the events, which will be detected by I-DEVS execution engine and handled based on the proposed IC-based computation. The atomic components follow the DEVS-Graph diagram in Figure 8B. The model is composed of an initial state (with infinite lifetime), an optional state and a mandatory state. Whenever there is an input the model transitions from "Idle" to "Mandatory" state, where after 500 ms it produces an output and transitions to the "Optional" state (as this is a synthetic model, inputs and output values are also synthetic; in the DEVS Graph in the example, it receives and sends a value of 1). The model with then toggle between the "Mandatory" state and the "Optional" one, generating outputs every 500 ms. This model is used to perform comprehensive performance tests and compare the results of the imprecise execution and precise execution. In the case of precise execution, all the states are assumed mandatory by the system. In all the following test, an input was injected to the model after 1 second of execution to trigger the state changes and output production.

For the first test, the model was executed in both precise and imprecise modes, for different execution times. Figure 9 illustrates the average response time of mandatory tasks (all tasks in case of precise mode) for execution times of 20, 50, 70, 120, and 180 seconds with 10 seconds processing time for T_λ and 5 seconds for T_I. The charts show that average response time in imprecise mode is less than the precise mode, due to discarding the optional tasks as a measure of once overrun prevention. The diagram also shows the number of discarded optional T_λ tasks. This result shows the effectiveness of the early detection strategy in reducing the response time of the mandatory tasks. This test does not use explicit deadlines for the models, as we are trying to show the response times under different conditions (we create a synthetic overload scenario, and then study the response time for the precise and imprecise solutions,

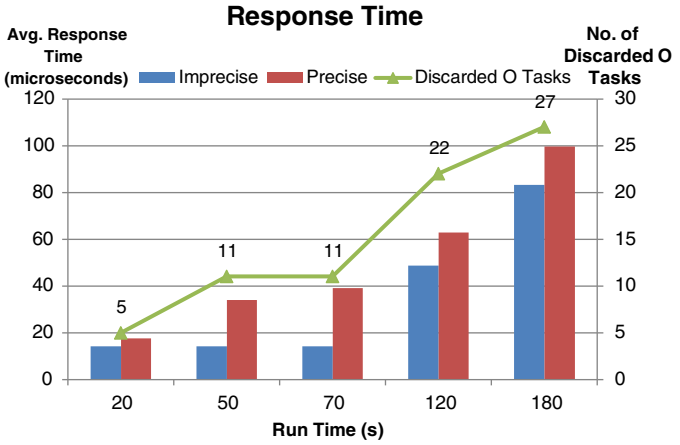


FIGURE 9 Average response time in different run times [Color figure can be viewed at wileyonlinelibrary.com]

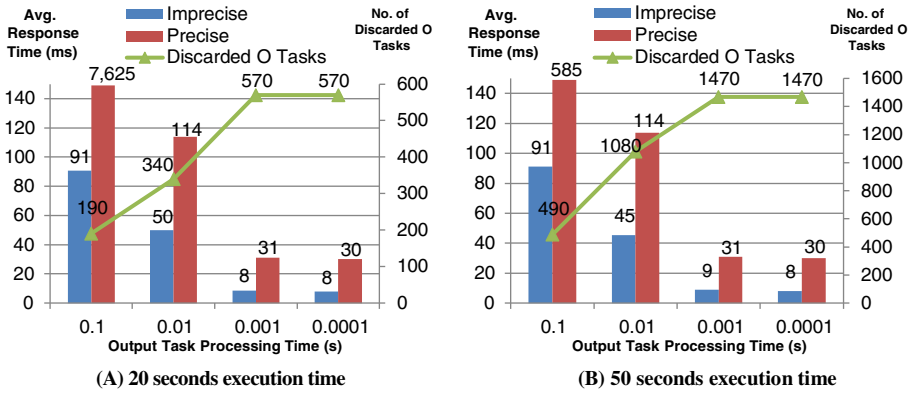


FIGURE 10 Average response time for different T_λ processing times. A, 20 seconds execution time; B, 50 seconds execution time [Color figure can be viewed at wileyonlinelibrary.com]

analyzing the number of tasks discarded, in order to see how the imprecise models can improve the number of tasks executed).

Figure 10 shows the average response time of mandatory tasks as well as the number of optional discarded T_λ task with zero processing time for T_I tasks and 100 ms, 10 ms, 1 ms, and 100 μs for T_λ task in 20, and 50 seconds execution time. Again, the chart shows the decline of response time of mandatory tasks when the imprecise processing is in effect. It is also observed that longer processing time of T_λ tasks causes more decline in average response time, which is related to more time that is saved by discarding optional T_λ tasks.

Figure 11 captures the response time of each T_λ task in precise and imprecise modes during an interval of about 2 seconds of the execution of the system. The grey areas depict the intervals when the early detection algorithm is engaged and starts discarding the optional tasks. The chart shows that, when the optional T_λ tasks are dropped (grey area) the response time of mandatory tasks also drops and stays flat around 10 ms until the high processing condition is relieved, while in precise mode the response time of tasks peaks to about 70 ms. Due to the periodic nature of the execution of T_λ task in this model, this situation happens periodically, as seen in the chart.

Figure 12 illustrates CPU utilization vs processing time of T_λ task in 20 and 50 seconds of execution time. The chart shows a meaningful decline in the CPU utilization in imprecise mode, due to the time saved by discarding optional T_λ tasks. Also, as the processing time of the T_λ tasks decreases, so the CPU utilization, because of the increase in the idle time (500 ms state lifetime) of the system proportional to the processing time of it. These tests show that running I-DEVS, the use of CPU is highly reduced, reducing the probability of overrun situations with a consistent and high margin of effectiveness.

5.1 | Performance evaluation

In a different test scenario, a synthetic model is used to measure and compare the performance and overhead of the execution of the imprecise vs precise models of the same scale. The tests are performed using two different sets of synthetic

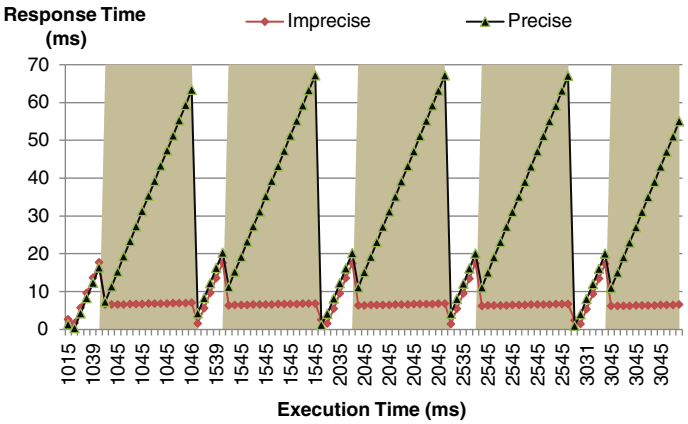
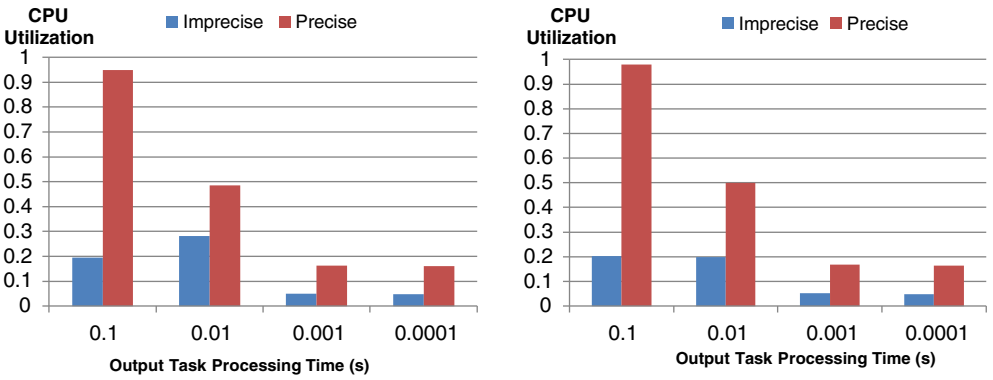


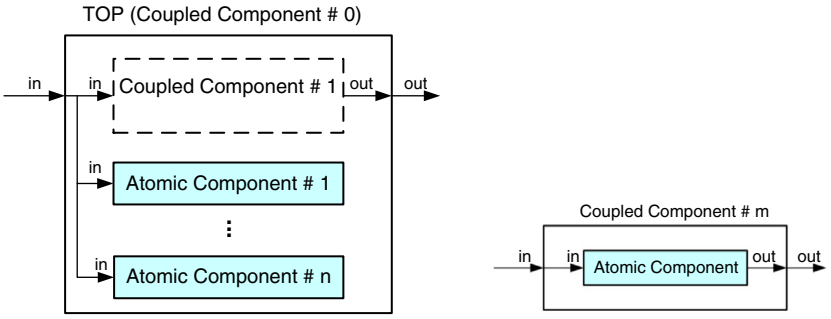
FIGURE 11 Response time of each T_λ task vs execution time [Color figure can be viewed at wileyonlinelibrary.com]



(A) 20 seconds execution time

(B) 50 seconds execution time

FIGURE 12 CPU utilization for different T_λ processing times. A, 20 seconds execution time; B, 50 seconds execution time [Color figure can be viewed at wileyonlinelibrary.com]



(A) (B)

FIGURE 13 Synthetic model architecture [Color figure can be viewed at wileyonlinelibrary.com]

models with different depth and width in the model hierarchy. The hierarchical model includes one coupled component and several atomic components in each level of the hierarchy. Figure 13A illustrates the Top coupled component along with its interconnections. As we can see, the coupled model includes *n* Atomic components, and one Coupled. Each of the Coupled models at each level in the hierarchy, has exactly the same structure as the one in Figure 12A. Figure 13B shows the last level, which only has one atomic component. The hierarchy has *m* levels. An input to this model propagates to each sub-component and is transmitted up to the last level. This will trigger the external function in each atomic component. All of the atomic components follow the same behavior, in which they are in a passive state (ie, state with ta(s) = infinity), until an input is received. The external transition (invoked by the input) changes the state to a temporary state with zero time-advance, which produces an output and then transitions to a passive state, waiting for the next input. This cycle continues as long as there is an input to the system.

The goal of this test is to measure the overhead of the processing occurred in the engine proportional to the processing time of the model. The overhead of executing a model is mainly associated with the abstract simulation algorithm's message transfer scheme, handling of input and message queues and the time-advance management. The major processing

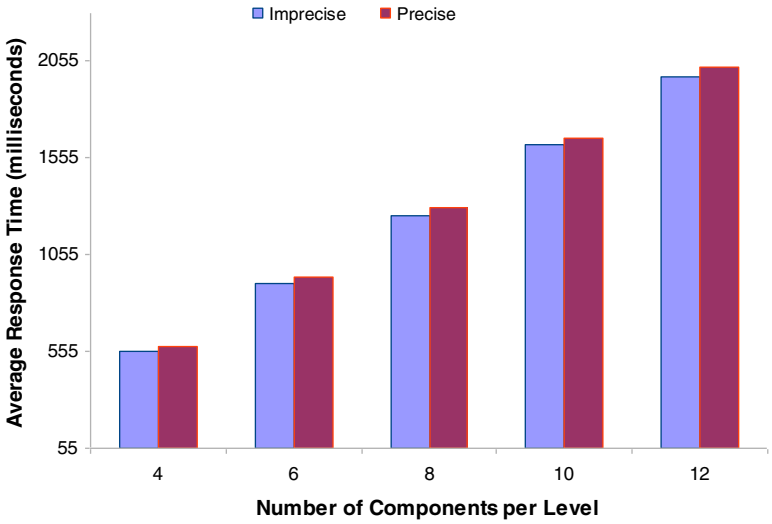


FIGURE 14 Number of components per level vs average response time [Color figure can be viewed at wileyonlinelibrary.com]

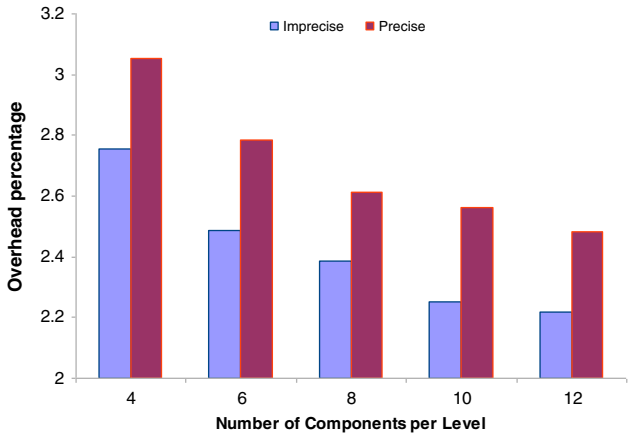


FIGURE 15 Number of components per level vs overhead percentage [Color figure can be viewed at wileyonlinelibrary.com]

in a DEVS model is performed in the external and internal transition functions in the atomic component. The percentage of overhead of the system relative to the model execution time is measured and compared. The percentage of software overhead is calculated using the following equation:

$$\text{Overhead}\% = \frac{\text{Total Processing Time} - \text{Total Transitions Processing Time}}{\text{Total Processing Time}} \times 100. \quad (2)$$

In this test, the models with fixed number of levels (four layers) and variable number of components in each level (ie, 4, 6, 8, 10, and 12) were used to measure the execution overhead and response time of the output tasks. The processing time of the T_X, T_λ, and T_I tasks was set to 10 milliseconds for each atomic component. Figure 14 shows the average response time of the tasks during an execution time of 40 seconds for different number of components per level. The output produced in each cycle of input, is marked as optional, therefore whenever the system faces an overrun condition, the T_λ tasks are skipped. The chart shows that the average response time of the tasks is slightly shorter in imprecise execution in each scenario, due to the time saved because of dropping the optional tasks. On the other hand, it is observed that, when the size of the model increases (the number of calls to the tasks also increases), the difference between the average response time of imprecise and precise runs, also increases. This is due to heavier workload produced in bigger models and propagation of data in the model, which is efficiently handled by the imprecise scheduling algorithm, reducing the response time of the tasks.

Figure 15 represents the overhead percentage (calculated using Equation (1)) of the execution engine relative to the tasks processing times, in imprecise and precise scenarios. The other parameters of the execution were the same as the previous test. Based on the results presented in this figure, the overhead percentage in imprecise mode is less than the one in precise mode, due to the drop of messaging overhead produced by the optional output tasks. A discarded output

task eliminates the overhead required for transfer of (@, t) and (done, t) messages from the atomic component to the Top model. The other interesting fact extracted from this diagram is the lower overhead percentage for heavier models (bigger modeling hierarchy with computation intensive tasks), due to the increase of the task processing time portion over the execution processing time. In other words, in a computation intensive model, as the size of the model grows the overhead percentage decreases, because the processor is mainly busy with the tasks rather than the execution overhead. The other interesting fact observed with this test is the lower overhead in the imprecise computation in heavy processing models. This is due to the relief caused by dripping optional tasks, while a precise model is processing tasks non-stop.

6 | CONCLUSIONS

We investigated a RT task model comprising of the DEVS intrinsic processes, which was integrated with Imprecise Computing in an innovative method, in which the model behavior is prioritized, allowing for efficient and dynamic task scheduling in the system. The overload management policy introduced in this framework provides an early reaction mechanism to transient overrun situations, saving critical outputs from lateness, preventing catastrophic results in the system.

The outcome of this research enables RT and embedded system designers to adopt an M&S-based approach, bridging the gap between simulation and RT software development. It also opens a new horizon towards model-based operating system design, allowing for creation of systems dedicated to running models as processes. All of our existing tools are open source and current updates can be found at https://github.com/SimulationEverywhere/.

RT systems working in the context of embedded hardware are prone to several limitations. One major constraint in these systems is the power consumption or battery life. High performance requirement in these systems conflicts with the low power objective. To achieve these goals performance degradation strategies can be incorporated. I-DEVS can be a natural choice for this purpose, providing a dynamic and early reaction scheme to tackle this problem. The graceful degradation strategy based on Imprecise Computations theory allows for degrading the system performance when needed by dropping the optional transitions. This threshold can include battery life or any other constraint conflicting with performance of the system.

On the other hand, performance of the system also depends on the underlying hardware. Figure 15 can be viewed as a scalability indicator in the current implementation of the I-DEVS approach on E-CD++ software. As the number of components per level increases, so does the average response-time. This means that the tasks are executed later to their release-times when the system scales up. Likewise, this delay also affects the deadline of the tasks, thus proposing a risk. A simple solution might include upgrading the underlying hardware resources in order to solve the scalability problem. As this will be a natural solution to this problem, however the "Speed-Performance Tradeoff Anomalies"[6] dilemma shows that in an RT system with timing and resource constraints, increasing the processor speed does not necessarily lead to a better performance, and vice versa.

The DEVStone benchmark we introduced in References 17 and 45, which has become a "de-facto" standard for testing DEVS environments,[47,42] could be used to test different scenarios, including models with multiple levels, or others with many subcomponents in a single layer, as done in References 47 and 45. The results presented here show that the imprecise computing method has potential to improve the performance in these cases, although further analysis is needed.

ORCID

Gabriel Wainer https://orcid.org/0000-0003-3366-9184

REFERENCES

1. Abrial JR. Formal methods in industry: achievements, problems, future. *Proceeding of the 28th International Conference on Software Engineering*. New York, NY: IEEE; 2006:761-768.
2. Amirijoo M, Hansson JS, Son H. Error-driven QoS management in imprecise RT databases. Paper present at: Proceedings of the 15th Euromicro Conference on RT Systems, July 2-4; 2003; Porto, Portugal.
3. Aydin H, Mejia-Alvarez P, Melhem R, Mossé D. Optimal reward-based scheduling of periodic RT tasks. *IEEE Trans Comput*. 1999;44(12):111-130.
4. Baruah S, Hickey M. Competitive on-line scheduling of imprecise computations. *IEEE Trans Comput*. 1998;47(9):1027-1032.
5. Blazewicz J, Ecker KH, Pesch E, Schmidt G, Sterna M, Weglarz J. Scheduling imprecise computations. *Handbook on Scheduling*. International Handbooks on Information Systems. Cham, Switzerland: Springer; 2019.
6. Buttazzo G. Achieving scalability in RT systems. *IEEE Comput*. 2006;39(5):54-59.

7. Cao K, Zhou J, Xu G, Wei T, Hu S. Exploring renewable-adaptive computation offloading for hierarchical QoS optimization in fog computing. *IEEE Trans Comput Aided Des Integr Circuits Syst*. 2019. https://doi.org/10.1109/TCAD.2019.2957374.
8. Cardoen B, Manhaeve S, Tuijn T, et al. Performance analysis of a PDEVS simulator supporting multiple synchronization protocols. *Proceedings of the Symposium on Theory of Modeling & Simulation*. Vol 4. San Diego, CA: SCS; 2016:1-8.
9. Chen X, Cheng AMK. An imprecise algorithm for RT compressed image and video transmission. *Proceedings of 6th International Conference on Computer Communications and Networks*. Las Vegas, NV: IEEE; 1997:390-397.
10. Chen JM, Lu WC, Shih WK, Tang MC. Imprecise computations with deferred optional tasks. *J Inf Sci Eng*. 2009;25(1):185-200.
11. Chishiro H, Yamasaki N. Practical imprecise computation model: Theory and practice. *IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. Vol 2014. Reno, NV: IEEE; 2014:198-205.
12. Chung JY, Liu JWS, Lin KJ. Scheduling periodic jobs that allow imprecise results. *IEEE Trans Comput*. 1990;39(9):1156-1174.
13. Feng W, Liu JWS. An extended imprecise computation model for time-constrained speech processing and generation. *Proceedings of the IEEE Workshop on RT Applications*. New York, NY: IEEE; 1993:76-80.
14. Finney K. Mathematical notation in formal specification: too difficult for the masses. *IEEE Trans Software Eng*. 1996;22(2):158-159.
15. Fujisawa K, Hayakawa S, Aoki T, Suzuki T, Okuma S. Real time motion planning for autonomous mobile robot, using framework of anytime algorithm. *Proceedings of the IEEE International Conference on Robotics & Automation*. Detroit, MI: IEEE; 1999:1347-1352.
16. Furfaro A, Nigro L. A development methodology for embedded systems based on RT-DEVS. *Innovations Syst Software Eng*. 2009;5(2):117-127.
17. E. Glinsky, G. Wainer. DEVSTONE: a benchmarking technique for studying performance of DEVS modeling and simulation environments. Proceedings of IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications DS-RT. Montréal, QC. IEEE/ACM 2005.
18. Hansson J, Thuresson M, Son S. Imprecise task scheduling and overload management using OR-ULD. *Proceedings of 7th International Conference on RT Computing Systems and Applications*. Cheju Island, South Korea: IEEE; 2000:307-314.
19. Hong JS, Song HH, Kim TG, Park KH. *A RT Discrete Event System Specification Formalism for Seamless RT Software Development*. Discrete Event Dynamic Systems. Kluwer Academic Publishers; 1997;7(4):355-375.
20. Huang X, Cheng AMK. Applying imprecise algorithms to RT image and video transmission. *Proceedings of RT Technology and Applications Symposium*. Chicago, IL: IEEE; 1995:390.
21. Kang BG, Seo K-M, Kim TG. Machine learning-based discrete event dynamic surrogate model of communication systems for simulating the command, control, and communication system of systems. *SIMULATION*. 2019;95(8):673-691. https://doi.org/10.1177/0037549718809890.
22. Kobayashi H, Yamasaki N. RT-Frontier: a RT operating system for practical imprecise computation. *Proceedings of the 10th IEEE RT and Applications Symposium*. Toronto, Canada: IEEE; 2004:255-264.
23. Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard RT environment. *J ACM*. 1973;20(1):46-61.
24. Liu JWS, Lin KJ, Shih WK, Chung JY, Yu A, Zhao W. Algorithms for scheduling imprecise computations. *IEEE Trans Comput*. 1991;24(5):58-68.
25. Liu JWS, Shih W, Lin KJ, Bettati R, Chung J. Imprecise computations. *Proc IEEE*. 1994;82(1):83-94.
26. Liu JWS, Lin KJ, Bettati R, Hull D, Yu A. Use of imprecise computation to enhance dependability of RT systems. *Int Ser Eng Comput Sci*. 1994;284(3):157-182.
27. Liu JWS, Shih WK. Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *IEEE Trans Comput*. 1995;44(3):466-471.
28. Liu JWS. *RT Systems*. Upper Saddle River, NJ: Prentice-Hall; 2000. ISBN:0‐13‐099651‐3.
29. de Mello BA, Wainer G. Scheduling predictability in I-DEVS by schedulability analysis. *Proceedings of 2016 SCS/ACM/IEEE Symposium on Theory of Modeling and Simulation, TMS'16*. Pasadena, CA: IEEE; 2016.
30. Bocciarelli P, D'Ambrogio A, Falcone A, Garro A, Giglio A. A model-driven approach to enable the simulation of complex systems on distributed architectures. *SIMULATION*. 2019;95(12):1185-1211. https://doi.org/10.1177/0037549719829828.
31. Moallemi M, Wainer GA. I-DEVS: imprecise real-time and embedded DEVS modeling. *Proceedings of ACM/IEEE/SCS TMS*. Orlando, FL: IEEE; 2011.
32. Moallemi M, Wainer GA. Modeling and simulation-driven development of embedded RT systems. *Simul Model Pract Theory*. 2013;38:115-131.
33. Monin JF, Hinchey MG. *Understanding Formal Methods*. London: Springer; 2003. ISBN:1852332476.
34. Nicolescu G, Mosterman PJ. *Model-Based Design for Embedded Systems*. London: CRC Press; 2010. ISBN:978‐1‐4200‐6784‐2.
35. Parker GB. Punctuated anytime learning for hexapod gait generation. *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and System*. Vol 3. Beijing, China: IEEE; 2002:2664-2671.
36. Camus B, Paris T, Vaubourg J, Presse Y, Bourjot C, Ciarletta L, Chevrier V. Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware. *SIMULATION*. 2018;94(12):1099-1127. https://doi.org/10.1177/0037549717749014.
37. Saadawi H, Wainer G. *Verification of RT DEVS Models*. San Diego, CA: Proceedings of DEVS Symposium; 2009.
38. Shih WK, Liu JWS, Chung JY. Algorithms for scheduling imprecise computations with timing constraints. *SIAM J Comput*. 1991;20(3):537-552.
39. Shih WK, Liu JWS. On-line scheduling of imprecise computations to minimize total error. *Proceedings of the 13th IEEE RT Systems Symposium*. Phoenix, AZ: IEEE; 1992:280-289.

40. Shih WK, Liu JWS. On-line algorithms for scheduling imprecise computations. *SIAM J Comput*. 1996;25(1):1105-1121.
41. Song HS, Kim TG. Application of RT DEVS to analysis of safety-critical embedded control systems: railroad crossing control example. *Simulation*. 2005;81(2):119-136.
42. Van Tendeloo Y, Vangheluwe H. An evaluation of DEVS simulation tools. *Simulation*. 2017;93(2):103-121. https://doi.org/10.1177/0037549716678330.
43. Vicino D, Niyonkuru D, Wainer G, Dalle O. Sequential PDEVS architecture. *Proceedings of 2015 SCS/ACM/IEEE Symposium on Theory of Modeling and Simulation, TMS'15*. Arlington, VA: SCS; 2015.
44. Wainer GA. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. London: CRC/Taylor & Francis; 2009.
45. Wainer G, Glinsky E, Gutiérrez-Alcaraz M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simulation: Trans Soc Model Simul Int*. 2011;87(7):555-580.
46. Wiedenhoft GR, Fröhlich AA. Using imprecise computation techniques for power management in RT embedded systems. *Proceedings of 6th IFIP Working conference on Distributed and Parallel Embedded Systems*. Milano, Italy: IFIP; 2008:121-130.
47. Risco-Martín JL, Mittal S, Fabero JC, Zapater M, Hermida R. Reconsidering the performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simulation*. 2017;93(6):459-476.
48. Zeigler B, Kim T, Praehofer H. *Theory of Modeling and Simulation*. New York: Academic Press; 2000. ISBN:0127784551.
49. Zhou J, Yan J, Wei T, Chen M, Hu XS. Energy-adaptive scheduling of imprecise computation tasks for QoS optimization in real-time MPSoC systems. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Lausanne, Switzerland: IEEE; 2017:1402-1407.

How to cite this article: Wainer G, Moallemi M. Designing real-time systems using imprecise discrete-event system specifications. *Softw: Pract Exper*. 2020;1–18. https://doi.org/10.1002/spe.2831