AN ABSTRACT DISCRETE-EVENT SIMULATOR CONSIDERING INPUT WITH UNCERTAINTY

Damián Vicino
Gabriel Wainer
Carleton University
Dept. of Systems and Computer Engineering
{dvicino,gwainer}@sce.carleton.ca

Olivier Dalle
Université Nice Sophia Antipolis
I3S UMR CNRS 7271
olivier.dalle@unice.fr

ABSTRACT

A timeline in Discrete-Event Simulation (DES) is a sequence of events defined in a numerable subset of $\mathbb{R}^+$. Discrete-Event Modeling and Simulation try to reproduce the behaviour of real-world experiments. Nevertheless, measuring the experimental data for science and engineering (which is later used for modeling and simulation) introduces the need for uncertainty quantifications. In modeling of Continuous Systems, numerous tools have been defined to propagate uncertainty from the input to the output results. Nonetheless, these tools cannot be applied to the study of propagation of uncertainty in DES. In a previous study, we presented a method for propagating uncertainty for subset of DES models. Here, we introduce a generalization of the previous method that can be used for simulating any DES model. The method defines a new Abstract Simulator for the Discrete-Events System Specification (DEVS). This new Simulator provides the propagation of uncertainty from input to resulting trajectories.

Keywords: DEVS, Uncertainty, Metrology, Time.

1 INTRODUCTION

Discrete-Event Simulation (DES) is a technique in which the simulation engine plays a history of events. Each event occurring corresponds to an instantaneous change in the state of the model. The set of events of the simulation is discrete, and occurrences can be freely placed at any point of a continuous timeline. Events in the history are related by causality: changing a single event in the history may produce an arbitrary cascade of changes in subsequent events, with unbounded consequences. In particular, we discussed errors related to time approximation in (Vicino 2015) (Vicino, Dalle, and Wainer 2015). We classifed these errors in three groups : time-shifting, event-reordering, and Zeno problem, and showed that any of these errors could break causality chains resulting in simulation trajectories that diverge.

DES has been widely used for decision-making in science and engineering. As part of the experimentation process, data are collected from real systems using a variety of measuring instruments and measurement procedures. From them, a set of measurement results are obtained and used to define the models and run the simulations.

Measurement results have uncertainty specifications (Joint Committee for Guides in Metrology 2012), usually represented by uncertainty intervals. Using an input with uncertainty may translate into uncertain results. Uncertainty propagation is the science of studying how the uncertainty of the input can affect the uncertainty of the outputs. Various mathematical tools were developed for uncertainty propagation on Continuous Systems (Hoover 1999). Nevertheless, DES are described by discontinuous functions (with excep-

SpringSim-TMS/DEVS 2017, April 23-26, Virginia Beach, VA, USA

tion of trivial models), and existing tools and methods for continuous systems cannot be directly applied to the study of DES.

In the case of Discrete-Time Systems (DTS), since the timeline and the states are discrete, it is possible to distinguish a finite number of steps for every trajectory produced by a set of inputs. Therefore, an uncertainty interval covers a finite number of values. We can obtain these finite steps by running the simulation using each possible input occurrence within the uncertainty interval. Unfortunately, this method cannot be used for studying uncertainty propagation in DES, where time is a Real variable, because all uncertainty intervals include an infinite number of values. To the best of our knowledge, no general method for propagating uncertainty in DES was defined before.

Although these concepts are important for all kinds of discrete-event simulation techniques, we are interested in applying these ideas to the Discrete-Event System Specification (DEVS) formalism (Zeigler, Praehofer, and Kim 2000). DEVS provides a theoretical framework that is universal for DES models, and it allows thinking about models using a hierarchical-modular approach. In addition, DEVS models are described using a formal notation.

It is possible, using DEVS notation, to define states and functions including uncertainty. We avoid this approach, because it requires the propagation mechanisms to be defined as part of each model. This over-complicates the model, and breaks the abstraction between model and simulator. In the case of occurrence uncertainties, it is not even possible to take similar approach. Time lapses representation is strictly defined in DEVS as positive reals. An option would be to modify this constrains in a redefinition of the formalism.

In (Vicino 2015) (Vicino, Dalle, and Wainer 2015), we defined algorithms and proposed a method for propagating uncertainty in a subclass of DES models. The subclass covered by the method was named Finite-Forkable Discrete-Event System Specification (FF-DEVS), which is reviewed in Section 2.2.

The main goal of this new research is to present a general uncertainty propagation method for DES simulations based on DEVS, without needing to define a new subclass. Furthermore, we want to preserve the DEVS formalism unchanged, such that our method can be applied to the large base of existing DEVS models and tools to support uncertain inputs.

To do so, we propose a new abstract simulator for DEVS. In this simulator, exogenous events accept uncertainty quantifications. This uncertainty can be introduced on both components of an event, the message and the time of occurrence. The method combines two techniques for advancing simulation steps. First, when processing the next event only faces a few options, the simulation is branched in order to cover all of them. Second, in cases where the possibilities are infinite, or too nymerous, we apply bound analysis techniques. This guarantees that we produce a finite number of branches at each simulation step. At every step events have the uncertainty quantification required for the resulting trajectories.

This new simulator introduces simulation errors, but with the good property that they always produce supersets of the expected trajectories. This is consistent with existing uncertainty propagation methods used for continuous systems.

In addition to the algorithms, we present a case study showing how to simulate the Classic Processor model described in (Zeigler, Praehofer, and Kim 2000) with uncertainty in the input jobs numbers and occurrences. The rest of the paper is organized as follows: Section 2 introduces the background and terminology used in following sections; Section 3 reviews related works; Section 4 introduces the proposed simulation algorithms; Section 5 presents a case study using a Processor model; Section 6 concludes the paper with final remarks and future lines of work being considered.

2 BACKGROUND

2.1 Discrete-Event System Specification (DEVS)

Discrete-Event System Specification (Zeigler, Praehofer, and Kim 2000) (DEVS) is a universal formalism for modeling Discrete-Event Systems. This formalism provides a theoretical hierarchical modeling language and an abstract simulator to simulate legit DEVS models.

In DEVS, the modeling hierarchy has two kinds of components: atomic models and coupled models. The atomic models are defined as a tuple: $A = \langle S, X, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where: X is the set of inputs; Y is the set of outputs; S is the set of states; $\delta_{int} : S \to S$ is the internal transition function; $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state set (where e is the time elapsed since last transition); $\delta_{ext} : Q \times X \to S$ is the external transition function; $\lambda : S \to Y$ is the output function; $ta : S \to \mathbb{R}^+$ is the time advance function.

The Processor atomic model is a classical example in the bibliography. A processor receives jobs to be performed; each job takes a fixed period to be executed. In case the processor is busy and a new job is received, the new job is queued until the current job is complete. Each job completion generates an output.

A $Processor_{process\_duration}$ can be defined as a tuple: $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$, where: $X = \mathbb{N}$; $Y = \mathbb{N}$; $S = \langle TOCJ, QJ \rangle$ where: $TOCJ$ represents the time ($\mathbb{R}^+$) until current job finishes processing, and QJ is a queue of processes identified by natural numbers; $\delta_{int}(S) = \langle process\_duration, S.QJ.dequeue\_first \rangle$; $\delta_{ext}(S, e, X) =$ if $(S.QJ.size = 0)$ then $\langle process\_duration, S.QJ.queue\_all(X) \rangle$ else $\langle S.TOCJ - e,$ $S.QJ.queue\_all(X) \rangle$; $\lambda(S) = S.QJ.first$; $ta(S) =$ if $(S.QJ.size = 0)$ then ∞ else $S.TOCJ$

Coupled models are used to compose models hierarchically in DEVS. A coupled model is defined by a set of DEVS sub-models and a description of their interactions.

A benefit of modeling using DEVS formalism is its closure under coupling property. This property states that each coupled model can be reduced to an equivalent atomic model. Some simulator implementations exploited this closure property, being the most noticeable aDEVS (Nutaro 2003).

2.2 Finite-Forkeable DEVS (FF-DEVS)

Several extensions have been proposed to DEVS. These extensions include: Parallel DEVS (PDEVS) (Chow and Zeigler 1994) targeting problems of serial computation caused by the SELECT function; Finite & Deterministic DEVS (Hwang and Zeigler 2006) to study of all reachable state based in graph theory are provided; Finite-Forkable DEVS (FF-DEVS) (Vicino 2015) (Vicino, Dalle, and Wainer 2015) targeting the propagation of uncertainty in a well-known subset of DEVS models; and many others.

The introduction of a single event with uncertainty can produce infinite resulting trajectories. The δ_{ext} function is involved in the processing of external events. This function is sensible to occurrence of the input. In some cases, the evaluating multiple occurrence times would determine multiple, sometimes infinite, new states. In others, state could be uniquely determined, but scheduling next internal transition is uncertain. This is because time-advance is added to current time, which is defined by the uncertainty of the input event.

The FF-DEVS algorithms branch the simulation for each possible sequence of states in a trajectory. This groups all the infinite placements of a sequence of states in the timeline together. The FF-DEVS simulations generate a tree-like summary of all the possible trajectories. Each path from the root to a leaf of the tree represents set of trajectories with a common sequence of states. Each node in the tree adds to the state the respective occurrence of uncertainty intervals. The main idea behind the algorithms is detecting the times when the same sequences of events are generated. We study the overlap of uncertainty of the occurrence between events. In addition, the algorithm checks the cardinality of the transition results. Some combination of models and inputs may produce infinite branches. For instance, the Processor model described in 2.1

could produce a single sequence of states for certain inputs. But, it could reach infinite number of states in a single step of simulation for other inputs, as discussed in (Vicino 2015).

2.3 Metrological uncertainty

Metrology is the science of measurements and its applications. A True Value of a measure, i.e., a single value obtained with perfect accuracy, is unknowable in practice. Thus, in metrology, the Uncertainty Approach is used. This approach represents the result of measuring a magnitude as an interval. This interval includes all reasonable values that could be assigned to the measurand.

The Bureau International des Poids et Mesures (BIPM) is the institution responsible for the standardization of international units, procedures and practices for proper measurement in industry and sciences (BIPM 2008) (Joint Committee for Guides in Metrology 2012).

Sometimes, obtaining the uncertainty specification of a measurement may involve probabilistic methods. However, no distribution of values in an uncertainty interval could be assumed.

2.4 Zermelo's Well-Ordering Theorem

In DEVS models, states and messages are defined as sets (usually named X, Y, S). And, the dynamic behavior of the model is described by functions (δ_{ext}, δ_{int}, and λ) over these sets. Many Set Theories exist nowadays. We follow in this paper Zermello-Fraenkel with axiom of Choice (ZFC).

An interesting result from ZFC is the Well-Ordering Theorem (Jech 2003). This theorem states: "Every set can be well-ordered". A proof of this theorem can be found in (Jech 2003).

dsWe relay on this theorem to guarantee that the order function parameters required for the proposed simulator exist for every DEVS model. Then, any model can be simulated for input with uncertainty.

3 RELATED WORK

The closer match to our work are studies on state reachability tools and properties. In (Hernandez and Giambiasi 2005), state reachability is studied for a subset of DEVS models having a finite set of states. An algorithm is provided to decide if a state is reachable or not when simulating a model. In (Hwang and Zeigler 2006), Finite & Deterministic DEVS is formally defined and algorithms to study of all reachable state based in graph theory are provided. In (Hwang and Cho 2004), Scheduling Preserved DEVS is introduced as a subclass of FD-DEVS. A timed language is also introduced that allows characterization of reachable states in a timed fashion for first time. All these solutions focus on the validation of the model properties and not on the reachability based on measured data; none of them consider the introduction of uncertainty on the time of occurence of events, making this the principal distinction from our work.

Other works related to uncertainty in DES are those in the area of Logical Processes Simulation proposing the use of Approximated Time. In these works (Beraldi, Nigro, and Orlando 2003) (Fujimoto 1999) (Loper and Fujimoto 2004), the main goal is to speed up simulation exploiting uncertainty. The approach is taking models defined with perfect precision and introduce small uncertainty to the time points in the time-line to improve the parallelism of the simulation. In other words, the idea is to choose an approximation of the initial simulation that fits in the uncertainty boundaries and results in a better parallel execution. These works are far from our goal. On the contrary, we focus on obtaining all possible trajectories that may result from input events having their time of occurrence defined by uncertainty intervals.

As part of our solution we propose forking the simulation at some points of the time-line. This approach is not new in simulation. Some examples of its use include (Peschlow and Martini 2007) (Hybinette and Fujimoto 1997) (Hybinette and Fujimoto 2002). In this paper we don't go into the details of how the

simulation is forked. The afore-mentioned papers are good starting points for an implementation of our algorithms. However some adaptation is necessary given neither of them is based on DEVS.

4 PROPOSED ABSTRACT SIMULATOR

We propose here, an alternative abstract simulator for DEVS models. Different to the classical one, this allows the insertion of events as inputs. These input events can be specified with uncertainty quantifications in both of their components, occurrence and message.

Having uncertainty can make the simulation traces of both output and state diverge. For this reason, the result of the simulation cannot be represented as sequence of states nor outputs. We decided then to produce result as a tree of states and a tree of outputs. Each node in the trees has associated uncertainty quantifications. The uncertainty could affect, the state or output and its occurrence. The uncertainties obtained reflect the propagation coming from input events uncertainty.

For simplicity, we present a simulator for atomic models only. This does not limit the generality of the method given that any model could be expressed as atomic because of the closure under coupling property.

We will discuss the problems solved by the simulator algorithms by dividing them into three groups. First, handling the propagation of uncertainty for each transition executed. Second, processing and input events with uncertainty into the simulator. Third, dealing with uncertainty interval overlaps.

The input parameters for our simulator are the following: the DEVS model being simulated; the set of input events with their occurrences and messages defined as uncertainty intervals; an order function over the set of states and set of messages defined in the model; an initial state for the model, which could have uncertainty associated; a limit to how many branches per step are accepted.

4.1 Abstract simulator for DEVS atomic models

In Algorithm 1, we present the Abstract Simulator for atomic models accepting events with uncertainty intervals for their time of occurrence. The simulator structure is similar to the Classic-DEVS abstract simulator presented in (Zeigler, Praehofer, and Kim 2000).

Internal variables
Every variable of the simulator representing time, state, or messages are intervals. Four properties define each interval. Two properties define the lower and upper limits, and two (boolean) properties are used for defining if each end is open or closed.

For starting the simulator, six parameters are required: the model to be simulated; order functions over the state and output sets of the model; initial time of the simulation; initial state of the simulation; and the max allowed branches for a simulation step.

The order functions are required to define lower bound (LB) and upper bound (UB) functions. The actual definition of LB and UB are implementation details. The most effective LB would be the infimum, and the most effective UB would be the supremum. However, they may be hard to compute and more relaxed bounds can still produce good results for particular scenarios. It is important to notice that the relaxation will only include new values to the solution. It never excludes a valid solution from the results' tree.

The s internal variable tracks the current state of the simulation. Its initial value is set in the *init* function defined by an uncertainty interval over S. The need to produce intervals of states is why we require order functions over them. The selection of an order function for the state set may not be trivial. However, in Section 2.4, we show the Well-Ordering Theorem that guarantees it is always possible to find an ordering

Data: *Atomic model* A, $Order\langle S\rangle$ os, $Order\langle Y\rangle$ oy, $Interval\langle\mathbb{R}^+\rangle$ init_time, $Interval\langle A :: State\rangle$ init_state, *Integer* max_branches_per_step

$Interval\langle\mathbb{R}^+\rangle\ t_{last}$ `// time of last event`
$Interval\langle\mathbb{R}^+\rangle\ t_{next}$ `// time of next event`
A::state s `// current state`
Function *init-message*$() \rightarrow$ *void*
  $t_{last} \leftarrow init\_time$
  $s \leftarrow init\_state$
  $s \leftarrow split\_state\_set(s)$
  $t_{inc} \leftarrow [LB\langle\mathbb{R}^+\rangle(A.ta(m)|m \in s), UB\langle\mathbb{R}^+\rangle(A.ta(m)|m \in s)]$
  $t_{inc}.open\_lowerend \leftarrow t_{inc}.lowerend \notin (A.ta(m)|m \in s)$
  $t_{inc}.open\_upperend \leftarrow t_{inc}.upperend \notin (A.ta(m)|m \in s)$
  $t_{next} \leftarrow t_{last} + t_{inc}$
Function $*$*-message*$(Interval\langle\mathbb{R}^+\rangle\ t) \rightarrow Y$
  if $\neg(t \subseteq t_{next})$ **then** raise an error
  $Interval\langle Y\rangle\ y \leftarrow [LB\langle os\rangle(A.\lambda(m)|m \in s), UB\langle os\rangle(A.\lambda(m)|m \in s)]$
  $s \leftarrow [LB\langle os\rangle(A.\delta_{int}(m)|m \in s), UB\langle os\rangle(A.\delta_{int}(m)|m \in s)]$
  $s \leftarrow split\_state\_set(s)$
  $t_{last} \leftarrow t$
  $t_{inc} \leftarrow [LB\langle\mathbb{R}^+\rangle(A.ta(m)|m \in s), UB\langle\mathbb{R}^+\rangle(A.ta(m)|m \in s)]$
  $t_{inc}.open\_lowerend \leftarrow t_{inc}.lowerend \notin (A.ta(m)|m \in s)$
  $t_{inc}.open\_upperend \leftarrow t_{inc}.upperend \notin (A.ta(m)|m \in s)$
  $t_{next} \leftarrow t_{last} + t_{inc}$
  return y
Function *x-message*$(Interval\langle X\rangle\ x, Interval\langle\mathbb{R}^+\rangle\ t) \rightarrow$ *void*
  $Interval\langle\mathbb{R}^+\rangle\ t_{local}$
  $t_{local}.upperend \leftarrow t.upperend - t_{last}.lowerend$
  $t_{local}.open\_upperend \leftarrow t.open\_upperend \vee t_{last}.open\_lowerend$
  if $t \cap t_{last}$ **then** $t_{local}.lowerend \leftarrow 0$, $t_{local}.open\_lowerend \leftarrow False$
  else
    $t_{local}.lowerend \leftarrow t.lowerend - t_{last}.upperend$
    $t_{local}.open\_lowerend \leftarrow t.open\_lowerend \vee t_{last}.open\_upperend$
  $s = [LB\langle os\rangle(A.\delta_{ext}(m,a,b)|m \in s, a \in t_{local},$
   $b \in x.message), UB\langle os\rangle(A.\delta_{ext}(m,a,b)|m \in s, a \in t_{local}, b \in x.message)]$
  $s.open\_lowerend = s.lowerend \notin (A.\delta_{ext}(m,a,b)|m \in s, a \in t_{local}, b \in x.message)$
  $s.open\_upperend = s.upperend \notin (A.\delta_{ext}(m,a,b)|m \in s, a \in t_{local}, b \in x.message)$
  $s \leftarrow split\_state\_set(s)$
  $t_{last} \leftarrow t$
  $t_{inc} \leftarrow [LB\langle\mathbb{R}^+\rangle(A.ta(m)|m \in s), UB\langle\mathbb{R}^+\rangle(A.ta(m)|m \in s)]$
  $t_{inc}.open\_lowerend \leftarrow t_{inc}.lowerend \notin (A.ta(m)|m \in s)$
  $t_{inc}.open\_upperend \leftarrow t_{inc}.upperend \notin (A.ta(m)|m \in s)$
  $t_{next} \leftarrow t_{last} + t_{inc}$
Function *split_state_set*$(Interlva\langle S\rangle\ s) \rightarrow$ *void*
  if $\#s \leq max\_branches\_per\_step$ **then**
    forall $v \in s$ **do**
      if *On forked child* **then return** v **else** EXIT
  else return s

Algorithm 1: Abstract Simulator for atomic models allowing input with occurrence uncertainty

function for the state set. The chosen function can affect significantly the uncertainty intervals associated to the result nodes. Same applies for input and output messages.

Two internal variables track simulation chronology. The t_{last} variable tracks the time of the last event processed by the simulator. And t_{next} tracks the next scheduled internal transition. The initial time received in the *init* function is used for initial t_{last}. The initial t_{next} is defined using time advance function over the initial state and the initial t_{last}. In case model is in passive state t_{next} is set to $\emptyset$ as convention.

In addition, a limit of branches per simulation step is needed. Based on it, the simulator decides how to advance the simulation. If current set of states are not above the limit, it branches the simulation for each of them, this produces exact advances. Else, it will use bound analysis to advance.

Init simulation
The initialization is introduced by the *init-message* function. This function reads the parameters, sets up the variables, and calls $split\_state\_set$. This function checks the maximum number of branches per step limit against the cardinality of the current state set. In case that the state set is low on values, it branches one new simulation per state value. After the state values are set, the time until next scheduled transition needs to be computed. To do that, we apply the ta function to the set of new states and construct an interval using lower and upper bounds of the obtained results. The interval obtained (t_{inc}) provides a relative increment of time, adding it to t_{last}, we obtain its absolute value (t_{next}).

Internal transitions
The **-message* function processes the generation of output and internal transition advances. First, the output is obtained by applying λ to the set described by the uncertainty interval of the current state. Using the lower and upper bounds of the set of outputs obtained, an uncertainty interval is constructed and returned. Second, δ_{int} is applied to the set of states described by the uncertainty interval of the current state. Using the lower and upper bounds of the set of states obtained, an uncertainty interval is constructed and set as the new state. Third, the $split\_state\_set$ function is called to branch simulation in case the set of states is finite and below the threshold. Finally, the time for next scheduled transition is computed into t_{next}.

External transitions
The *x-message* function processes the input of events into the simulated atomic model. For simplicity, multiple input events with conflicts are handled at the Main-loop level. Here, we only work with one event at the time and we assume it is not competing with any other one.

We start by applying δ_{ext} to the values and occurrences defined by the input uncertainty intervals. Using the lower and upper bounds of the obtained set of results, we define the new state uncertainty intervals. Then, we call the $split\_state\_set$ function to branch the simulation if the results are finite and below the threshold. Finally, we set the time variables required for next simulation step.

4.2 Main-loop for DEVS simulators handling uncertainty quantifications

The main-loop algorithm takes care of advancing the simulation and feeding the simulator with input events. The input is an ordered queue of events. Each event contains a message and a time of occurrence. The message is any non-empty uncertainty interval over model's input set X. The time is any non-empty interval in $\mathbb{R}^+$. The main loop branches the simulation when events conflict with each other. This loop runs until every input in the queue is consumed and the model reaches a passive state.

The input queue order is defined by the following criteria: the interval having a value lower than any value in the other interval goes before the other, and in the case of a draw, the one having a value greater than any value in the other goes after the other. For example the interval [1, 3] is before [2, 4], because 1 is lower than any value in [2, 3]. And, [1, 3] is before [1, 4] because 4 is higher than every value in [1, 3]. We note this order as $\ll$ in the algorithms.

The main-loop, showed in Algorithm 2, receives the same 6 parameters as the Simulator, and a queue of events. A simulator for the model is created using the first 6 parameters (s), and the loop iterates until the input queue is empty and the model is passivated. This algorithm is similar to the one used for simulating FF-DEVS. The main difference is the type of messages, which include uncertainty quantifications.

Every time main-loop iterates, we advance a simulation step. For advancing, we can call the *x-message* function, or the **-message* function. What we chose depends on the current schedule of internal transition, and the events in the input queue. In some cases, it is easy to decide, for example if the queue is empty, and an internal transition was scheduled, we call the **-message* function. In other cases, overlaps and order of events can make the choice more difficult. Then, we classify each combination of current queue of input events, and the next scheduled internal transition, as belonging to any of three possible scenarios: **No-Collision**, **Input-Collision**, or **Scheduled-Collision**.

In the **No-Collision** scenario, a scheduled transition or an input event is ready to be processed, and its time uncertainty interval does not intersect with any other event. In the **Input-Collision** scenario, the first event in the input queue overlaps its time uncertainty interval with another event in the queue, or with the scheduled internal transition. In addition, at least one value in the uncertainty interval of the first event of the queue is before any value in the scheduled internal transition. In the **Scheduled-Collision** scenario, the uncertainty interval of the first event in the queue overlaps with the one for the scheduled internal transition. In addition, there is at least one value in the uncertainty interval of the scheduled internal transition before any value in the first event of the input queue. Depending on the scenario detected on Algorithm 2, the strategy to advance current simulation step follows Algorithms 3, 4, or 5.

Data: *Atomic model* A, $Order\langle A :: State\rangle$ os, $Order\langle A :: Y\rangle$ oy, $Interval\langle\mathbb{R}^+\rangle$ init_time, $Interval\langle A :: State\rangle$ init_state, *Integer* max_branches_per_step, $Queue\langle Interval\langle A :: X\rangle\rangle$ input_events
Simulator $s \leftarrow Simulator(A, os, oy, init\_time, init\_state, max\_branches\_per\_step)$
$s.init\text{-}message(t_{init})$
while $input \neq \emptyset \vee s.t_{next} \neq \emptyset$ **do**
  if $No\_Collision(s.t_{next}, input)$ **then** Check Algorithm 3
  if $Input\_Collision(s.t_{next}, input)$ **then** Check Algorithm 4
  if $Scheduled\_Collision(s.t_{next}, input)$ **then** Check Algorithm 5

Algorithm 2: Main-loop for coordinating simulation using measured input

if $input \neq \emptyset \wedge input.front.time \ll s.t_{next}$ **then**
  $s.x\text{-}message(input.front.message, input.front.time)$
  $input.pop()$
else $s.*\text{-}message(s.t_{next})$

Algorithm 3: Main-loop advancing on a No-Collision scenario

Having the Order functions as parameter of the simulation allows their usage for results refinement. Running the same model using the same input with different order functions produces different trajectory-trees. Booth trees produce supersets of the set of expected trajectories. Then, intersecting them could reduce the error introduced by the bound analysis steps.

5 CASE STUDY

Here, we present the execution of a Processor model that receives events with uncertainty in both, value and occurrence. As mentioned in Section 2.2, the Processor model cannot be simulated using the FF-DEVS Simulator. The parameters of the simulation are described in Table 1.

When using the algorithms presented in Section 4 for this model, we obtain the Figure 1 trajectories.

$Interval\langle\mathbb{R}^+\rangle$ *bound* `// time slice for advancing simulation`
$\mathbb{R}^+$ *limit* `// Time limit used for conflict resolution`
if $\exists x \in input : x \neq input.front \wedge x.time \ll s.t_{next}$ **then**
  $limit \leftarrow x.time.upperend : (x \in input \wedge \forall y \in input, x.time.upperend \leq y.time.upperend)$
  $bound \leftarrow [input.front.time.lowerend, limit]$
  $bound.open\_upperend \leftarrow (\exists x \in input, bound.upperend = x.upperend \wedge x.open\_upperend)$
else
  $limit = s.t_{next}.lowerend$
  $bound \leftarrow [input.front.time.lowerend, s.t_{next}.lowerend]$
  $bound.open\_upperend \leftarrow \neg s.t_{next}.open\_lowerend$
$bound.open\_lowerend \leftarrow input.front.time.open\_lowerend$
forall $x : x \in input \wedge x.time \cap bound \neq \emptyset$ **do**
  FORK
  if *On forked child* **then**
    remove x from *input*
    forall $y : y \in input \wedge y.time \cap bound \neq \emptyset$ **do**
      $Interval\langle\mathbb{R}^+\rangle i \leftarrow [max(x.time.lowerend, y.time.lowerend), y.time.upperend]$
      $i.open\_upperend \leftarrow y.time.open\_upperend$
      if $x.time.lowerend = y.time.lowerend$ **then**
        $i.open\_lowerend \leftarrow x.time.open\_lowerend \wedge y.time.open\_lowerend$
      else if $y.time.lowerend < x.time.lowerend$ **then** $i.open\_lowerend \leftarrow x.time.open\_lowerend$
      else $i.open\_lowerend \leftarrow y.time.open\_lowerend$
      replace y on *input* by Event(i, $y.message$)
    $s.x\text{-}message(x.message, x.time \cap bound)$
  else
    WAIT(every branch is created)
    if $\exists y \in input : y.time \subseteq bound$ **then** EXIT
    else
      forall $y : y \in input \wedge y.time \cap bound \neq \emptyset$ **do**
        replace y on *input* by Event($y.time \setminus bound$, $y.message$)

Algorithm 4: Main-loop advancing on a Input-Collision scenario

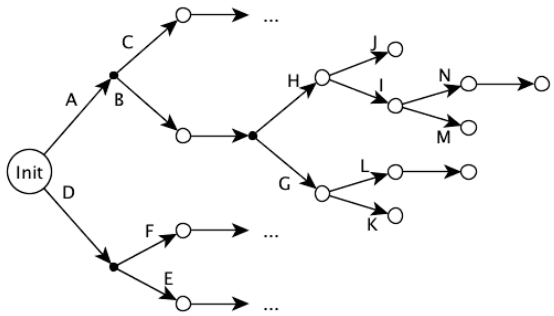

Figure 1: Trajectories-tree of Processor simulation.

Table 1: Parameters used in the case study.

Processes duration	1 second
Bound functions	Infimum and Supremum
Init time	0 seconds
Init state	Queued Jobs = $\emptyset$ Current Job Finishes in 1s
Max branches per step	4
Input queue	$\{\langle[2,3]s,[1,2]\rangle, \langle[2.5,3.5]s,[3,4]\rangle\}$
Order<Y>	$\mathbb{N}_<$
Order<S>	$S_1 < S_2 \Leftrightarrow S_1.DJQC < S_2.DJQC$ $\vee(S_1.DJQC = S_2.DJQC$ $\wedge S_1.QJ < S_2.QJ$

```
Interval⟨ℝ+⟩ bound // time slice for advancing simulation
if ∃x ∈ input : x.time ⊆ s.t_next = x.time then
    ℝ+ limit // Time limit used for conflict resolution
    limit ← x.time.upperend : (x ∈ input ∧ ∀y ∈ input, x.time.upperend ≤ y.time.upperend)
    bound ← [s.t_next.lowerend, limit]
    bound.open_lowerend ← s.t_next.open_lowerend
    bound.open_upperend ← (∃x ∈ input, bound.upperend = x.time.upperend ∧ x.time.open_upperend)
else bound ← s.t_next
forall x : x ∈ input ∧ x.time ∩ bound ≠ ∅ do
    FORK
    if On forked branch then
        remove x from input
        forall y : y ∈ input ∧ y.time ∩ bound ≠ ∅ do
            Interval⟨ℝ+⟩ i ← [max(x.time.lowerend, y.time.lowerend), y.time.upperend]
            i.open_upperend ← y.time.open_upperend
            if x.time.lowerend = y.time.lowerend then
                i.open_lowerend ← x.time.open_lowerend ∧ y.time.open_lowerend
            else if y.time.lowerend < x.time.lowerend then i.open_lowerend ← x.time.open_lowerend
            else i.open_lowerend ← y.time.open_lowerend
            replace y on input by Event(i, y.message)
        s.x-message(x.message, x.time ∩ bound)
    else
        WAIT(every branch is created)
        s.∗-message(bound)
```

Algorithm 5: Main-loop advancing on a Scheduled-Collision scenario

For producing the trajectories-tree of shown in Figure 1, the following steps are followed. First, we set the initial node using the init variables, which in this case are exact values not requiring branching. Before any job is introduced as input, the model passivates, therefore t_{next} is set to $\emptyset$.

Second, the first event in the input queue is read by the main loop. A message in the interval [1, 2] occurs between 2 to 3 seconds. This is an **Input-Collision** event, since next queued event occurs in the interval 2.5 to 3.5 seconds and no internal event is scheduled. Here, the simulation is branched using the bound 2 to 3 seconds. In the first branch, the second event is modified to occur between 3 to 3.5 seconds. In the second branch, the first event is modified to occur between 2.5 to 3 seconds. In both branches the simulation is advanced calling the *x-message* function and new nodes are added in the tree.

For the first branch (A), t_{local} is set to [2, 3] seconds, and the state is changed to <1s, [1, 2]>. Since the second component of state is an interval on Natural numbers, [1, 2], this is a finite set and below the limit of branches per step. Here, the *split_state_sets* function branches the simulation in two, having states <1s, 1> (B) and <1s, 2> (C). The t_{next} in both new branches is set to [3, 4].

For second branch (D), t_{local} is set to [2.5, 3] seconds, and the state is set to <1s, [3, 4]>. Similar to branch A case, the states are branched into <1s, 3> (E) and <1s, 4> (F). For both branches, t_{next} is set to [3.5, 4]s.

Continuing on branch B, we have again an input collision, this time between the next event (<[2.5, 3.5] s, [3,4]>) and t_{next} ([3, 4]s). This time, the bound is [2.5, 3] s. We branch consuming the event during the bound (G), and with no consumption (H). In G, t_{local} is [0-1] s, and the complete set of possible states could be described as <[0-1]s, [1:[3, 4], [3, 4]] >. Here, we have four possible queue values in the state 1:3, 1:4, 3, or 4. The first two values are for the case that the time elapsed is less than 1 and the previous jobs are not dequeued. The last two are the cases when the jobs are dequeued. However, since the first component is a proper real interval, [0, 1], the set of states is not below the branches per step limit. We work here with its

bounds and obtain the interval [<0, 1:3>, <1, 4>]. These bounds introduce infinite sets not included in the complete description, as i. e. <0.5, 1:2>. Finally t_{next} is then set to [2.5, 4].

For the second branch of B (branch H), the event in the queue is rewritten to <[3, 3.5]s, [3, 4]>. After that, the loop is iterated one more time entering in a **Scheduled-Collision** scenario. Here, the bound is set to [3, 3.5]s and two branches are created. One for consuming the event input, which may reschedule the internal transition (I), and other for consuming internal transition first (J). In I, local is set to [0-1.5], this produces again same set of infinite value states described in G, but at different occurrence time, [3, 3.5]s. The new t_{next} is then set to [3, 4.5]. In the branch J, the Internal transition is executed and a single value is returned for every case, <1s, $\emptyset$>, and t_{next} is set to infinite.

Following now with the G branch, we have no more events for input and we only have to consume the internal transitions using **No-Collision**. Here we consume one job for every queue in the set and set first component of state to 1. Then the new bounded set of states is [<1, $\emptyset$>, <1, 3>]. The new set has only two values, then we branch them in K and L. The K branch having the empty queue sets t_{next} to $\emptyset$ and the L branch sets t_{next} to [3.5, 4]. Next step in L branch consumes the last job and sets t_{next} to $\emptyset$. Similarly, we proceed on branch I. Following the same steps we complete branches C and D.

The trajectories-tree in Figure 1, shows infinite possible results that superset all reachable trajectories. We have certainty, e.g., that it is not possible for a job to stay queued after 4.5 seconds. However, we cannot state that there will be one at 4 seconds. However, it may be possible to say that using different order functions.

6 CONCLUSIONS

We showed algorithms based on the DEVS formalisms and the FF-DEVS simulation algorithms. These algorithms can be used for simulating any DEVS model for input events with uncertainty quantifications. The results obtained are summarized in a tree, named trajectories-tree. Each branch shows possible set of occurrences and states using uncertainty quantifications. The results provided are supersets of those searched, by introducing computation errors. However, we guarantee no result is missing in the tree.

We provided a case study of a Processor model with uncertain input events, and showed the kind of assertions that could be made.

For future work, we want to explore methods to reduce the introduced errors by splitting intervals in finite pieces. In addition, we plan to introduce the required language features to the modeling language to describe uncertain behavior. This will allow for the introduction, i.e., of instrumental models in closed loop systems. And, study performance in concrete computer implementations.

REFERENCES

Beraldi, R., L. Nigro, and A. Orlando. 2003. “Temporal uncertainty time warp: an implementation based on Java and ActorFoundry”. *Simulation* vol. 79 (10), pp. 581–597.

BIPM 2008. “Guide to the Expression of Uncertainty in Measurement, (1995), with Supplement 1, Evaluation of measurement data, JCGM 101: 2008”. *Organization for Standardization, Geneva, Switzerland*.

Chow, A. C. H., and B. P. Zeigler. 1994. “Parallel DEVS: A parallel, hierarchical, modular, modeling formalism”. In *Proceedings of the 26th conference on Winter simulation*, pp. 716–722. Society for Computer Simulation International.

Fujimoto, R. M. 1999. “Exploiting temporal uncertainty in parallel and distributed simulations”. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pp. 46–53. IEEE Computer Society.

Hernandez, A., and N. Giambiasi. 2005. "State Reachability for DEVS Models". In *Proc. of Argentine Symposium on Software Engineering*.

Hoover, W. G. 1999. *Time reversibility, computer simulation, and chaos*. World Scientific.

Hwang, M. H., and S. K. Cho. 2004. "Timed Behavior Analysis of Schedule Preserved DEVS". In *Proceedings of 2004 Summer Computer Simulation Conference*, pp. 26–29.

Hwang, M. H., and B. P. Zeigler. 2006. "A Modular Verification Framework Based on Finite & Deterministic DEVS". *SIMULATION SERIES* vol. 38 (1), pp. 57.

Hybinette, M., and R. Fujimoto. 1997. "Cloning: a novel method for interactive parallel simulation". In *Proceedings of the 29th conference on Winter simulation*, pp. 444–451. IEEE Computer Society.

Hybinette, M., and R. M. Fujimoto. 2002. "Scalability of parallel simulation cloning". In *Simulation Symposium, 2002. Proceedings. 35th Annual*, pp. 275–282. IEEE.

Jech, T. 2003. "Set theory. The third millennium edition". *Springer Monographs in Mathematics. Springer-Verlag*.

Joint Committee for Guides in Metrology 2012. *The international vocabulary of metrology—basic and general concepts and associated terms (VIM), 3rd edn. JCGM 200: 2012*. Joint Committee for Guides in Metrology.

Loper, M. L., and R. M. Fujimoto. 2004. "A case study in exploiting temporal uncertainty in parallel simulations". In *Parallel Processing, 2004. ICPP 2004. International Conference on*, pp. 161–168. IEEE.

Nutaro, J. 2003. "ADEVS: User manual and API documentation". *On-line document, http://www. ece. arizona. edu/~nutaro/adevs-docs/index. html*.

Peschlow, P., and P. Martini. 2007. "A discrete-event simulation tool for the analysis of simultaneous events". In *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, pp. 14. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Vicino, D. 2015. *Improved Time Representation in Discrete-Event Simulation*. Ph. D. thesis, Université Nice Sophia Antipolis; Carleton University (CA).

Vicino, D., O. Dalle, and G. Wainer. 2015. "Using finite forkable DEVS for decision-making based on time measured with uncertainty". In *Proceedings of the 8th International Conference on Simulation Tools and Techniques*, pp. 89–98. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press.

AUTHOR BIOGRAPHIES

DAMIAN VICINO received his PhD from University of Nice-Sophia Antipolis and Carleton University in 2015. He received his Computer Science degree from University of Buenos Aires. He develops the Cadmium Simulator. His email address is dvicino@sce.carleton.ca.

GABRIEL A. WAINER is an Associate Professor in the School of Operations is Professor at the Department of Systems and Computer Engineering, Carleton University. He is a Fellow of SCS. His email address is gwainer@sce.carleton.ca

OLIVIER DALLE is assistant professor in the C.S. dept. of Faculty of Sciences at University of Nice-Sophia Antipolis (UNS). He received is BSc from U. of Bordeaux 1 and his M.Sc. and Ph.D. from UNS. His email address is olivier.dalle@unice.fr.