# Uncertainty on Discrete-Event System Simulation



DAMIAN VICINO, Université Nice Sophia Antipolis—I3S UMR CNRS 7271
GABRIEL A. WAINER, Department of System and Computer Engineering, Carleton University
OLIVIER DALLE, Université Nice Sophia Antipolis—I3S UMR CNRS 7271

Uncertainty Propagation methods are well-established when used in modeling and simulation formalisms like differential equations. Nevertheless, until now there are no methods for Discrete-Dynamic Systems. Uncertainty-Aware Discrete-Event System Specification (UA-DEVS) is a formalism for modeling Discrete-Event Dynamic Systems that include uncertainty quantification in messages, states, and event times. UA-DEVS models provide a theoretical framework to describe the models' uncertainty and their properties. As UA-DEVS models can include continuous variables and non-computable functions, their simulation could be non-computable. For this reason, we also introduce Interval-Approximated Discrete-Event System Specification (IA-DEVS), a formalism that approximates UA-DEVS models using a set of order and bounding functions to obtain a computable model. The computable model approximation produces a tree of all trajectories that can be traversed from the original model and some erroneous ones introduced by the approximation process. We also introduce abstract simulation algorithms for IA-DEVS, present a case study of UA-DEVS, its IA-DEVS approximation and, its simulation results using the algorithms defined.

CCS Concepts: • **Computing methodologies** → **Simulation theory**; • **Mathematics of computing** → Continuous mathematics;

Additional Key Words and Phrases: Uncertainty, discrete-event simulation, DEVS

**ACM Reference format:**
Damian Vicino, Gabriel A. Wainer, and Olivier Dalle. 2021. Uncertainty on Discrete-Event System Simulation. *ACM Trans. Model. Comput. Simul.* 32, 1, Article 2 (September 2021), 27 pages.
https://doi.org/10.1145/3466169

## 1 INTRODUCTION

When we model a real system, we produce an abstract representation of what we observe in reality. This abstraction process implies a loss of information [18] that could be noted as imprecision on the model specification. Such partial knowledge can be a product of technical limitations, tools to observe the phenomenon being studied, representation limitation leading to approximations, lack of access to the real system, modeling a system that is yet not created. The partially defined data is usually called "data with uncertainty".

Damian Vicino worked for the CNRS at the time the research was conducted.
Authors' addresses: D. Vicino (corresponding author) and O. Dalle, Université Nice Sophia Antipolis—I3S UMR CNRS 7271, 2000 Route des Lucioles, Sophia Antipolis 06900, France; emails: vicinod@amazon.com, olivier.dalle@unice.fr; G. A. Wainer, Department of System and Computer Engineering, Carleton University, 1125 Colonel By Drive, Ottawa, Ontario K1S 5B6, Canada; email: gwainer@sce.carleton.ca.

1049-3301/2021/09-ART2 $15.00
https://doi.org/10.1145/3466169

In many areas, data uncertainty can be characterized mathematically, and the uncertainty of the virtual experiments can be computed from the components of the virtual experiment. This computation is known as "Uncertainty Propagation". For example, data collected for continuous magnitudes are known to be inexact [1], however, they are used in science and engineering every day.

The **International Bureau of Weights and Measures** (**BIPM**) proposes to capture the imprecision as a set including all possible values attributed to a measurand. We call this set of values the "uncertainty quantification" of the "measurement result" [2]. The concepts of uncertainty and uncertainty propagation can be used also outside of the Metrology domain. We can easily use them to characterize approximated discrete data sets or to describe theoretical conjectures.

There are well-established methods for studying Uncertainty Propagation in continuous models, and uncertainty in discrete-time models can be studied by exploring each set of values in the uncertainty quantification. Nevertheless, there are no methods for the propagation of uncertainty in **Discrete-Event Dynamic Systems** (**DEDS**). This research presents a family of formalisms based on **Discrete-Event System Specification** (**DEVS**), a universal formalism for DEDS [20].

As there are no formalisms for specifying uncertainty in DEDS models, there are no DEDS simulators with uncertainty propagation mechanisms. Thus, in [17] we proposed a simulation method to propagate the uncertainty of a set of uncertain exogenous events. Here, we extend those concepts and introduce **Uncertainty-Aware DEVS** (**UA-DEVS**), a formalism that can be used to model DEDS including uncertainty quantifications. UA-DEVS allows the modeler to describe uncertainty for the variables used to represent time, states, and messages by using sets that represent uncertain values.

As we will discuss in Section 3, UA-DEVS models have no restrictions on cardinality, and they can contain non-numerable elements, allowing one to think about the model more intuitively. As well, this allows studying the model's properties without compromising accuracy at the modeling level. Although this can be done to improve computational costs, it normally carries a cost: the models built are complex to understand, as they need to include computational artifacts related to the simulation process, which is not needed with UA-DEVS.

From here on, we will say that a formalism is *simulable* when we can find a method that, when conducting virtual experiments, generates a set of trajectories (i.e., we can formally prove that we can generate the trajectories). Similarly, we will say that a formalism is *simulatable* when the method used for generating the trajectories can be automated to be executed in computers (i.e., without human intervention).

Since UA-DEVS can generate an infinite unsorted and non-continuous set of states in a single simulation step, we can say that the formalism is simulable, but not simulatable. This means that UA-DEVS can be used for defining models formally that can generate trajectories that can be defined formally, but there is no mechanism to execute those formal models in a computer using an algorithm to generate such trajectories. We thus also need to define a simulatable formalism, that we named **Interval-Approximated DEVS** (**IA-DEVS**), which approximates the formal UA-DEVS models but allows the models to be simulated on computers. Converting a UA-DEVS model to an IA-DEVS model only requires adding a few functions to the model definition. These functions are based on intervals in place of arbitrary sets for the simulation variables. We need to define an abstract simulation algorithm for IA-DEVS that guarantee that the approximation errors do not exclude any trajectory from the UA-DEVS model been approximated. However, UA-DEVS allows us to model the system and describe its uncertainty properties as accurately as possible. And, IA-DEVS can be used to construct multiple approximations based on computational constraints (time, memory, etc.). This separation of concerns allows the domain expert to define the model once, and then we can iterate its simulatable approximations without losing sight of the expected model behavior.

The rest of the article is organized as follows: In Section 2, we present background and related work relevant for our research; in Section 3, we present the formal specification of UA-DEVS models; in Section 4, we present the formal specification of IA-DEVS models; in Section 5, we explain how to obtain an approximated IA-DEVS model from a UA-DEVS; in Section 6, we present the algorithms to simulate IA-DEVS models; in Section 7, we present a case study; and in Section 8, we discuss conclusions and future work.

## 2 BACKGROUND

### 2.1 Uncertainty Theory

Uncertainty is a concept used to describe incomplete knowledge, partial unknowns, and vague definitions. For example, we introduce uncertainty in colloquial language when we use phrases like "approximately one meter", or "around 39 Celsius". In technical environments, we introduce uncertainty to measure magnitudes [1, 2].

A common source of uncertainty for **modeling and Simulation** (**M&S**) is that we use real systems measurements. In Metrology, the science of measurements and its applications, the True Value of a measure (i.e., the single value obtained with perfect accuracy) cannot be known. The measurement results of a magnitude must include quantification of its uncertainty as described in *Guide to the Expression of Uncertainty in Measurement* [1] published by the Bureau International des Poids and Measures (BIPM, an international agency for standardizing international Units). According to BIPM [1], a measurement result can be described as a set that includes all the reasonable values that could be assigned to the measurand (the target of the measure). Uncertainty concepts are also used in other areas that can benefit from Simulation, as for designing innovative technologies [19], and for describing tolerance descriptions in industrial Quality Assurance [13], and so on.

*Uncertainty Theory* [9] presents an axiomatic definition of Uncertainty whose core concepts are the Uncertainty Measure, Uncertain Variables, and Uncertainty Distributions. The Uncertainty Measure is based on the degree of belief over an uncertain event. Uncertainty Variables represent imprecise quantities. Finally, Uncertainty Distributions model how a belief distribution is interpreted. It is a common misconception that uncertainty can be modeled using fuzzy logic or stochastic models. However, it was showed uncertainty does not behave like any of them [9].

### 2.2 Discrete-Event System Specification

DEVS [20] is a formalism for modeling Discrete-Event Systems that provides a formal specification for hierarchical modeling and an abstract method to simulate legitimate models. The simulator is independent of the models, and it has been proven to simulate the DEVS models correctly. In DEVS, a model is built as a hierarchy of two kinds of components: atomic models and coupled models.

In DEVS, the atomic models are defined as a tuple $A = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where:

$X$ is the set of inputs; $Y$ is the set of outputs; $S$ is the set of sequential states;

$Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ is the total state set (where $e$ is the time elapsed since last transition);

$\delta_{int} : S \to S$ is the internal transition function; $\delta_{ext} : Q \times X \to S$ is the external transition function;

$\lambda : S \to Y$ is the output function; $ta : S \to \mathbb{R}^+$ is the time advance function.

It is a common misconception to believe that $S$, the sequential state set of DEVS models, must be discrete. The set $S$ can be defined as an arbitrary set of values that is not required to be discrete [20, Section 4.1]. In [20] the authors introduce various models using $S$ defined over an uncountable set.

DEVS is defined as a DEDS because, even for models with uncountable possible values in $S$, the trajectories generated only visit a discrete set of events in that set.

As we need to use these theoretical concepts, we now discuss a classic example of a DEVS model that will be used to explain uncertainty ideas. This model, called *Processor* in the DEVS literature [20], uses an $S$ set defined over an uncountable set. It represents a server that receives jobs to be executed. Each job takes a fixed time to run. In case the Processor is busy and a new job is received, the new job is queued until the current one is completed. Each job completion generates an output.

Therefore, the model $Processor_{process\_duration} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$, can be defined as

$X = Y = \mathbb{N}$;

$S = \langle TOCJ, QJ \rangle$ where: $TOCJ$ (Time Of Current Job) represents the time ($\mathbb{R}^+$) until the current job started processing; $QJ$ (Queue of Jobs) is a queue of processes, each of which is a value in $\mathbb{N}$;

$\delta_{int}(s) = \langle process\_duration, s.QJ.dequeue\_first \rangle$;

$\delta_{ext}(s, e, x) =$ if $(s.QJ.size = 0)$ then $\langle 0, S.QJ.queue(x) \rangle$ else $\langle s.TOCJ + e, s.QJ.queue(x) \rangle$;

$\lambda(s) = s.QJ.first$; $ta(s) =$ if $(s.QJ.size = 0)$ then $\infty$ else $process\_duration - s.TOCJ$.

Here, $process\_duration$ defines a fixed amount of time that each job needs to be processed. The inputs ($X$) are Natural numbers, each of which identifies a process in the queue. The outputs ($Y$) are Natural numbers identifying the processes that are completed. The sequential state set ($S$) is represented as the Cartesian product of Reals by the power set of jobs. For simplicity, we call $TOCJ$ the first component of elements in $S$, a Real number that represents how long the process has been active. We name the second component $QJ$, a list of jobs to be processed. $TOCJ$ is needed because time is local to the model, and the elapsed time of the model ($e$) is reset after each event is processed. In addition, as the first element of the pair is a Real number, the sequential state set is not discrete. This is not a problem: DEVS only needs trajectories to visit discrete states to be legitimate.

The internal transition function ($\delta_{int}$), which is executed at the end of processing a job, dequeues the next job from the head of $QJ$ and resets the processed time counter. The simulation algorithm will never call this function if the model is passive (which is represented using a time advance of infinity). The model is passive when there are no jobs in the queue. When the external transition function ($\delta_{ext}$) receives an external event (i.e., a new process), it queues it for processing. If the queue was empty at that time, the process starts immediately, and the counter ($TOCJ$) is reset; otherwise, the timer is set to the time representing what was processed before, plus the elapsed time since the last event. The output function ($\lambda$) executes when a job is completely processed and transmits the identifier of the first job in the queue. The time advance function ($ta$) computes the remaining time until the end of the current job (if there are any jobs left in the queue). As discussed earlier, when there are no jobs in the queue, $ta$ returns $\infty^+$, indicating that the model is passive.

We also use a Generator model [20]. In its simplest form, it receives no input, and it outputs the same value periodically, like a clock in an electronic component.

In DEVS, the model $Generator_{period} = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$, can be defined as

$X = \mathbb{R}$; $\quad Y = \{1\}$;

$S = \{r | r \in \mathbb{R}^+ \wedge r < period\}$ where $period \in \mathbb{R}^+$ is the period between generated values;

$\delta_{int}(s) = 0$; $\quad \delta_{ext}(s, e, x) = s + e$; $\quad \lambda(s) = 1$; $\quad ta(s) = period - s$.

Any real number is a valid input: None are expected, and if any is received, it is ignored. The output is a single value (in this example, number 1), repeated periodically. To keep the period aligned to the absolute timeline, we need to keep track of the possible interruptions, as we did with the *Processor*. Then, we accumulate the time advance in $S$. The function $\delta_{int}$ resets the counter after each output. The function $\delta_{ext}$ keeps track of time advances when an external event is received (and ignored, as explained). Function $\lambda$ always outputs the value 1. The time advance function $ta$ uses the value stored in $S$ and the specified period to schedule the next output.

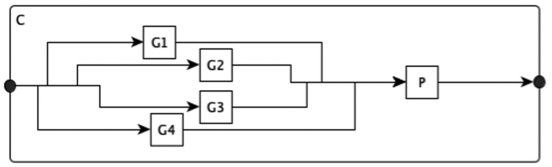


Fig. 1. Coupled model C.

Coupled models are used to compose models hierarchically in DEVS. We next present how these models can be combined into a larger model to produce an emergent behavior of their collaborations. A coupled model is defined by a set of DEVS sub-models and a description of how these sub-models interact with each other [20]. Formally, a coupled Model is defined as a tuple $C = \langle X, Y, D, M, I, Z, SELECT \rangle$ where:

$X$ is the set of input values; $Y$ is the set of output values;

$D$ is an index for the DEVS component models of the coupled model (which can be atomic or coupled);

$M = \{M_d | d \in D\}$ is a tuple of DEVS models as defined previously;

For each $d \in (D \cup \{self\}), I_d$ is the influencer set of $d, I_d \subseteq (D \cup \{self\}), d \notin I_d$

For each $i \in I_d$, $Z_{i,d}$ is the i to d output translation function with:

$$
\begin{array}{ll}
Z_{i,d} : X \rightarrow M_d.X & \texttt{if } i = self\texttt{, known as the external input coupling,} \\
Z_{i,d} : M_i.Y \rightarrow Y & \texttt{if } d = self\texttt{, known as the external output coupling,} \\
Z_{i,d} : M_i.Y \rightarrow M_d.X & \texttt{otherwise, known as the internal coupling;}
\end{array}
$$

$SELECT : (\mathcal{P}(D) - \emptyset) \rightarrow D$ is the tie-breaker function that sets priority in case of simultaneous events.

Here, the model uses $X$ and $Y$ input/output sets to build a modular interface to couple models with each other. The coupled model uses an index of its components, named $D$. The $M$ set identifies each of the coupled model subcomponents, and associates them to the index set $D$. For each model in the set, we define the influencer set $I$, which defines how models in the coupled models are connected. Using the $I$ set, we build the $Z$ translation function, which defines how to connect the outputs in one model and convert them into inputs into another. The $SELECT$ function acts as a tie-breaker in the case that two or more models are activated simultaneously: it decides which of the sub-models execute first. Let us consider this definition in the example shown in Figure 1. Here, we connect four Generators sending jobs to a single Processor. Each Generator has a period of 1,000 ms, and the Processor can process one job every 250 ms. The formal specification for this coupled model is defined as $C = \langle X, Y, D, M, I, Z, SELECT \rangle$ where:

$X = Y = \mathbb{N}; \quad D = \{G1, G2, G3, G4, P\};$

$I_{G1} = C, \quad I_{G2} = C, \quad I_{G3} = C, \quad I_{G4} = C, \quad I_P = \{G1, G2, G3, G4\}, \quad I_C = \{P\}$

$M_{G1}$, $M_{G2}$, $M_{G3}$, $M_{G4}$ and $M_P$ are atomic Models;

$Z_{G1,P}(v) = \{1\}$, $Z_{G2,P}(v) = \{2\}$, $Z_{G3,P}(v) = \{3\}$, $Z_{G4,P}(v) = \{4\}$, $Z_{P,self}(v) = v$;

$G_{G1,P}(v) = v$, $C_{G2,P}(v) = v$, $C_{G3,P}(v) = v$, $C_{G4,P}(v) = v$;

$SELECT(d) = min(d)$ where $G1 < G2 < G3 < G4 < P$.

Here, our coupled model ($C$) receives any natural number as input. The output ($Y$) is the job identifier sent by the Processor. We define the same set of values used in the Processor. The identifiers set ($D$) contains $C$, $G1$, $G2$, $G3$, $G4$, and $P$. The influencer set $P$ ($I_P$) which contain $G1$, $G2$, $G3$, $G4$. The influencer set of $C$ ($I_C$) contains $P$. The influencer set of each Generator which contains $C$. Identifiers ($D$), and influencers ($I$) can be diagramed as shown in Figure 1 for a quicker-to-read representation. The translation function ($Z$) transforms each output from a different Generator into

a number identifying the source of the message, and it uses the identity function for values being sent from the Processor to the coupled model itself and from the coupled model to the Generators. Finally, the *SELECT* function gives priority to the execution of simultaneous events, so that any generated job is queued before any other is dequeued.

If we study the execution trajectories of this coupled model, we obtain the following results. Each second, four jobs are queued (in *P*) in order: 1, 2, 3, and 4. After 250 ms, the first job (1) is completed. After another 250 ms, the second job (2) is completed; then the third (3), and the fourth (4). Then, the internal transition (which will send its completion message) must be executed simultaneously with the queuing of four more jobs. *SELECT* gives priority to the queuing of new jobs; therefore, there are five jobs in the queue. After all the jobs are queued, the message is sent, and the fourth job (4) is dequeued. This repeats every 1,000 ms.

A property of the model is that it never has more than five jobs in the queue at the same time. An extension to the previous definitions, known as DEVS with ports, defines DEVS models with multiple ports for inputs and outputs. DEVS with ports is proven equivalent to DEVS [20]. In this research, we use DEVS (without ports) as we focus on the definition of uncertainty; using "classic" DEVS allows us to keep the algorithms simpler to describe and understand. However, as the specifications are equivalent, we recommend developing computer implementations supporting ports. The adaptation only needs a few syntactic replacements, and makes tools more intuitive to be used by modelers.

One of the advantages of DEVS is that all the models can be simulated by the same simulation algorithm, which follows an abstract specification defined in [20]. These abstract algorithms proposed by Zeigler are based on a definition consisting of three simulation entities:

- *Simulators*, which provide the mechanisms to simulate atomic models.
- *Coordinators*, which provides the mechanisms to simulate coupled models by coordinating the **Simulators** or **Coordinators** of their sub-models.
- *Root Coordinator*, which provides a top-level component to advance the simulation by advancing the time in the top coupled model.

Both **Simulators** and **Coordinators** use their own data structures to track the advance of time and the current state of the component they control. They use four functions to advance the simulation when needed. The functions are named after the messages they receive:

- *init-function* initializes all the data structures at the start of the simulation.
- $^*$*-function* advances the simulation because of an internal state transition.
- *x-function* represents an external event received, and it triggers an external transition.
- *y-function* processes the return messages from lower-level **Simulators** or **Coordinators** and passes them to the corresponding **Coordinators** and **Simulators**.

### 2.3 DEVS Extensions

Several extensions have been proposed to DEVS, for example, **Parallel DEVS** (**PDEVS**) [3] targets solving problems on serial computation caused by the *SELECT* function; and Finite & Deterministic DEVS [6] can be used to study all reachable states based on graph theory.

Our early studies of Discrete-Event Simulation with uncertainty assumed models perfectly accurate. First, we presented an extension to DEVS with new simulation algorithms [15] that allowed exogenous events with uncertainty. The events propagate uncertainty through the simulation using a branching mechanism. We then characterized a subset of DEVS models that were simulatable for every input. We called the subset Finite-Forkable DEVS. Then, we introduced new algorithms with approximation errors, simulatable for DEVS models with uncertain input events [17], using

order functions over the set of sequential states. Using these functions, the simulator can combine infinite reachable states into a finite number of intervals. The method can lead to unreachable states, but never misses a reachable one (similarly to propagation methods used in Continuous Systems M&S).

This research is significantly different. We propose the first system specification with uncertainty quantification for all model properties (previously we only allowed uncertainty in the inputs). In our previous research, we found out that we need to sort the functions using interval approximations. We defined a system specification to include these ordering and bounding functions, and a method to keep them separated from the model. In addition, we introduce a new intermediate specification (called IA-DEVS) that allows studying multiple approximation strategies for each model, to better understand optimization opportunities. For example, we could study the impact of using floating-point against fixed-point of a particular model without changing neither the simulator nor the model. In addition, we here introduce coordination algorithms not presented earlier.

Fujimoto et al. discussed Uncertainty and Discrete-Event Simulation [4, 10]. They take models defined as perfectly accurate, and they introduce artificial uncertainty intervals to the events. They use the concepts of Approximate Time and Approximate Time Event Ordering, where a common ordering used is called "Approximate Time Causal". In this ordering, two events are considered concurrent if the interval representing them has a non-empty intersection. For concurrent events, a second ordering is used, the causality relation *before than* [8]. This approach state that there is a relation between the simulation accuracy and uncertainty, but they do not provide tools to limit the errors, quantify the uncertainty of the results, or express qualitative information about the validity of the results. Other authors proposed to introduce uncertainty on the spatial properties of the model for obtaining speed-ups [5, 11]. Although these methods discuss uncertainty, the problem they are trying to solve is different than the one presented in this research. The authors introduce a method that starts from a model that is described accurately and then they introduce artificial uncertainty to generate a similar model that runs faster. The new model is not guaranteed to generate the same trajectories as the one been approximated, neither have its error bounded in any way. However, in some scenarios where the time to make a decision is tight, and one knows that the models will provide similar results, this method can be useful.

Saadawi and Wainer explored in [12] replacing time data type in DEVS models by intervals and presented the RTA-DEVS extension formalism. The major limitation of replacing the time variables with intervals that represent an uncertain value is that it would be not possible to identify the next event. The algorithms in IA-DEVS focus on how to explore all the possible outcomes of different orders between those events whose uncertain interval overlap. In addition, UA-DEVS and IA-DEVS provide methods to specify uncertainty in the state, input, and output variables in addition to the time variable.

### 2.4 Formal Notes

In this section, we discuss a few issues related to the formal definition of DEVS models and the simulation algorithms, which have not been formally defined in the past.

— In [20], the explanation of DEVS simulation algorithms focuses on the $*$, $x$, and $y$ messages used to advance each step of a simulation. There is little detail given about how the initialization *init* messages are defined. In our case, *init* is a tree structure describing the hierarchy of the Model simulated. In the leaves of the tree, we keep a value from the total state set ($Q$) of each atomic model. Traversing the tree, each atomic model is initialized with its value from $Q$. We also need to have a clear definition of the use of *init* by the DEVS' simulation

algorithms, whose purpose is defining an initial state for a simulation experiment. This initial state could be defined artificially, but also could be used for resuming a paused simulation, if the state at the time it was paused was captured.

— In the definition of DEVS, the Set Theory used for defining states and messages is never mentioned explicitly, but there are many Set Theories. In the context of this article, we follow **Zermelo–Fraenkel with the axiom of Choice** (**ZFC**) Set Theory [7]. ZFC theory includes the Well-Ordering Theorem, which states that there is a relation of total order for every set (i.e., every set can be well-ordered). It includes another important property: all non-empty subsets have an element that is the smallest in that order. The use of ZFC for defining our models' guarantees that we can find order functions for defining approximated models, as described in Section 4.

— Some authors refer to intervals as segments of $\mathbb{R}$. In the context of this research, we use the word interval as a broader definition applicable to subsets of any ordered set that could be described by an upper bound, a lower bound, and their inclusion or not.

— We adopt the following notation for common set operations. $A^B$ refers to all possible functions from the Set $B$ to the Set $A$. For example, if $A = \{1, 2\}$ and $B = \{2, 3\}$, then $A^B$ has four functions in $B \to A$: $f(x) = 1$, $f(x) = 2$, $f(x) = x - 1$, and $f(x) = 4 - x$. $\mathcal{P}(D)$ refers to the Power Set of $D$ including $\emptyset$. For example, if $D = \{1, 2\}$, then $\mathcal{P}(D)$ refers to $\{\{\}, \{1\}, \{2\}, \{1, 2\}\}$.

## 3 UNCERTAINTY-AWARE DEVS (UA-DEVS)

The UA-DEVS is a DEVS formal description that considers Uncertainty properties. Domain experts can capture the data model without worrying about the approximations needed for the simulation of experiments, or the implementation details for generating the trajectories of the simulated dynamic behavior.

UA-DEVS are also composed of behavioral models (Atomic), and Structural models (Coupled) as in DEVS. The models' input, output, state, and time variables include uncertainty. Each combination of the individual elements from the specification description specifies a DEVS model. UA-DEVS is a specification of a family of DEVS models where all models are related by describing all behaviors of systems with uncertainty.

### 3.1 Atomic Model

In UA-DEVS, an atomic model is defined as a tuple $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where $X$ , $Y$, $S$, and $Q$ are defined as those in DEVS discussed in Section 2.2. Based on these, we now define

$X_p = \mathcal{P}(X) - \emptyset$ is the set of uncertainty-aware inputs;
$Y_p = \mathcal{P}(Y)$ is the set of uncertainty-aware outputs;
$S_p = \mathcal{P}(S) - \emptyset$ is the set of uncertainty-aware sequential states;
$Q_p : \mathcal{P}(Q) - \emptyset$ is the set of uncertainty-aware total states;
$\delta_{int} : S_p \to S_p$ is the internal state transition function;
$\delta_{ext} : Q_p \times X_p \to S_p$ is the external state transition function;
$\lambda : S_p \to Y_p$ is the output function; $ta : S_p \to \mathcal{P}(\mathbb{R}^+)$ is the time advance function;
As discussed in the previous section, $\mathcal{P}$ is the notation used for the power set operator.

The definition of UA-DEVS atomic models extends classic DEVS. $X$, $Y$, $S$, and $Q$ have the exact same meaning as in DEVS, and for each of them, a new power set is defined (represented with the $p$ subscript). These new sets capture all the values in an uncertainty context. For example, $Y$ defines values of output, while $Y_p$ defines all sets of output combinations that could result from propagating uncertainty during the process. In the case of $X$, $S$, and $Q$, we explicitly prohibit the use of the empty set, however, we allow it in $Y$ to represent models with no outputs. The restriction is

based on semantics: an empty input conflicts with the meaning "not having an input", so, it should not trigger the external transition. In addition, a model with an empty state set would mean that no state was reached by the simulator, and this is an impossible situation. However, an output with no events during an internal transition is convenient for defining the complex evolution of a system.

The functions defined for UA-DEVS have domains and co-domains in $X_p$, $Y_p$, $S_p$, and $Q_p$. The functions must be defined from *sets of sets* to *set of sets*, allowing the representation of uncertainty in the function parameters and the results. For example, if $S = \{0, 1\}$ then $S_p = \{\{0\}, \{1\}, \{0, 1\}\}$, and an internal transition function could be defined as $\delta_{int}(s) = s \cup \{1\}$. Here, if the previous sequential state was 0, evaluating the internal transition function adds uncertainty by returning the set of sequential states $\{0, 1\}$. Likewise, if the sequential state was $\{1\}$ or $\{0, 1\}$ the set is not altered and $\delta_{int}$ returns the same value received by parameter.

The time advance function is defined from $S_p$ to $\mathcal{P}(\mathbb{R}^+)$ to account for the case where different sequential states schedule their next internal event at different times. DEVS models normally use a time advance of infinite to define a passive model (i.e., one without a scheduled future internal event). The time advance function, now with its domain in $S_p$ and co-domain in $\mathcal{P}(\mathbb{R}^+)$, cannot use infinite as a value to represent passive models: infinite is not an element of the co-domain. We have two alternatives; we could specify it as a set with infinite as its only element or use the empty set to represent passive models. We prefer the latter to simplify the notation in the algorithms. For example, if we have a passive system with a future events list noted as $\emptyset$, and another with an event scheduled at $1{,}000ms$ in the future with its future events list noted as $\{1{,}000ms\}$, the union of their future events list will produce a single event at $1{,}000ms$ in the future, as expected, since $\emptyset \cup \{1{,}000ms\} = \{1{,}000ms\}$.

In DEVS, the simulation stops if all models are passive. As mentioned earlier, this is represented using a time advance function that returns the value infinity. Similarly, in UA-DEVS a simulation reaches a final state and stops if all models are passive, which in UA-DEVS are represented with all models producing an empty set as a result of evaluating their time advance function.

Recall that events in Discrete-Event Systems always produce an instantaneous change. In UA-DEVS, events are still instantaneous, but we evaluate all the possible instants they may occur.

The following is the specification of a Generator model that sends an output approximately every $1{,}000ms$. In this example, we set the uncertainty for the output to be between $997ms$ and $1{,}005ms$. The output generated is one of two values, 1 or 2. Any input message (any real number) is discarded. Formally, $Generator_{UA}$ is an atomic model described by the tuple $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta\rangle$ where:

$X = \mathbb{R}$; $Y = \{1, 2\}$; $S = \{r | r \in \mathbb{R}^+ \wedge r \leq 1005ms\}$;
$X_p = \mathcal{P}(\mathbb{R}) - \emptyset$; $Y_p = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$; $S_p = \mathcal{P}(S) - \emptyset$; $\delta_{int}(s) = \{0\}$;
$\delta_{ext}(q, x) = \{(s + e) | s \in q.state \wedge e \in q.time\}$; $\lambda(s) = \{1, 2\}$;
$ta(s) = \{(l - e) | l \in [997, 1005]ms \wedge e \in S\}$.

This model is an extension with uncertainty to the DEVS Generator defined in Section 2.2. The uncertainty is presented in the output of two possible values (1 and 2) and the period between outputs, which can take values between $997ms$ and $1{,}005ms$.

We define $S$ as the set of real values between 0 and $1{,}005ms$, which are used to keep track of the time elapsed since the last internal transition when an external event is processed. We allow the model to receive any set of real values as input message ($X_p$); however, we ignore those inputs. The definition of $Y$ comes directly from the description of the model: we want to output a value of 1 or 2. The $\delta_{int}$ function reset the elapsed time counter in the sequential state variable to $\{0\}$, since its execution indicates an output is being produced. The $\delta_{ext}$ function ignores any input, to do so

it updates the elapsed time counter in the sequential state adding the value of $e$ at the time it is processed. The $\lambda$ function always outputs both $Y$ values as needed by the description of the model.

Without any inputs, this model outputs $\{1, 2\}$ every $[977, 1{,}005]ms$. When there are inputs, we adjust the sequential state to hold the elapsed time since the last output, and this lets $ta$ compute the difference between the next scheduled internal event and the time elapsed since the last output. $\delta_{int}$ sets the same state in every iteration when there are no external stimuli. However, when an external event is received, $\delta_{ext}$ executes, and new sub-sets of $S$ can be assigned to the sequential state.

This model captures all specifications of classic DEVS models for the constraints of the uncertainty intervals provided. When we simulate this model, we generate a set of trajectories that includes all trajectories of every DEVS Generator model whose output is 1 or 2, and their time advance period is between $997ms$ and $1{,}005ms$. In addition, other trajectories are included, for example, variants of generators with alternating time advance periods inside the range of values considered.

To show that the behavior of a DEVS Generator with a $1{,}000ms$ period is included in the set of trajectories of this model, we just need a syntactic replacement. This is because $1{,}000ms$ is included in $[997, 1{,}005]ms$, so it is simulated as part of all the values in the interval. Similarly, the simulation trajectories also include all those of the DEVS Generator with a period between $997ms$ and $1{,}005ms$. Furthermore, the trajectories from all models with different periods between outputs but with periods between $997ms$ and $1005\,ms$ in all iterations are also included. The last includes asymptotic incremental models, stochastic models, and fuzzy models or chaotic models, to mention a few options.

Similarly, we can define a Processor model as the one presented in Section 2.2 but with Uncertainty. In this example, our Processor propagates the values being introduced by input events with uncertainty. The processing time is $250ms$.

Formally, $Processor_{UA}$ is an atomic model described by the tuple $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta\rangle$ where:

$X = Y = \mathbb{N}$; $\quad X_p = \mathcal{P}(\mathbb{N}) - \emptyset$; $\quad Y_p = \mathcal{P}(\mathbb{N})$

$S = \langle TOCJ, QJ\rangle$ where $TOCJ$ represents the time ($\mathbb{R}^+$) since the current job started processing, and $QJ$ is a queue of processes identified by natural numbers;

$S_p = \mathcal{P}(S) - \emptyset \quad \delta_{int}(s) = \{\langle 0, dequeue(k.QJ)\rangle | k \in s\}$;

$\delta_{ext}(q, x) = \{\langle 0, queue(k.state.QJ, x)\rangle | k \in q \wedge k.state.QJ.size = 0\}$

$\cup \{\langle k.state.TOCJ + k.elapsed, queue(k.state.QJ, x)\rangle | k \in q \wedge k.state.QJ \neq \emptyset\}$;

$\lambda(s) = \{k.QJ.first | k \in s\}$; $\quad ta(s) = \{0.25 - k.TOCJ | k \in s \wedge k.QJ \neq \emptyset\}$.

Here, sets $X$, $Y$, and $S$ are specified in the same way as in the Processor model presented in Section 2.2, but the functions have different signatures. $\delta_{int}$ uses a set of pairs $s \in S$, each containing a **Time of Current Job** (**TOCJ**) and a **Queue of Jobs** (**QJ**). In every case, TOCJ is reset to $0ms$, and the first job is dequeued from QJ. $\delta_{ext}$ receives a sub-set of input events ($X$), and an element of the set of current total states $Q_p$. The total state is the product of the elapsed time since the previous event ($e$), and the set of possible sequential states (an item of $S_p$). The $\delta_{ext}$ uses those values to produce a set of new possible sequential states $S_p$. The new set of sequential states contains the same queues in QJ components as before with the new jobs received queued to each of them, and the TOCJ incremented the elapsed time ($e$). Alternatively, if there are no elements in the set of sequential states, a new element is added with the job in QJ and $0ms$ in TOCJ. We do not have to consider the case of $S_p = \emptyset$ because it is not a valid $S_p$ value by definition in UA-DEVS atomic models. The set of all outputs is obtained by reading the first element in the queue of each parin the $S_p$ set of states. Finally, time advance is the set considering every possible TOCJ component in the received set of all possible sequential states, or in case all the queues are empty, the $\emptyset$ indicating that the model is passive.

### 3.2 Coupled Model

The coupled model is a hierarchical specification of how other models interact. This does not need modifications from the original DEVS model. Even when messages passed will use $X_p$ and $Y_p$ values, they can be calculated from the original $X$ and $Y$, as done in the atomic models.

We formally define a UA-DEVS coupled model as the tuple $C = \langle X, Y, D, M, I, Z, SELECT \rangle$

If we want to build an Uncertainty-Aware version of the model in Section 2.2 and shown in Figure 1, we specify $C = \langle X, Y, D, M, I, Z, SELECT \rangle$ where:

$X = Y = \mathbb{N}$; $D = \{G1, G2, G3, G4, P\}$;

$I_{G1} = C, I_{G2} = C, I_{G3} = C, I_{G4} = C, I_P = \{G1, G2, G3, G4\}, I_{self} = \{P\}$;

$G1$, $G2$, $G3$, $G4$ are $Generator_{UA}$ models, and $P$ is a $Processor_{UA}$ model, all of them defined as in this section;

$Z_{self,G1}(v) = v, Z_{self,G2}(v) = v, Z_{self,G3}(v) = v, Z_{self,G4}(v) = v, Z_{P,self}(v) = v,$

$Z_{G1,P}(v) = \{1\}, Z_{G2,P}(v) = \{2\}, Z_{G3,P}(v) = \{3\}, Z_{G4,P}(v) = \{4\}$;

$SELECT(d) = min(d)$ where $G1 < G2 < G3 < G4 < P$.

This specification is almost identical to the one provided in Section 2.2, but we use the UA-DEVS versions of the sub-models. The introduction of uncertainty invalidates the property of the DEVS model in Section 2.2, where the maximum number of jobs in the queue at any time was five. Here, we can have longer queues. For example, evaluating all generators always using the minimum period in their uncertainty interval ($997ms$), the global maximum number of jobs in the queue at any time is unbounded. However, it is still possible to find the local maximum for a given period.

### 3.3 Using DEVS Models in UA-DEVS

In many cases, it is practical to transform a DEVS model into a UA-DEVS. This is key to reuse models previously developed, allowing them to interact with new ones with uncertainty. If the model being transformed is coupled, we can copy its specification as is, but replacing its sub-models with UA-DEVS. If the model being transformed is atomic, we can copy the $X$, $Y$, and $S$ set definitions without modifications. The functions need to be replaced by higher-order versions, where the result of the new function is equivalent to apply the function to every value in the input sets. For example, $\delta_{int}(s) = s + 1$ can be replaced by $\delta_{int}(s) = \{k + 1 | k \in s\}$.

### 3.4 Summary

We presented UA-DEVS, a specification language to describe Discrete-Event Systems for events with uncertainty based in DEVS. Models specified with UA-DEVS can be interpreted as sets of DEVS models for the analysis of their behavior and properties. However, any model producing an infinite set would need an infinite combination of parameters to evaluate in each simulation step, making it impossible to simulate on computers. In the next section, we present an approximation method for obtaining a model in a new formal description called IA-DEVS. Using IA-DEVS we will show computable algorithms to produce approximate sets of behavior trajectories.

## 4 INTERVAL-APPROXIMATED DEVS (IA-DEVS)

Here, we present a formal specification in which uncertainty is characterized by intervals. When compared to UA-DEVS, the use of intervals adds some constraints to what can be modeled and not. However, it also enables us to represent infinite sets using finite variables. An interval is defined by its borders, which can be open or closed. Thus, they can be represented by using two values and two Boolean. Representing the inputs, outputs, states, and time with a finite set of variables is the first step toward producing trajectories of the dynamic behavior of the model in a computable

way. One of the main reasons to define IA-DEVS to approximate UA-DEVS models instead of using a single formalism, is to separate concerns. UA-DEVS describes a system as accurately as possible using variables with uncertainty. Instead, IA-DEVS considers the tradeoffs of increasing uncertainty to simplify the simulation. During modeling, UA-DEVS models could be defined by domain experts, and IA-DEVS could be defined by simulation experts, who may not know the modeling domain, but know better the computational constraints in the software application. In addition, multiple IA-DEVS approximations of the same UA-DEVS model can lead to a different set of trajectories that can be used to refine results by intersecting them, this is because those coming from the original model are included in the results of every approximation.

### 4.1 Atomic Model

In IA-DEVS, in addition to the sets defining the inputs ($X$), outputs ($Y$), and sequential states ($S$), we need to define intervals over them. Thus, we define them all as a quadruple with four fields, the first is named *values* holding the set of values as defined in DEVS, the second is named *order* and defines an order function over the values set, the third and fourth are functions that given an arbitrary subset of the *values* produce lower and upper bound values that can be used to define an interval containing all elements given. These functions are used to provide approximation parameters for the simulation. The bound functions are used by the **Simulator** algorithms to approximate the result set after computing each function.

In addition to the bounds for the defined sets, we need bound functions over $\mathbb{R}$ for the approximations of the outputs for the time advance function. We do not use the supremum, infimum, minimum, or maximum functions, because they may be too expensive (or even impossible) to compute depending on the model. This will be in charge of the modeler. The total state set ($Q$) does not need special order functions we define it as a function of the sequential state set quadruple ($S$), $\mathbb{R}^+$, and the *timebound* function pair which already define orders and bound functions.

The importance of the ordering and bound functions is that they map every value in the set to an interval with borders in computable representations containing them. In addition, the selection of different representations can be used to access results quicker as a tradeoff of degrading accuracy.

Based on these ideas, an IA-DEVS atomic model is a tuple $\langle X, Y, S, timebounds, \delta_{int}, \delta_{ext}, \lambda, ta\rangle$ where:

$X$ is a tuple $\langle X_{values}, X_{order}, X_{lowerbound}, X_{upperbound}\rangle$ characterizing the set of input values;
$Y$ is a tuple $\langle Y_{values}, Y_{order}, Y_{lowerbound}, Y_{upperbound}\rangle$ characterizing the set of output values;
$S$ is a tuple $\langle S_{values}, S_{order}, S_{lowerbound}, S_{upperbound}\rangle$ characterizing the set of sequential states;
$timebounds$ is a pair $\langle T_{lowerbound}, T_{upperbound}\rangle$ of bound functions for time approximations;
$Q : S_{values} \times \mathcal{P}(\mathbb{R}^+)$ is the set of total states; $\delta_{int} : S_p \to S_p$ is the internal state transition function;
$\delta_{ext} : Q_p \times X_p \to S_p$ is the external state transition function;
$\lambda : S_p \to Y_p$ is the output function;
$ta : S_p \to \mathcal{P}(\mathbb{R}^+)$ is the time advance function.
For these functions we define as follows:

$X_p : \mathcal{P}(X_{values}) - \emptyset$ is the set of uncertainty-aware inputs;
$Y_p : \mathcal{P}(Y_{values})$ is the set of uncertainty-aware outputs;
$S_p : \mathcal{P}(S_{values}) - \emptyset$ is the set of uncertainty-aware sequential states;
and $Q_p : \mathcal{P}(Q_{values}) - \emptyset$ is the set of uncertainty-aware total states.

Using the set of values ($X_{values}$, $Y_{values}$, $S_{values}$, $Q$) we produce combinations of values using power-sets as we did for UA-DEVS, and we name them $X_p$, $Y_p$, $S_p$, and $Q_p$. These sets are used to

define the domain and co-domain of $\delta_{int}$, $\delta_{ext}$, $\lambda$, and $ta$. Having the same domain and co-domains allows us to define these functions as in the UA-DEVS model, making the transition from UA-DEVS to IA-DEVS model simple. Similarly to UA-DEVS, we define that an empty set defines a passive model, and the simulation can stop when all the models are passive as we did in UA-DEVS.

$\delta_{int}$, $\delta_{ext}$, $\lambda$, $ta$ use arbitrary domain and co-domains. Although in theory, this is correct, in practice is sometimes impossible to manage these sets, which can have infinite elements (which cannot be bounded and could even be continuous). Then the functions *order*, *lower bound*, and *upper bound* are used to define sets of intervals ($X_I$, $Y_I$, $S_I$, $Q_I$), and we use them to define new functions ($\Delta_{int}, \Delta_{ext}, \Lambda, TA$) that approximate the previous ones using intervals in their definitions, as follows:

$X_I$ is the set of all intervals in $X_{values}$ ordered by $X_{order}$ whose lower bounds are in the Image set of $X_{lowerbound}$ and the upper bounds are in the Image set of $X_{upperbound}$;

$Y_I$ is the set of all intervals in $Y_{values}$ ordered by $Y_{order}$ whose lower bounds are in the Image set of $Y_{lowerbound}$ and the upper bounds are in the Image set of $Y_{upperbound}$;

$S_I$ is the set of all intervals in $S_{values}$ ordered by $S_{order}$ whose lower bounds are in the Image set of $S_{lowerbound}$ and the upper bounds are in the Image set of $S_{upperbound}$;

$\mathbb{R}_I^+$ is the set of all intervals in $\mathbb{R}^+$ ordered by $<$ whose lower and upper bounds are in the Image sets of the *timebound* functions.

$Q_I$ is the set of all pair of intervals where the first component is in $S_I$, and the second component is in $\mathbb{R}_I^+$.

Using these approximations, we define the functions to be used by the simulation algorithms. We note them with uppercase Greek letters to differentiate them from the original functions.

$$\Delta_{int} : S_I \to S_I \qquad \Delta_{int}(s) = \Theta(\delta_{int}(s), S_{lowerbound}, S_{upperbound}),$$

$$\Delta_{ext} : Q_I \times X_p \to S_I \qquad \Delta_{ext}(q, x) = \Theta(\delta_{ext}(q, x), S_{lowerbound}, S_{upperbound}),$$

$$\Lambda : S_I \to Y_p \qquad \Lambda(s) = \Theta(\lambda(s), Y_{lowerbound}, Y_{upperbound}),$$

$$TA : S_I \to \mathbb{R}_I^+ \qquad TA(s) = \begin{cases} \emptyset & \texttt{if } ta(s) = \emptyset \\ \Theta(ta(s), timebounds_{lower}, timebounds_{upper} & \texttt{otherwise}, \end{cases}$$

$$\texttt{where } \Theta(vals, lb, ub) = \begin{cases} [lb(vals), ub(vals)] & \texttt{if } lb(vals) \in vals \wedge ub(vals) \in vals \\ (lb(vals), ub(vals)] & \texttt{if } lb(vals) \notin vals \wedge ub(vals) \in vals \\ [lb(vals), ub(vals)) & \texttt{if } lb(vals) \in vals \wedge ub(vals) \notin vals \\ (lb(vals), ub(vals)) & \texttt{if } lb(vals) \notin vals \wedge ub(vals) \notin vals \end{cases}.$$

The approximated functions of the model are similar to the original transition functions. Each approximation function follows the same principle: The variables are passed to the original function, and an interval including all the results of the original function is used as a result. The interval is constructed using the bound functions. It may be impossible to compute the original functions for an infinite set of values; for example, it is impossible to compute all the square roots of the Real numbers between 1 and 3; but defining a function to limit these values is trivial. For example, the *id* function, as all square roots of Real numbers between 1 and 3, are in fact in the interval $[1, 3]$.

These approximations allow us to represent arbitrary sets by only using two elements (upper bound and lower bound), and by defining if the boundaries are open or closed. The advantage of interval variables is that they can represent an infinite set of values in a compact form that can be used to make the simulation simpler—but at the expense of calculation errors. The errors are treated so they only add unexpected trajectories to the set of solutions, and a valid trajectory is never missed. For example, if $S_{values}$ includes all the natural numbers between 2 and 100, the order function for the approximation is $<$, the lower and upper bound functions are *minimum* and

*maximum*, the values "2, 4, 6" are approximated by the interval [2, 6] which also includes 3 and 5. These two additional values will propagate through the simulation introducing new trajectories; nevertheless, the original trajectories remain; we just increment the number of elements.

### 4.2 Coupled Model

IA-DEVS coupled models need new definitions. We need bounding functions and order functions for $X$ and $Y$ as we did for the atomic models. Although all the inputs will be intervals (as they can only be produced by atomic models), using $Z$ functions as in UA-DEVS can produce infinite discontinuous intervals. The output of these functions must be bound to a single interval. The bounds over $Y$ are used for external output couplings, and the bounds over $X$ are used for external input couplings, internal couplings use $X$ and $Y$ bounds defined in the models been connected. We formally define an IA-DEVS coupled model as $C = \langle X, Y, D, M, Z, SELECT \rangle$ where:

$X$ is a tuple $\langle X_{values}, X_{order}, X_{lowerbound}, X_{upperbound} \rangle$ characterizing the set of input values;

$Y$ is a tuple $\langle Y_{values}, Y_{order}, Y_{lowerbound}, Y_{upperbound} \rangle$ characterizing the set of output values;

$X_p : \mathcal{P}(X_{values}) - \emptyset$ is the set of uncertainty-aware inputs;

$Y_p : \mathcal{P}(Y_{values})$ is the set of uncertainty-aware outputs;

$X_I$ is the set of all intervals in $X_{values}$ ordered by $X_{order}$ whose lower bounds are in the Image set of $X_{lowerbound}$ and the upper bounds are in the Image set of $X_{upperbound}$;

$Y_I$ is the set of all intervals in $Y_{values}$ ordered by $Y_{order}$ whose lower bounds are in the Image set of $Y_{lowerbound}$ and the upper bounds are in the Image set of $Y_{upperbound}$;

For each $d \in D, M_d$ is an IA-DEVS model, for each $d \in (D \cup self), I_d$ is the influencer set of $d : I_d \subseteq (D \cup self), d \notin I_d$ and for each $i \in I_d$, $Z_{i,d}$ is the i to d output translation function; and $SELECT : (\mathcal{P}(D) - \emptyset) \rightarrow D$ is the tie-breaker function in case of simultaneous events. First, $X$ and $Y$; they are defined as in the IA-DEVS atomic model, they are quadruples with *values*, *order*, *upper bound*, and *lower bound* fields. Second, we define $X_p$, $Y_p$, $X_I$, and $Y_I$ to describe all uncertainty sets, and all the intervals approximating those sets. Third, as in UA-DEVS, the model structure is as in DEVS. Finally, the translation ($Z$) and $SELECT$ functions are similar to UA-DEVS. In addition, we define new translation functions using the intervals for approximation ($Z^{IA}$) below.

For each $d \in D$, $M_d$ is an IA-DEVS model, for each $d \in (D \cup self)$, $I_d$ is the influencer set of $d : I_d \subseteq (D \cup self), d < I_d$, and for each $i \in I_d$, $Z^{IA}_{i,d}$ is defined as

if $d = self$:

$$Z^{IA}_{i,d} : M_i.Y_I \rightarrow M_{self}.Y_I,$$
$$Z^{IA}_{i,d}(x) : \Theta(Z_{i,d}(x), Y_{lowerbound}, Y_{upperbound}),$$

otherwise:

$$Z^{IA}_{i,d} : M_i.Y_I \rightarrow M_d.X_I,$$
$$Z^{IA}_{i,d}(x) : \Theta(Z_{i,d}(x), M_d.X_{lowerbound}, M_d.X_{upperbound}).$$

The definition of the approximated translation functions ($Z^{IA}$) uses intervals as input and output. In the case of the external input couplings (those connecting inputs in a model with inputs of its sub-models), the domain of the function is $X_I$ of the coupled models, and the co-domain is in $X_I$ of the receiving sub-model. In the case of external output couplings (those connecting the output of sub-models with the model outputs), the functions have domain in $Y_I$ of the origin sub-model and co-domain in $Y_I$ of the coupled model. Finally, in the case of internal couplings (those connecting sub-models outputs to inputs), the functions have domain in $Y_I$ of the sub-model origin of the translation and co-domain in $X_I$ of the receiving models. The functions are built using the previous definition of transition functions and applying bound functions to them.

For $Z^{IA}$ functions, the bound functions of the input and output need to be applied to allow reutilization of models, otherwise, we would need to requirements for each input and output participating in a coupling to match completely their definitions.

### 4.3 Summary

We defined a new formal specification of models including uncertainty specifications as intervals. Its goal is to approximate general UA-DEVS models to create computable simulations. Having intervals in place of general sets allows us to keep track of the simulation state, messages, and time in quadruples (the intervals) when UA-DEVS models may need uncountable sets.

To work with intervals, we needed to add approximation parameters as order and bounding functions. The approximation by interval introduces errors in the uncertainty propagation. However, these errors only introduce new non-reachable trajectories, and reachable ones are never discarded. When IA-DEVS is used to approximate a UA-DEVS model, the set of trajectories of its simulation are expected to super-set those from the UA-DEVS model, as shown in Section 7.

Having IA-DEVS and UA-DEVS definitions allows us to separate concerns. On one hand, UA-DEVS allows us to model the system and its uncertainties accurately. On the other hand, IA-DEVS allows us to simulate the UA-DEVS model based on computational constraints (time, memory, etc.). This separation of concerns allows the domain expert to define the model once, and then we can simulate it with different constraints without redefining the model.

## 5 UA-DEVS TO IA-DEVS APPROXIMATION PROCESS

The process for approximating an atomic IA-DEVS model consists of four steps:

— Defining the input set, order, and bounds ($X$),
— Defining the output set, order, and bounds ($Y$),
— Defining the sequential state set, order, and bounds ($S$), and
— Defining the $timebound$ functions.

First, we copy the definition of the $\delta_{int}$, $\delta_{ext}$, $\lambda$, and $ta$ functions defined in the UA-DEVS model has been approximated. Then, define order and bounding functions for $X$, $Y$, $S$, and $time$. The simulation algorithms for IA-DEVS, defined in the next section, only compute $\Delta_{int}$, $\Delta_{ext}$, $\Lambda$, and $TA$ functions; all other functions are part of the theoretical framework to produce them. In Section 4, we showed how $\Delta_{int}$, $\Delta_{ext}$, $\Lambda$, and $TA$ are mechanically derived from $X$, $Y$, $S$, $timebounds$, $\delta_{int}$, $\delta_{ext}$, $\lambda$, and $ta$. Choosing different order and bounding functions for $X$, $Y$, $S$, and $time$ allows us to adjust domain and image set for the $\Delta_{int}$, $\Delta_{ext}$, $\Lambda$, and $TA$, determining if those functions are computable or not, the expectations about errors introduced by approximation, and the complexity of the calculations.

For example, if we have a UA-DEVS model with $X = \mathbb{R}$, we define $X_{values} = \mathbb{R}$, and we must choose functions for $X_{order}$, $X_{lowerbound}$, and $X_{upperbound}$. A common example would be to choose $<_{\mathbb{R}}$ as the order function. For the bounding functions, we can start from a target computer data type, e.g., 32-bit IEEE754 floating point (FP32), and work backward. We define bounding functions from any subset of $\mathbb{R}$ into the finite set of values that FP32 can represent. A trivial bound function with the image in FP32 would be to define them as constant functions $\infty^-$, and $\infty^+$. This produces a single input interval for $\Delta_{ext}$ which is $(\infty^-, \infty^+)$. Other alternatives include, but are not limited to

— the lower bound is the maximum FP32 value lower than the lowest value in the input set, and the upper bound is the minimum FP32 value strictly above the highest value of the input set.

— the lower bound is the maximum FP32 value lower than or equal to the lowest value in the input set, and the upper bound is the minimum FP32 value above the highest.
— the lower bound is $\infty^-$, and the upper bound is the minimum FP32 value above the highest value in the input set.
— if using an integer in place of FP32 for representation, the lower bound could be the floor function, and the upper bound the ceiling. We should reserve two values to represent infinities.

Let us show the process for approximating the Generator model. If we consider the approximation definition for the $Generator_{UA}$ model as presented in Section 3, we can choose to order elements in $X$ and $S$ using the lower-than function of $\mathbb{R}$, and the $Y$ elements by a custom order function, as follows:

$X = \langle \mathbb{R}, <_{\mathbb{R}}, x \to \infty^-, x \to \infty^+ \rangle \qquad Y = \langle \{1,2\}, <_{bin}, y \to 1, y \to 2 \rangle$ where $1 <_{bin} 2 \wedge 2 \not<_{bin} 1$

$S = \langle \mathbb{R}, <_{\mathbb{R}}, s \to s > -1000?IntFloor(inf(s) * 100)/100 : \infty^-, s \to s < 1000?IntCeil(sup(s) * 100)/100 : \infty^+ \rangle$

$timebounds(t) = \langle t \to IntFloor(inf(t) * 100)/100, t \to t < 1000?IntCeil(sup(t) * 100)/100 : \infty^+ \rangle$

Here, the bound functions for $X$ are constant with infinity as result to avoid calculations (always producing the interval $(\infty^-, \infty^+)$, as the model discards all inputs). This is a case the modeler can notice and use as an optimization for the simulation. Similarly, we keep fixed bounded results for $Y$. For $S$ and $timebounds$, we round the bounds to integer multiples of 0.01. This allows a simple representation of the states using pairs of decimal fixed-point variables. The decimal representation should also support infinity for high and low values to keep a finite representation, in this case, we use infinity for values below "1,000 and above 1,000".

A second example shows the IA-DEVS approximation of the UA-DEVS Processor model presented in Section 3. In this case, we are going to use the $<$ function of $\mathbb{N}$ for the order of $X$ and $Y$, and minimum/maximum bounds capped on 32, 768 to be able to fit them in a 16 bits integer variable. For $S$, we first define an order function comparing the $S_{values}$ first component (TOCJ); when they are equal, we use the second component (QJ) to decide. For $timebounds$ we round to integer multiples of 0.001, using the floor and ceiling for lower/upper bounds and infinite for values larger than 1,000$ms$.

The formal definition of $Processor_{IA}$ is then the tuple $P = \langle X, Y, S, timebounds, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$ where:

$X = \langle \mathbb{N}, <_{\mathbb{N}}, min(x, <_{\mathbb{N}}), x <_{\mathbb{N}} 32768?max(x, <_{\mathbb{N}}) : \infty^+ \rangle; \quad Y = \langle \mathbb{N}, <_{\mathbb{N}}, min(y, <_{\mathbb{N}}), y <_{\mathbb{N}} 32768?max(y, <_{\mathbb{N}}) : \infty^+ \rangle;$

$S_{values} = \langle TOCJ, QJ \rangle$ with $TOCJ(\mathbb{R}^+)$ the time of processing, and $QJ$ a queue of processes $(\mathbb{N})$;

$S_{order}(a, b) = a.TOCJ <_{TOCJ} b.TOCJ \vee (a.TOCJ = b.TOCJ \wedge a.QJ <_{QJ} b.QJ)$ where:

$a <_{TOCJ} b \Leftrightarrow a <_{\mathbb{R}} b;$

$a <_{QJ} b \Leftrightarrow (a.size < b.size \vee (a.size = b.size \wedge a \neq \emptyset \wedge (a.first < b.first \vee dequeue(a) <_{QJ} dequeue(b))));$

$S_{lowerbound}(s) = IntFloor(inf(s.TOCJ, <_{TOCJ}) * 100)/100, min(s.QJ, <_{QJ}) \rangle;$

$S_{upperbound}(s) = IntCeil(sup(s.TOCJ, <_{TOCJ}) * 100)/100, max(s.QJ, <_{QJ}) \rangle;$ and

$timebounds(t) = \langle IntFloor(inf(t) * 1000)/1000, t < 1000?IntCeil(sup(t) * 1000)/1000 : \infty^+ \rangle.$

Finally, using the atomic models above, we show how to compose the IA-DEVS version of the coupled model of Figure 1. The specification only differs from the UA-DEVS in the $X$ and $Y$ sets definition. Since $X$ is not connected to any model, we define it as a set of reals with bounds in the infinity. In the case of $Y$, we use the same specification as the output of the Processor model.

$X = \langle \mathbb{R}, <_{\mathbb{R}}, \infty^-, \infty^+ \rangle \qquad Y = \langle \mathbb{N}, <_{\mathbb{N}}, min(y, <_{\mathbb{N}}), y <_{\mathbb{N}}?max(y, <_{\mathbb{N}}) : \infty^+ \rangle.$

## 6 IA-DEVS SIMULATION ALGORITHMS

In this section, we present an Abstract Simulation algorithm for IA-DEVS. Similarly to DEVS abstract simulator, we define three components: **Simulator**, **Coordinator**, and **Root Coordinator**. Each of them presents a similar interface to the algorithms for DEVS presented in [20].

In previous work [17], we discussed algorithms for simulating introducing a set of external events with uncertainty to a DEVS atomic model to produce a set of trajectories. The trajectories were guaranteed to produce a superset of trajectories expected to be generated by evaluating all the events in the uncertainty intervals. This work allowed us to explore basic conflict resolution of multiple events overlapping their uncertain occurrence intervals while simulating. The algorithms were limited to simulate only atomic DEVS models. Thus, uncertainty in models could not be modeled, only was allowed on the input. In previous sections, we introduced new formalisms to specify models with uncertainty as part of their definition and here, we add support to handle those new definitions and simulate both atomic and coupled models.

### 6.1 Simulator

The **Simulator** (Algorithm 1) manages atomic models. The only parameter needed for a **Simulator** is the model to be simulated. Three internal variables track the simulation progress: $t_{last}$ holds the time of the last processed event; $t_{next}$ holds the time of the next scheduled internal event; and $state$ holds the last computed sequential state. All these variables are expressed as intervals with bounds in timebound functions image set and $S_I$ set.

We mentioned in Section 2.2 that DEVS algorithms were specified using four functions: *init-function*, **-function*, *x-function*, and *y-function*. The algorithm we present here uses only three functions: *init-function*, **-function*, and *x-function*. The *y-function* is embedded in the other functions to provide a sequential algorithm per simulation branch. In IA-DEVS, algorithms use concurrency to explore multiple simulation paths that a trajectory could advance based on the model uncertainty. We believe that using sequential algorithms inside each of the explorations keeps algorithms simpler to understand [16]. It is possible to translate the abstract algorithms into a distributed implementation per exploratory branch, as far as race conditions and branch explosion are considered.

The *init-function*, used for initializing the simulation, receives an interval of total states ($q$) and an interval of a current simulation time ($t$). The state variable is assigned with the sequential state component of $q$, and $t_{last}$ is computed as the current time ($t$) minus the elapsed time component of $q$. Finally, the $t_{next}$ is computed as the obtained $t_{last}$ plus the time advance ($TA$) for the current state. The initial time and the initial states are not a part of the model; we need to pass them as parameters, which allows stopping and resuming simulations, skipping the processing of well-known parts of the simulation, or setting up multiple initial conditions without changing the model.

The **-function* (or advance-function) advances the simulation at the time of internally scheduled events. First, we get the interval of output values for the current state using $\Lambda$ computed from the model. Second, a new interval of sequential states is set using $\Delta_{int}$ computed from the model. The call to $\Delta_{int}$ does not require a time parameter, given it is implicitly known as the time-advance of the current state. Third, $t_{last}$ is set to the current time, and $t_{next}$ is the current time plus the $TA$ function results. Finally, we return the interval of output values that was obtained at the start of the function.

In the **-function* first two lines, we check if parameter $t$ is included in $t_{next}$; if not, we throw an error. This invariant helps early detection of implementation errors. We return an interval of output values; however, we set the return type as an Optional of output values (to keep the same interface for **Simulator** and **Coordinator**; we will discuss the Optional type in the next section).

**ALGORITHM 1:** Simulator for IA-DEVS

```
1  Atomic IA-DEVS a
2  R_I^+ t_last
3  R_I^+ t_next
4  a.S_I state
5  Function init(initstate q, R_I^+ t)→ void
6      state = q_state
7      t_last = (t − q_time).apply(timebounds)
8      t_next = (t_last + a.TA(state)).apply(timebounds)
9  Function *-function(R_I^+ t)→ Optional < a.Y >
10     if t ⊈ t_next then
11         RAISE ERROR
12     a.Y_I y = a.Λ(state)
13     state = a.Δ_int(state)
14     t_last = t
15     t_next = (t_last + a.TA(state)).apply(timebounds)
16     return Optional.of(y)
17 Function x-function(a.X_I x, R_I^+ t) → void
18     R_I^+ t_local
19     t_local.upperend = t.upperend − t_last.lowerend
20     t_local.upperclosed = t.upperclosed ∧ ¬t_last.lowerclosed
21     if t ∩ t_last ≠ ∅ then
22         t_local.lowerend = 0
23         t_local.lowerclosed = true
24     else
25         t_local.lowerend = t_lowerend − t_last.upperend
26         t_local.lowerclosed = t.lowerclosed ∧ t_last.upperclosed
27     state = a.Δ_ext(⟨state, t_local⟩, x)
28     t_last = t
29     t_next = (t_last + a.TA(state)).apply(timebounds)
```

The *x-function* processes external events. First, we define a relative time interval ($t_{local}$) that represents an interval between the possible times of the event being introduced and the possible times of the last event. This does not match with the definition of the difference between intervals, so we do not use the interval subtract operation for it. The lower end of the $t_{local}$ is set to zero when the introduced event interval overlaps with the interval of the last event processed. Here, we account for the case that both events are occurring at the same time, but it is decided which to execute first based on the *SELECT* function. The upper end is calculated as the largest difference between a value in the $t_{last}$ interval and one in the $t$ interval, this is the difference between the largest value in $t$ and the lowest in $t_{last}$. There is no concern about the upper end of $t_{local}$ being higher than the next scheduled internal event; that case is managed at the **Coordinator** level, as we will show in the next section.

Once the local time is known, we advance the simulation by using the $\Delta_{ext}$ computed function to change the current state, and we set $t_{last}$ and $t_{next}$ as we do in **-function*.

At the end of every function, we normalize the $t_{next}$ variable value using the timebound functions to be certain the result is representable.

### 6.2 Coordinator

The **Coordinator** (Algorithm 2) manages the simulation of coupled models. The only parameter for the construction of a **Coordinator** is the model to be simulated. Three internal variables track the simulation progress: $t_{last}$ holds the time of the last processed event; $t_{next}$ holds the time of the next expected internal event; and $E$ holds a tuple of Engines (**Simulators** or **Coordinators**) for simulating the sub-models of the model being simulated ($c$). Similarly to the **Simulator**, time variables are expressed as intervals, and three functions are defined: *init-function*, **-function*, and *x-function*.

**ALGORITHM 2:** Coordinator for IA-DEVS

```
1  Coupled IA-DEVS c
2  ℝ_I^+ t_last
3  ℝ_I^+ t_next
4  Tuple of Engines E /* Coordinators and Simulators                    */
5  Function init(initstate q, ℝ_I^+ t)→ void
6      forall the d ∈ c.D do
7          if c.M_d is atomic then
8              E_i = Simulator(M_i)
9          else
10             E_i = Coordinator(M_i)
11         E_i.init(q_i, t)
12     BoundTs()
13 Function x-function(a.X x, ℝ_I^+ t) → void
14     forall the c.I_self : i do
15         x-function(Z^IA_self,i(x, t))
16     BoundTs()
17 Function BoundTs() → void
18     t_last.upperend = max(E_i.t_last.upperend|i ∈ c.D)
19     t_last.lowerend = max(E_i.t_last.lowerend|i ∈ c.D)
20     t_last.upperclosed = ∃i : t_last.upperend ∈ E_i.t_last
21     t_last.lowerclosed = ∀i ∈ c.D, E_i.t_last.lowerend = t_last.lowerend ⇒ E_i.t_last.lowerclosed
22     t_next.upperend = min(E_i.t_next.upperend|i ∈ c.D)
23     t_next.lowerend = min(E_i.t_next.lowerend|i ∈ c.D)
24     t_next.upperclosed = ∀i ∈ c.D, E_i.t_next.upperend = t_next.upperend ⇒ E_i.t_next.upperclosed
25     t_next.lowerclosed = ∃i : t_next.lowerend ∈ E_i.t_next
   /* Continues in Algorithm 3                                          */
```

The *init-function* is used for initializing the simulation. First, we construct the tuple of engines ($E$) based on the sub-models of $c$. For each atomic sub-model, we construct a **Simulator**, and for each coupled sub-model, we construct a **Coordinator**. After initialization, we treat these engines indistinctly, that is why we used the exact same interface for **Simulators** and **Coordinators** and called it the Engine interface. The *init-function* receives a tuple of initialization values ($q$) and a time interval ($t$). Each element in the $q$ tuple represents the initialization values of an engine simulating a sub-model of $c$. For **Simulator** engines, the element forwarded is an interval of total states, while for **Coordinator** engines, the element forwarded is a tuple of initialization values. For deciding which element to forward, the tuple is indexed by model identifiers ($D$). In addition to

**ALGORITHM 3:** Coordinator for IA-DEVS Continuation

```
1  Function *-function(ℝ_I^+ t)→ Optional < a.Y >
2      List of Engines Imminents
3      Imminents = d|d ∈ c.D, E_d.next ∩ t ≠ ∅ sorted by ≪_ℝ+
4      Interval of ℝ^+ limit
5      limit = first(Imminents).t_next ∩ t
6      if ∃i∀k∃p|i ∈ Imminents, k ∈ E_i.t_next, p ∈ E_first(imminents).t_next, p < k then
7          limit.upperend = E_i.t_next.lowerend
8          if E_i.lowerclosed then
9              limit − {E_i.lowerend}
10     if limit.lowerend = limit.upperend ∧ limit.closed then
11         Engine E_next = c.SELECT({E_i|i ∈ Imminents ∧ E_i.t_next ∩ limit ≠ ∅})
12         if limit ≠ E_next.t_next then
13             FORK
14             On Child
15                 E_next.t_next = E_next.t_next − limit
16                 return this.*-function(t)
17             c.Y y = route(E_next, limit)
18             BoundTs()
19             return y
20     forall the {i|i ∈ Imminents ∧ E_i.t_next ∩ limit ≠ ∅} : i do
21         FORK
22         On Child
23             c.Y y = route(E_i, limit)
24             BoundTs()
25             return y
26     if ∀i|i ∈ Imminents ⇒ E_i.t_next − limit ≠ ∅ then
27         forall the {i|i ∈ Imminents} do
28             E_i.t_next = E_i.t_next − limit
29     else
30         EXIT
31 Function route(Engine e, Interval ℝ^+ t)→ Optional < a.Y >
32     y = e.*-function(t)
33     forall the {r|r ∈ c.I_i} : r do
34         E_r.x-function(Z^IA_{i,r}(y, t)
35         if self ∈ I_i then
36             return Z^IA_{i,self}(y)
37         else
38             return nil
```

the initialization values, every engine receives the time interval parameter ($t$). After every engine is initialized, the time values are set using the helper function *BoundTs*.

*BoundTs* function collects the $t_{last}$ and $t_{next}$ intervals from all elements in $E$ and constructs a single interval for each variable. The $t_{last}$ is set to an interval containing the most recent $t_{last}$

values. The $t_{next}$ is set with the closest to happen interval of event times scheduled in $E$. Depending on competing scenarios between engines in $E$, the intervals may be adapted and not match any of the original ones.

For coupled models, we do not define timebound functions as in atomic models. We do not want to extend the calculated set of values obtained by $E$ elements after resolving competing scenarios, because those scenarios could be reintroduced. Also, we know that all resulting intervals are representable, given the ends are ends of those represented already in $E$ elements.

The **-function* advances the simulation when an internal event is scheduled. First, we find the imminent engines, which are engines from $E$ candidate executing its **-function* inside the $t$ interval passed as a parameter. We keep the list of imminent models sorted by applying $\ll$ to the engines' $t_{next}$. The $\ll$ function sorts intervals first by the lowest value in each set, and when the lowest value is the same, it sorts by the higher value contained in each set. Once the imminent model is chosen, we define a $limit$ variable setting the interval of event times that we can advance the simulation in a single step. This interval is chosen by removing all the event times described by $t_{next}$ intervals starting later than the first interval in the list and restricting the upper-end to be as most as the lowest of the intervals starting at the same time as the first engine's $t_{next}$ of the list of imminent engines.

If the limit interval is punctual, we use *SELECT* to decide which engine to advance first. If the punctual $limit$ interval is included in the $t_{next}$ of the engine is advanced, we branch simulation for this case, and the case where the interval does not include this particular point. Else, we advance without branching. If the $limit$ interval is not punctual, we branch the simulation in as many cases as imminent engines we got on the list. In each branch, we advance simulation by a different engine first. In both cases, punctual and non-punctual $limit$ intervals, the output from the imminent is routed using *x-function* to other models accordingly to the coupling definitions described by $I$ and $Z^{IA}$. Finally, if the model is an influencer of $self$, we use the corresponding $Z^{IA}$ function to return the interval of $Y$ values or return nothing (empty Optional). After the simulation advances in all elements of $E$ requiring it, the *BoundTs* function is used to obtain the new intervals for $t_{last}$ and $t_{next}$.

The *x-function* processes external events. In case an external event is received by a **Coordinator**, we review the set of influenced ($I$) and use the computed $Z^{IA}$ to make necessary transformations and route the event to the engines in $E$ requiring it by calling *x-function* for each of them. After all engines processed the events sent for processing, we use the *BoundTs* function to set $t_{next}$ and $t_{last}$.

We apply *BoundTs* functions after each of the operations. This is to enable modelers to use flexible data types that may not be closed under addition, for example, in Floating Point adding two values in the domain may approximate the result to a value close to its arithmetic result. For those datatypes, when adding $t_{next}$ to $TA$, if the results need to be approximated for the interval borders we use the *BoundTs* function to guarantee that the borders are approximated in the directions that include all the values required.

### 6.3 Root Coordinator

**Root Coordinator**, the main-loop of the simulation, uses two variables, one to hold the reference to a **Coordinator** of the topmost coupled model ($E$), usually called top-model, and the other to keep track of current time as an interval ($t_{current}$).

The **Root Coordinator** has a single function called $simulate$, which uses three parameters, the top coupled model ($c$), the initialization values tuple ($q$), and the initial time of the simulation ($t$). First, a **Coordinator** is created for $c$ and assigned to the variable $E$. Second, $q$ and $t$ are forwarded to $E$ for initialization, and $t_{current}$ is set to $E$'s $t_{next}$ value. Finally, the simulation loops calling $E$'s **-function* with $t_{current}$ and advancing $t_{current}$ to $E$'s $t_{next}$ value until it reaches a passive state ($t_{current} = \emptyset$).

## 7 CASE STUDY

In this section, we present the steps for generating behavior trajectories for the example model described in previous sections using the algorithms proposed.

### 7.1 Derived Model Functions

First, we need to obtain the derived functions for both atomic models and the top coupled model. For this purpose, we apply the functions provided in Section 4 to the definitions provided in each example in Section 5.

The functions derived for the $Generator_{IA}$ model are as follows:

$\Delta_{int}(s) = \Theta(\delta_{int}(s), S_{lowerebound}, S_{upperbound}) \therefore \Delta_{int}(s) = \Theta(\{0\}, S_{lowerebound}, S_{upperbound})$, $\therefore \Delta_{int}(s) = [0,0]$,

$\Delta_{ext}(q,x) = \Theta(\delta_{ext}(q,x), S_{lowerbound}, S_{upperbound})$ where $\delta_{ext}(s) = \{(s+e)|s \in q.state \wedge e \in q.time\}$,

$\Lambda(s) = \Theta(\lambda(s), Y_{lowerbound}, Y_{upperbound}) \therefore \Lambda(s) = \Theta(\{1,2\}, Y_{lowerbound}, Y_{upperbound}) \therefore \Lambda(s) = [1,2]$

$$TA(s) = \begin{cases} \emptyset & \texttt{if } (ta(s) = \emptyset) \\ \Theta(ta(s), T_{lowerbound}, T_{upperbound}) & \texttt{otherwise} \end{cases}$$

where $ta(s) = \{(l-e)|l \in [997, 1005]ms \wedge e \in s\}$

The functions derived for the $Processor_{IA}$ model are as follows:

$\Delta_{int}(s) = \Theta(\delta_{int}(s), S_{lowerebound}, S_{upperbound})$ where $\delta_{int}(s) = \{\langle 0, dequeue(k.QJ)\rangle | k \in s\}$

$\Delta_{ext}(q,x) = \Theta(\delta_{ext}(q,x), S_{lowerbound}, S_{upperbound})$ where $\delta_{ext}(q,x) = \{\langle 0, queue(k.state.QJ, x)|k \in q \wedge k.state.QJ.size = 0\} \cup \{\langle k.state.TOCJ - k.elapsed, queue(k.state.QJ, x)\rangle | k \in q \wedge k.state.QJ \neq \emptyset\}$

$\Lambda(s) = \Theta(\lambda(s), Y_{lowerebound}, Y_{upperbound})$ where $\lambda(s) = \{k.QJ.first|k \in s\}$

$$TA(s) = \begin{cases} \emptyset & \texttt{if } (ta(s) = \emptyset) \\ \Theta(ta(s), T_{lowerbound}, T_{upperbound}) & \texttt{otherwise} \end{cases}$$

$$\text{where } ta(s) = \begin{cases} \{k.TOCJ|k \in s\} & \texttt{if } \nexists q|q \in s \wedge q.QJ = \emptyset \\ \{\infty^+\} \cup \{k.TOCJ|k \in s \wedge k.QJ \neq \emptyset\} & otherwise \end{cases}$$

The derived $Z^{IA}$ for the top model are:

For each $d \in D, M_d$ is a IA-DEVS model, for each $d \in (D \cup self), I_d$ is the influencer set of $d : I_d \subseteq (D \cup self), d \notin I_d$ and for each $i \in I_d, Z^{AI}_{i,d}$ is defined as

$$Z^{AI}_{i,d}(x) = \begin{cases} \{(a,b) + c|a = Y_{lowerbound}(c), b = Y_{upperbound}(c)\} & \texttt{if } d = self \\ \{(a,b) + c|a = M_d.X_{lowerbound}(c), b = M_d.X_{upperbound}(c)\} & \texttt{otherwise} \end{cases}$$

where $c = Z_{i,d}(x)$

Applying to each case in the UA-DEVS coupled model definition of $Z$ we obtain: $Z^{IA}_{Gi,P}(v) = \Theta(\{i\}, P.X_{lowerbound}, P.X_{upperbound}) = [i,i]$, where $i$ in $\{1,2,3,4\}$, $Z^{IA}_{P,self}(v) = (Y_{lowerbound}(v), Y_{upperbound}(v)) + v$.

### 7.2 Initialization

We define the initialization values provided to the top coupled model as a tuple of initialization values for each subcomponent. Each generator starts with a sequential state interval of $[0,0]$, and a time interval of $[0,0]ms$. The Processor starts with an empty queue and with no scheduled internal transition. Formally, we write $C_{init} = \langle G1_{init}, G2_{init}, G3_{init}, G4_{init}, P_{init}\rangle$ where:

$$G1_{init} = G2_{init} = G3_{init} = G4_{init} = \langle \underbrace{[0,0]}_{state}, \underbrace{[0,0]}_{time}\rangle, \text{ and } P_{init} = \langle \underbrace{[\langle \overbrace{0}^{TOCJ}, \overbrace{\emptyset}^{QJ}\rangle, \langle \overbrace{0}^{TOCJ}, \overbrace{\emptyset}^{QJ}\rangle]}_{state}, \underbrace{\emptyset}_{time}\rangle$$

Table 1. Simulators Variables After Initialization

| | G1 | G2 | G3 | G4 | P |
|---|---|---|---|---|---|
| $state$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[\langle 0, \emptyset \rangle, \langle 0, \emptyset \rangle]$ |
| $t_{last}$ | $[0, 0]ms$ | $[0, 0]ms$ | $[0, 0]ms$ | $[0, 0]ms$ | $[0, 0]ms$ |
| $t_{next}$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1.005]ms$ | $\emptyset$ |

**ALGORITHM 4:** Root Coordinator for IA-DEVS

```
1 Coordinator E
2 R_I^+ t_current /* Current time                                    */
3 Function simulate(Coupled c, initstate q, R_I^+ t)→ void
4     E = Coordinator(c)
5     E.init(q, t)
6     t_current = E.t_next
7     while t_current ≠ ∅
8         E.*-function(t_current)
9         t_current = E.t_next
```

Following, we show how to initialize a model $C$ with the values $C_{init}$. First, the **Root Coordinator** initializes the variables as described in Algorithm 4. A **Coordinator** ($E$) is defined for model $C$ and its *init-function* is called with $C_{init}$ and initial time ($t$) parameters. In the *init-function* of the Coordinator Algorithm 2, the **Coordinator** will create engines (**Coordinators** or **Simulators**) for each sub-model and forward each initialization element to the corresponded subcomponent's engine. When an init message is forwarded to a **Simulator**, Algorithm 1, the state is assigned to the internal variable $state$, the time component of the init is assigned to the $t_{last}$ variable, and the value of the $t_{next}$ variable is computed as the addition of $TA$ to $t_{last}$. The passed $t$ value represents the current time of the simulation and its only used for validation purposes by checking that it is always between $t_{last}$ and $t_{next}$ for every initialized model. After all sub-models' engines of a **Coordinator** complete their initialization, the *BoundTs* function is used to set the **Coordinators'** $t_{last}$ and $t_{next}$ values. Finally, the $t_{next}$ value of the **Coordinator** of the top model ($E$) is assigned as $t_{current}$ in the **Root Coordinator**.

Table 1 shows the values on the local variables for the **Simulator** of each atomic model at the end of the initialization.

In addition to the variables in the **Simulators**, the **Coordinator** sets $t_{last}$ to $[0, 0]ms$ and $t_{next}$ to $[997, 1{,}005]ms$ using the *BoundTs* function. Similarly, the **Root Coordinator** sets $t_{current}$ to $[997, 1{,}005]ms$.

### 7.3 Advancing Simulation

After initialization is completed, the **Root Coordinator** iteratively advances the simulation. At each step, the $^*$*-function* of the top-level **Coordinator** ($E$) is called to simulate the events of the current time ($t_{current}$), which is then updated with the new $t_{next}$ of the **Coordinator**.

In the **Coordinator**, a set of imminent engines is chosen, in this case, the **Simulators** of all the Generators. Using the first imminent and the current time, an advance *limit* is defined. The *limit* has the role of preventing advancing when we have conflicting events. A conflict would be produced when two or more events occur within non-equal overlapping intervals on the timeline. When this happens, the *limit* is used to break the intervals in smaller cases where intervals fully overlap, or they do not overlap at all. In this example, all imminent have fully overlapped

Table 2. **Simulators** Variables After First Step of Simulation when Advancing G1's **Simulator** First

| | G1 | G2 | G3 | G4 | P |
|---|---|---|---|---|---|
| $state$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[\langle 0, \langle 1 \rangle \rangle, \langle 0, \langle 1 \rangle \rangle]$ |
| $t_{last}$ | $[997, 1{,}005]ms$ | $[0, 0]ms$ | $[0, 0]ms$ | $[0, 0]ms$ | $[997, 1{,}005]ms$ |
| $t_{next}$ | $[1{,}994, 2{,}010]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[1{,}247, 1{,}255]ms$ |

Table 3. **Simulators** Variables After Four Steps in G1-G2-G3-G4 Branch

| | G1 | G2 | G3 | G4 | P |
|---|---|---|---|---|---|
| $state$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[\langle 0, \langle 1, 2, 3, 4 \rangle \rangle, \langle 0, \langle 1, 2, 3, 4 \rangle \rangle]$ |
| $t_{last}$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ |
| $t_{next}$ | $[1{,}994, 2{,}010]ms$ | $[1{,}994, 2{,}010]ms$ | $[1{,}994, 2{,}010]ms$ | $[1{,}994, 2{,}010]ms$ | $[1{,}247, 1{,}255]ms$ |

occurrence intervals ($[997, 1{,}005]ms$), which also overlap with the current time. Then the limit is the occurrence of them all ($[997, 1{,}005]ms$).

In lines 10–19 of the **Coordinator**'s **-function*, Algorithm 3, we advance the simulation for scenarios where the interval of event times is punctual, meaning having a single value. In this example, we do not run these lines because the interval been evaluated is not punctual.

The rest of the function forks the simulation to have a branch for each imminent engine advancing inside the $limit$. In each branch, a different engine is treated as imminent. The imminent's **-function* is called in the sub-engines until we reach a **Simulator**. The **Simulator**'s **-function*, Algorithm 1, uses the $\Lambda$ function to produce an output, the $\Delta_{int}$ function to advance to the following sequential state, and the $TA$ to compute the new $t_{next}$. The output is routed to the influenced engines using the $Z^{IA}$ and the influencee's *x-function*.

In this example, the **Coordinator** creates four similar simulation branches. Table 2 shows the variables of the branch when *G1* is chosen to be executed first. There is no conflict in the changes to the variables, as the Processor queued a message that will be output after all generators advance.

For each branch, the process is repeated three more times where the other generators are also queuing and advancing similarly to the first one. After four simulation steps, we have 24 branches, each with a different permutation of inputs to the Processor. In Table 3, we show the variables of the branch were generators advanced in order $G1$, $G2$, $G3$, $G4$.

Afterward, the $t_{next}$ variable in the **Root Coordinator** is set to $[1{,}247, 1{,}255]ms$. This matches the time to advance the simulation on the Processor. Then, the **Root Coordinator** calls the **-function* of the coupled model, which forwards the call to the Processor, which, being the only imminent model, has no special restriction applied by limit.

The **-function* on the Processor **Simulator** first applies the $\Lambda$ function and obtains the output set. The output set will output the first process that was queued, which depends on the current branch history. In the case of executing models in the order $G1$, $G2$, $G3$, $G4$, $P$, the output is $[1, 1]$. Then, a new state is obtained using the $\Delta_{int}$ function. The new state sets the TOCJ component to 0 and dequeues the first job in $QJ$ obtaining $[\langle 0, \langle 2, 3, 4 \rangle \rangle, \langle 0, \langle 2, 3, 4 \rangle \rangle]$. Finally, $t_{last}$ is assigned the current time $[1{,}247, 1{,}255]ms$ and $t_{next}$ is computed using the $TA$ function and set to $[1{,}497, 1{,}505]ms$.

Finally, the output interval obtained is returned to the **Coordinator**.

Back in the **Coordinator**, the value is routed using the $Z^{IA}$ functions. In this case, the output is routed to as a coupled model output.

Table 4. **Simulators** Variables After Seven Steps in G1-G2-G3-G4-P-P-P Branch

| | G1 | G2 | G3 | G4 | P |
|---|---|---|---|---|---|
| $state$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[\langle 0, \langle 4 \rangle\rangle, \langle 0, \langle 4 \rangle\rangle]$ |
| $t_{last}$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[1{,}747, 1{,}755]ms$ |
| $t_{next}$ | $[1{,}994, 2{,}010]ms$ | $[1{,}994, 2{,}010]ms$ | $[1{,}994, 2{,}010]ms$ | $[1{,}994, 2{,}010]ms$ | $[1{,}997, 2{,}005]ms$ |

Table 5. **Simulators** Variables After Eight Steps in G1-G2-G3-G4-P-P-P-Nothing Branch

| | G1 | G2 | G3 | G4 | P |
|---|---|---|---|---|---|
| $state$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[0, 0]$ | $[\langle 250, \langle 4 \rangle\rangle, \langle 250, \langle 4 \rangle\rangle]$ |
| $t_{last}$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[997, 1{,}005]ms$ | $[1{,}747, 1{,}755]ms$ |
| $t_{next}$ | $[1{,}997, 2{,}010]ms$ | $[1{,}997, 2{,}010]ms$ | $[1{,}997, 2{,}010]ms$ | $[1{,}997, 2{,}010]ms$ | $[1{,}997, 2{,}005]ms$ |

The $t_{next}$ is set to the value obtained in the Processor for both the **Coordinator** and the **Root Coordinator**, and the next two steps of simulation advance in an analogous way to obtain the variables shown in Table 4.

At this point, all branches reach a similar conflicting state, where the Processor is expected to output the last previously queued job at $[1{,}997, 2{,}005]ms$, and all the generators are trying to queue new jobs at $[1{,}994, 2{,}010]ms$. This is the first conflict that will need slicing the intervals using the $limit$ variable to cover all the possible scenarios.

We start the iteration at the **Root Coordinator** with a $t_{current}$ of $[1{,}994, 2{,}005]ms$, which corresponds to the start of the generators' $t_{next}$ interval and the end of the Processor's $t_{next}$ interval obtained by *BoundTs* in the previous simulation step. The **-function* of the **Coordinator** is called and the $limit$ is set to $[1{,}994, 1{,}997)ms$. Then, the simulation is branched into five scenarios, one per Generator that could be advanced plus one for the case none of them advance. In total, we have now 120 branches being explored. However, many of the branches have equal values in all the variables and could be merged to avoid recalculations. For example, from our 120 branches, only 20 are different.

In all branches' next step of the simulation, the $t_{next}$ of the **Root Coordinator** matches the $[1{,}997, 2{,}005]ms$ from the Processor's **Simulator**. In particular, the case in which none of the models advanced its variables is assigned as showed in Table 5.

Here, the next step produces a conflict between the five simulators. The $limit$ is set to $[1{,}997, 2{,}005]ms$ matching the Processor's $t_{next}$. Five new branches are created, each advancing a different **Simulator**. In the case a Generator advances, the next step will have the limit coming from the Processor again. In the case the Processor advances, the next step will have a conflict between the four generators like the one at the start of the simulation with the difference that the $t_{next}$ interval will be smaller ($[1{,}997, 2{,}010]ms$) than the initial one ($[1{,}995, 2{,}010]ms$).

### 7.4 Results Discussion

By following different branches, we can observe some characteristics from the possible behaviors of the model. First, we can see that the four Generators will queue a job before any gets processed by the Processor. The order could be permuted, but no action from the Processor happens before the four of them are queued. Second, after three jobs are processed multiple things could occur depending on if the Processor processes the fourth job before getting a request for a new one or not. In the case the fourth job is processed before the next request, the queue gets empty. In the case a new job is queued before the fourth gets processed, we will have a queue of up to five jobs when generators send the second wave of jobs.

Analyzing the models and the simulation steps, we can see that the maximum size of queued jobs is a linear function of the number of job waves received, with a minimum of 0 jobs. The Simulator produces results consistent with these limits and no possibility is left uncovered by our algorithms. Each different possibility is explored in a different branch of simulation which is reflected as a set of branches in the resulting trajectories tree.

This example uses a model simple enough to be able to reason about it and validate its results. However, the method works for arbitrary complex systems composed of large networks of interconnected models.

## 8 CONCLUSIONS

We discussed the current limitations of uncertainty propagation in Discrete-Event Systems models and proposed a formal generic specification for DES with uncertainty components named UA-DEVS inspired in DEVS. The UA-DEVS formalism captures uncertainty generation and propagation by replacing the perfectly accurate events of DEVS with events defined by sets of messages and sets of occurrences.

Advancing a single step of simulation of UA-DEVS models evaluating all combinations of states and event occurrence times could be done by finite means in trivial cases only. In general, infinite computer power is needed for processing a single simulation step. Thus, we define a second formalism for approximating UA-DEVS models in a way that they could be simulated in computers. This new formalism, IA-DEVS, approximates the sets of UA-DEVS models by intervals containing them. To achieve this, the model needs the definition of several functions not relevant to the conceptual model itself, but mandatory for its computable implementation. We introduce order and bounding functions to make the variables able to represent their values by intervals. We introduce new output transformation approximations to map between representations used by different sub-models when writing an approximation of coupled models.

We presented the algorithms to simulate the approximated models with uncertainty propagation; these algorithms guarantee that the simulation of a single step could be simulated when the model functions are computable. In addition, the algorithms guarantee that the set of generated trajectories includes all trajectories expected from the original model being approximated.

We presented a method for defining an IA-DEVS model starting from a UA-DEVS model and an example of how to capture a DEVS model as a UA-DEVS one.

In the future we propose as follows:

- Studying pruning mechanisms to remove trajectories that are not coming from the original UA-DEVS model after producing a superset of them by approximating to IA-DEVS.
- Pruning to remove repeated branches producing the same set of trajectories.
- Exploring the use of IA-DEVS as an approximation of other formalisms. For instance, to approximate DEVS models when simulated using approximated variables as Floating Point whose problems we discussed in our previous research [14].
- Simulating interaction between different variants of the BitTorrent protocol in scale (swarms larger than 20k peers). This is the original problem that motivated our research project, when connecting the thousands of peers, we need to model a fully distributed system over unknown network topology (Internet) with some well-known constraints. We would also like to model asynchronous circuits and evaluate architectures for future processors.
- Studying applicability of our ideas to use PDEVS as base formalism in place of DEVS. Confluence function and bags of results introduce new challenges for the branching mechanisms.
- Defining the use of ports for input and output sets.

## REFERENCES


[1] BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, and OIML. 2008. *Evaluation of Measurement Data—Guide to the Expression of Uncertainty in Measurement*, Joint Committee for Guides in Metrology.

[2] BIPM, IEC, IFCC, ILAC, ISO, IUPAC, IUPAP, and OIML. 2012. *International Vocabulary of Metrology—Basic and General Concepts and Associated Terms (VIM)* (3rd ed.). Joint Committee for Guides in Metrology.

[3] Alex C. H. Chow and Bernard P. Zeigler. 1994. Parallel DEVS: A parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Conference on Winter Simulation*. Society for Computer Simulation International, 716–722.

[4] Richard M. Fujimoto. 1999. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 46–53.

[5] Valerio Gheri, Giovanni Castellari, and Francesco Quaglia. 2008. Controlling bias in optimistic simulations with space uncertain events. In *12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. IEEE, 157–164.

[6] Moon Ho Hwang and Bernard P. Zeigler. 2006. A modular verification framework based on finite & deterministic DEVS. In *Proceedings of 2006 Spring Simulation Multi-conference: DEVS Symposium*. Society for Computer Simulation, 57–65.

[7] Thomas Jech. 2003. *Set Theory*. Springer, Berlin.

[8] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (1978), 558–565.

[9] Baoding Liu. 2010. *Uncertainty Theory: A Branch of Mathematics for Modeling Human Uncertainty*. Springer, Berlin.

[10] Margaret L. Loper and Richard M. Fujimoto. 2000. Pre-sampling as an approach for exploiting temporal uncertainty. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*. IEEE Computer Society, 157–164.

[11] Francesco Quaglia and Roberto Beraldi. 2004. Space uncertain simulation events: Some concepts and an application to optimistic synchronization. In *18th Workshop on Parallel and Distributed Simulation*. IEEE, 181–188.

[12] Hesham Saadawi and Gabriel A. Wainer. 2010. Rational time-advance DEVS (RTA-DEVS). In *Proceedings of the 2010 Spring Simulation Multiconference*. Society for Computer Simulation International, 143–151.

[13] Isabel Taverniers, Marc De Loose, and Erik Van Bockstaele. 2004. Trends in quality in the analytical laboratory. II. Analytical method validation and quality assurance. *TrAC Trends in Analytical Chemistry* 23, 8 (2004), 535–552. DOI: https://doi.org/10.1016/j.trac.2004.04.001

[14] Damian Vicino. 2015. *Improved Time Representation in Discrete-Event Simulation*. Ph.D. Dissertation. Université Nice Sophia Antipolis, Carleton University (CA).

[15] Damian Vicino, Olivier Dalle, and Gabriel A. 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*. 89–98.

[16] Damian Vicino, Daniella Nayunkuru, Gabriel A. Wainer, and Olivier Dalle. 2015. Sequential PDEVS architecture. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. Society for Computer Simulation International, 165–172.

[17] Damian Vicino, Gabriel A. Wainer, and Olivier Dalle. 2017. An abstract discrete-event simulator considering input with uncertainty. In *Proceedings of the Symposium on Theory of Modeling & Simulation*. Society for Computer Simulation International, 1–12.

[18] Gabriel A. Wainer. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. CRC Press, Boca Raton, FL.

[19] Thomas A. Zang, Michael J. Hemsch, Mark W. Hilburger, Sean P. Kenny, James M. Luckring, Peiman Maghami, Sharon L. Padula, and Jefferson W. Stroud. 2002. *Needs and Opportunities for Uncertainty-Based Multidisciplinary Design Methods for Aerospace Vehicles*. Technical Report. National Aeronautics and Space Administration, Langley Research Center.

[20] Bernard P. Zeigler, Alexandre Muzy, and Ernesto Kofman. 2018. *Theory of Modeling and Simulation: Discrete Event & Iterative System Computational Foundations*. Academic press.


Received December 2019; revised March 2021; accepted May 2021