Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.

SPECIFYING AND EXECUTING THE COMBINATION OF TIMED FINITE STATE AUTOMATA AND CAUSAL-BLOCK DIAGRAMS BY MAPPING ONTO DEVS

Randy Paredis
Joachim Denil
Hans Vangheluwe

University of Antwerp – Flanders Make
Middelheimlaan 1, Antwerp, BELGIUM

ABSTRACT

Multi-Paradigm Modeling (MPM) advocates to explicitly model every part and aspect of a system, at the most appropriate level(s) of abstraction, using most appropriate formalism(s). We show, starting from a representative Personalized Rapid Transportation rail car example, how MPM naturally leads to the need to combine formalisms. To give these formalisms a precise semantics and to make them executable, we choose to map them all onto behaviorally equivalent (modulo some level of approximation in the case of continuous formalisms) Discrete EVent system Specification (DEVS) models. Our focus and main contribution is the principled combination TFSA>(CBD+StEL) of Timed Finite State Automata (TFSA) and Causal Block Diagrams (CBDs) using a State Event Location "glue" formalism StEL, and their mapping onto DEVS. The result of our principled workflow, explicitly modeled in a Formalism Transformation Graph + Process Model (FTG+PM) is an accurate and efficient simulator. This is demonstrated on the rail car case.

1 INTRODUCTION AND EXAMPLE PROBLEM

To set the stage for our contribution, we consider the behaviour of a simplified Personalized Rapid Transportation (PRT) (Buchanan et al. 2005) *rail car*, traveling along a single track from an origin to a destination station.

The Environment in which the rail car System under Study (SuS) operates consists of passengers arriving at the origin station with an Inter-Arrival Time (IAT) uniformly distributed in $[0, IAT_{MAX}(=10)]s$. Passengers can board one by one (as the car doors are narrow) as long as the car has not yet departed. Boarding takes 5 seconds per passenger. If either the car has departed or other passengers are waiting to get on or are getting on the train, passengers have to queue inside the station.

The car has a capacity of $MAX(=10)$ passengers. Once all passengers have boarded, the origin station is notified (so no more passengers are allowed to board) and the car starts up during 5 seconds. This includes, *i.e.*, powering up the engine, making departure announcements, and closing the doors. The car then departs for its destination. A bang-bang controller (also known as an on-off controller) regulates the car's velocity v. Initially in the *on* mode, the controller instructs the car's engine to accelerate until the car reaches the maximum allowed speed $v_{max}(=30\ km/h)$. The controller then switches to *off* mode, letting the car coast. Due to air resistance and wheel friction, the rail car will decelerate. When the speed drops below $v_{min}(=24\ km/h)$, the controller switches back to the *on* mode. The alternation between *on* (accelerating) and *off* (decelerating) modes continues until the car reaches position $x_{stop}(=2.8\ km)$. It then starts braking to come to a standstill at the destination station. Once arrived, passengers leave the car one by one, every 5 seconds.

We wish to model and simulate the behaviour of the rail car: the position x, velocity v of the car as well as the number of *passengers* on board, over time. The upper part of Figure 1 shows the velocity v (solid red line, left axis) and the number of *passengers* (blue dot-dashed line, right axis). The lower part

978-1-6654-3311-2/21/$31.00 ©2021 IEEE

of that same figure traces the mode changes of the car's movement (left axis) and of its passengers (right axis).

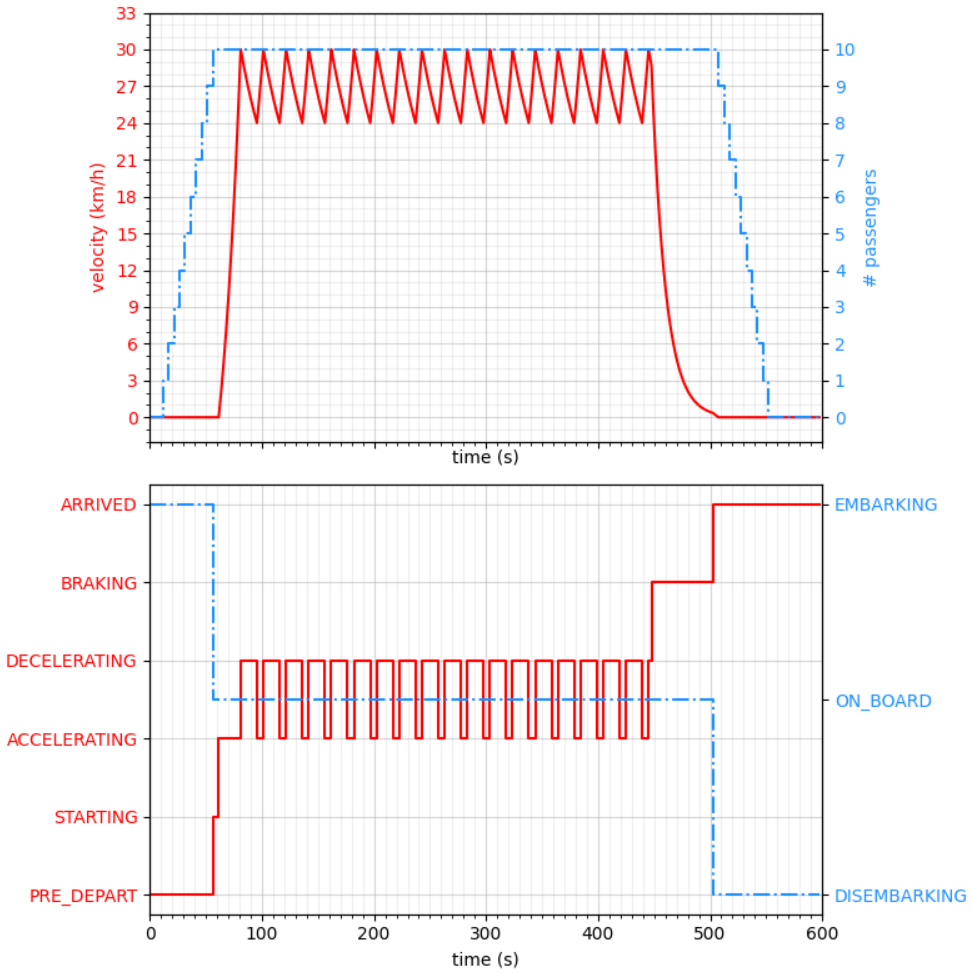

Figure 1: Rail car dynamics.

Starting from the above specification, we now want to model the system. Following *Multi-Paradigm Modeling* (MPM) (Mosterman and Vangheluwe 2004) principles, each part and aspect of the system's structure and behaviour should be modeled using at the most appropriate level(s) of abstraction, using the most appropriate formalism(s) (also known as modeling language(s)). As is often the case, this will lead to a multi-formalism model.

In the following, the (sub-)models and their formalisms will be informally introduced. In section 3, a precise semantics will be given to these formalisms as well as to their combinations by mapping onto the *Discrete EVent system Specification* (DEVS) (Zeigler et al. 2000) formalism.

Figure 2 shows a hierarchical model of the rail car system and its environment following its earlier informal description. The boxes represent model instances. The notation "modelName:Formalism" (in the top left of the boxes) is used to specify the formalism the model is specified in. At the top level of the hierarchy, the causal *Network* formalism is used to describe the connectivity of individual components Boxes have *input* (black triangles) and *output ports* (white triangles) and the directed connections between them denote interaction/communication "channels". The total state set of a network is the cross product of all total state sets of each of its components.

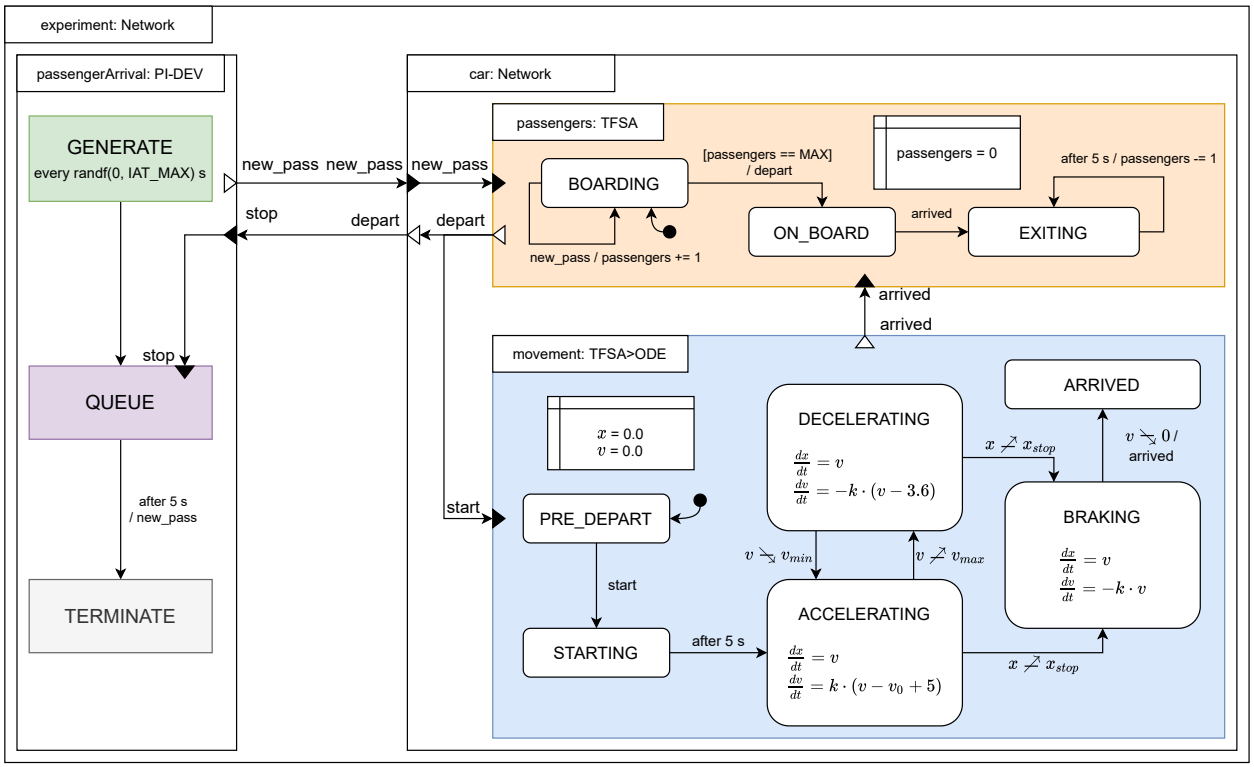

Figure 2: Passenger arrival and rail car dynamics multi-formalism model.

The *arrival of the passengers* on the platform in the origin station is most naturally modeled using a *Process Interaction Discrete Event* (PI-DEV) World View modeling language such as GPSS (Overstreet and Nance 2004).

The rail car's number of passengers and its movement are mostly independent. We modularly separate these *concerns*, each using a most appropriate formalism.

The evolving number of *passengers* in the rail car can be elegantly described using a *Timed Finite State Automaton* (TFSA). A TFSA consists of multiple distinct states or modes (represented by roundangles in this paper) and transitions (arrowed connections) between them. A transition is labeled with three (optional) parts: a *guard* (between square brackets, denoting a condition that must be satisfied), an *input* (the event that triggers the state change) and an *output* (output events to broadcast or action code to execute when this transition is taken). The input event is either a normal event (when it's a word), a delayed event (when using "*after*"), or a crossing through a state variable that requires state-event location (when either $\nearrow$ or $\searrow$ is present). passengers:TFSA starts in BOARDING mode with the number of passengers initialized to 0. Each time a new_pass event is received from the arrival process, passengers is increased, until MAX is reached. This brings passengers:TFSA in the ON_BOARD mode. When the train arrives at its destination, the *movement* sub-model sends an arrived event. This causes a transition to the EXITING mode during which passengers is decremented every 5 seconds until no passengers are left.

The *movement* of the car is best expressed in a *hybrid* (discrete-continuous) formalism (Nilsson et al. 2003). Hybrid formalisms are typically used, either because not enough details of the physics of a system are known to construct a fully continuous model, and time-scale and/or parameter abstractions need to be used, or to increase simulation performance. The continuous behaviour of the rail car is naturally described using a set of Ordinary Differential Equations (ODEs) obtained from Newton's Second Law of Motion. The continuous behaviour changes depending on which of three distinct movement modes the car is in: ACCELERATING, DECELERATING, or BRAKING. For the remaining states PRE_DEPART,

STARTING, and ARRIVED, we refer to the earlier description of the movement. The discrete changes between these modes can again be described in the TFSA formalism. In each of these modes, an ODE model is *embedded* describing the laws governing the movement of the car while in that mode. The resulting *hybrid* formalism is TFSA>ODE. The ">" identifies the direction of embedding: ODE is embedded in TFSA. The trigger to transition from one continuous mode to another is done by monitoring (some function of) the variables in that continuous mode. *When* such a monitored variable crosses a threshold in a particular direction (*e.g.*, from below), this generates a *state event*. This is the trigger for the mode transition. Such monitors are described in a State Event Location (StEL) formalism. The *hybrid* formalism is thus really TFSA>(ODE+StEL). When entering a new mode, the ODE embedded in that mode needs to be (re-)initialized. This is done based on the values of the variables in the old mode at the time of the state event. The behaviour traces of such a hybrid model are *piecewise continuous*.

Note that in our Network models, we do not consider direct connections between ODE sub-models. This leads to a *co-simulation* (Gomes et al. 2018; Gomes et al. 2019) problem for which solutions exist.

In the rest of this paper, Section 2 introduces the DEVS (Discrete-EVent systems Specification) and CBD (Causal Block Diagram) formalisms used as building blocks later on. Section 3 subsequently describes a workflow combining different model transformations to transform hybrid models in the TFSA>(ODE+StEL) formalism to a single hierarchical DEVS model. Section 4 presents some related work. Section 5 concludes the paper and outlines avenues of future work.

2 BACKGROUND

This section briefly introduces the two main formalisms Causal-Block Diagrams (CBDs) and DEVS used later on to construct the hybrid formalism TFSA>(CT-CBD+StEL).

2.1 DEVS

DEVS (Zeigler et al. 2000) is a modular discrete-event formalism introduced by Bernard Zeigler in the '70s. It consists of basic components, called atomic DEVS, which have the following structure: $\langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$. Here, the *input set* X denotes the set of admissible input events of the model and *output set* Y denotes the same for the output events. When properly structured, X and Y describe a collection of *ports* through which a model sends or receives events. S is the *state set* and indicates the set of sequential states of the model. The *internal transition function* $\delta_{int} : S \to S$ specifies the state the system transitions to after ta time, unless when interrupted before. The *time advance function* $ta : S \to \mathbb{R}^{+}_{0,+\infty}$ specifies how long the system remains in a state, before δ_{int} takes it to the next sequential state. Prior to the application of δ_{int}, the *output function* $\lambda : S \to Y$ is called, specifying the output event that is to be produced. The *external transition function* $\delta_{ext} : Q \times X \to S$, where $Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$ is called when an *external input* (*i.e.*, an interrupt) arrives.

Multiple DEVS models can be combined into a network structure. Such a coupled DEVS is fully characterized by $\Delta = \langle X_\Delta, Y_\Delta D, M_i, I_i, Z_{i,j}, select \rangle$. Similar to atomic DEVS, X_Δ and Y_Δ denote the in- and output sets of the network. The *set of component references* is denoted by D, where M_i identifies the DEVS model component $i, \forall i \in D$. I_i is the *set of influencees* of component $i, \forall i \in D \cup \{\Delta\}$ (with $i \notin I_i$). I fully specifies the connection topology of a coupled model. The *transfer function* $Z_{i,j}$ is defined $\forall i \in D \cup \{\Delta\}, j \in I_i$ as $Z_{\Delta,j}$: $X_\Delta \to X_j$; $Z_{i,\Delta}$: $Y_i \to Y_\Delta$; and $Z_{i,j}$: $Y_i \to X_j$. It serves to translate events as they travel along network connections. Finally, $select : 2^D \to D$ is the *select function* that allows the selection of a single component $i \in D$ multiple models in the network are simultaneously *imminent* (*i.e.*, they are due to transition at the same time). As DEVS is closed under coupling, coupled DEVS models may be nested to any arbitrary depth.

2.2 Causal-Block Diagrams (CBDs)

A system that consists of multiple interacting components is naturally modelled using "boxes and arrows", which may be hierarchically composed. This is a common notation used in many different domains, including schematic system overviews and electrical circuit modeling (Åström et al. 1998). Causal-Block Diagrams (CBDs) (Gomes et al. 2020) is a formalism which specializes the box-and-arrow notation. The arrows denote signals and the boxes represent mathematical operations over the input signals, producing an output signal. The denotational semantics of a CBD with continuous time base ($\mathbb{R}$) is the corresponding set of Algebraic, Ordinary Differential, or Differential Algebraic Equations (and ultimately, the signals, continuous functions of time, that satisfy the constraints imposed by these equations). As such, ODE and CBDs can be transformed into each other without loss of information. Time-discretization leads to an approximation, but does allow for iterative simulation. To simulate a discretized CBD model, two nested loops are used. In the *outer loop*, a stepping variable k is started at 0 and increased for every iteration of the loop. In-between k and $k+1$, a time-delay $h_k \in \mathbb{R}$ is used to advance the simulated time. The simulation finished when some termination condition (a function of current simulation time $\sum_{i=0}^{k} k_i$ and/or state variables) becomes true. When h_k is independent of k, the simulation is said to have a *fixed step size*. The smaller h_k, the better the numerical simulation results will approximate the continuous solution. *Adaptive step size* algorithms vary h_k throughout the simulation, to keep the stepwize approximation error within given bounds. A commonly used adaptive step size discretization is the fourth-order *Runge-Kutta Fehlberg* method (RKF45). The difference between the fourth and fifth order approximations is used as an estimate for the stepwize error.

The *inner loop* focuses on the computation of an output signal for a given iteration k. Here, a schedule is defined, determining the order in which the blocks of the model will be traversed. This schedule is typically based on the *topological order* of the *dependency graph* of the model, making sure a block is only visited once all its dependencies have been computed. Each block has their own definition of what this "computation" may entail. For instance, the adder block will output the sum of all its inputs.

An *algebraic loop* occurs when there is a cycle in the dependency graph. The *strong components*, *i.e.*, all blocks belonging the algebraic loop, should therefore be computed as a whole. It's possible to distinguish two methods for solving a strong component:

1. The blocks are converted to a system of equations, such that a (non-)linear solver is able to find a solution for the system. For instance, when these equations are all linear, the *Gauss-Jordan Elimination* (GJE) algorithm can be used.
2. *Tearing* the algebraic loop. Some connections in the strong component are broken and replaced by initial guesses for the values that are supposed to be signaled over the connection. When this torn loop has found a solution, the initial guesses are replaced by this solution and the loop is computed again. This small algorithm is executed until convergence.

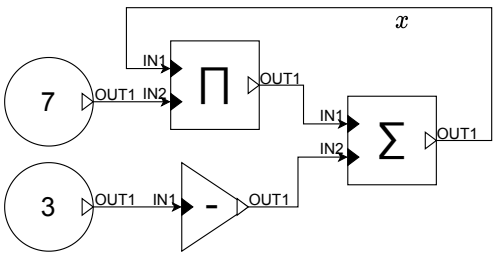

Figure 3: CBD model with algebraic loop.

In Figure 3, a simple CBD model is shown, representing the algebraic equation $x = 7x - 3$. See Table 1 for an overview of the used blocks and their meanings. The inner loop will compute 7, 3 and the negation of 3 ($= -3$), before arriving at the algebraic loop containing the Product block and the Adder block. As the equation is linear, GJE can be used to find $x = 0.5$.

Continuous-time blocks Integrator and Derivative will be symbolically expanded based on some discretization schema when they appear in a CBD model. If the step size is fixed, these blocks can be replaced by a hierarchical block, containing Delay and algebraic blocks. The Delay block has two inputs (IC and IN1) and a single output (OUT1). For iteration k, the

Table 1: CBD block syntax and meaning for the blocks used in this paper. Italic text identifies the shape's description.

Symbol / Description	Name	Meaning
value in circle	Constant	Outputs a constant value
small black triangle	Input	Takes an external input value
small white triangle	Output	Produces an external output value
$-$ (*triangle*)	Negator	Negates the input
Σ	Sum	Adds both input values
$\prod$	Product	Multiplies both input values
$\int$ (*triangle with box*)	Integrator	Computes the integral of the input signal over time
clock	Delta	Outputs the h_k value

signal on OUT1 will however always be the signal that arrived on IN1 in iteration $k-1$. Only when $k=0$, OUT1 will output the obtained input from IC. It can be assumed that computations from past iterations are known in future iterations, giving the Delay block the unique property of being able to "break" an algebraic loop by partially delaying it until the next iteration.

When the model uses an adaptive step size, specifically RKF45, the original model will be transformed. The set of blocks required to obtain the integration/derivation result(s) will be extracted as a custom function block f, which is to be used and computed at multiple positions in the transformed model. Figure 4 shows the symbolic expansion for the Integrator block using a fixed step size backwards difference discretization.

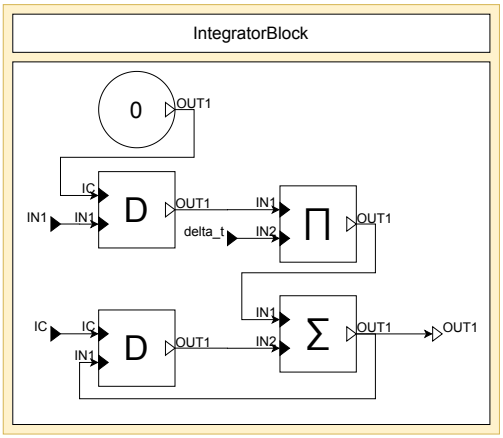

Figure 4: Symbolic fixed step size approximation expansion for the Integrator block.

3 FORMALISM TRANSFORMATIONS AND A HYBRID FORMALISM

As discussed in (Vangheluwe 2000), the DEVS formalism is a "common denominator" onto which all discrete-time and discrete-event formalisms can be mapped. If a discretization or quantization step is introduced, continuous formalisms can also be mapped onto DEVS, albeit resulting in an *approximation* of the mathematical semantics. The following will present translations onto DEVS for each of the modeling formalisms used in Figure 2. This gives a precise and executable (by means of a DEVS simulator) semantics to these informally introduced formalisms.

TFSA to DEVS A Timed Finite State Automaton (TFSA) adds time to Finite State Automata. The mapping onto an atomic DEVS is straightforward. TFSA time delays are encoded in the DEVS time advance function. The timed transitions are translated to the DEVS internal transition function and incoming events are translated to the DEVS external transition function.

PI-DEV to DEVS (Paredis et al. 2020) presented a translation procedure GPSS2DEVS from models in the process interaction discrete-event (PI-DEV) language GPSS to a behaviorally equivalent model in DEVS. As the passengerArrival process is very simple, and as the focus of this paper is the hybrid language combining TFSA and CBD, we avoid the overhead of translation and directly build a coupled DEVS model composed of a *generator* and a *queue* atomic DEVS model.

Hybrid Formalisms For hybrid formalisms, to fully benefit from the simplicity of the previously mentioned translations, a translation workflow is provided. Hence, in Figure 5, a Formalism Transformation Graph and Process Model (FTG+PM) (Mustafiz et al. 2012) for the construction of a hybrid DEVS model from a system description is shown. Manual activities are annotated with a "person" icon. All other activities are automated. The *System Modeling* activity uses the given *requirements* to obtain a TFSA>ODE. This

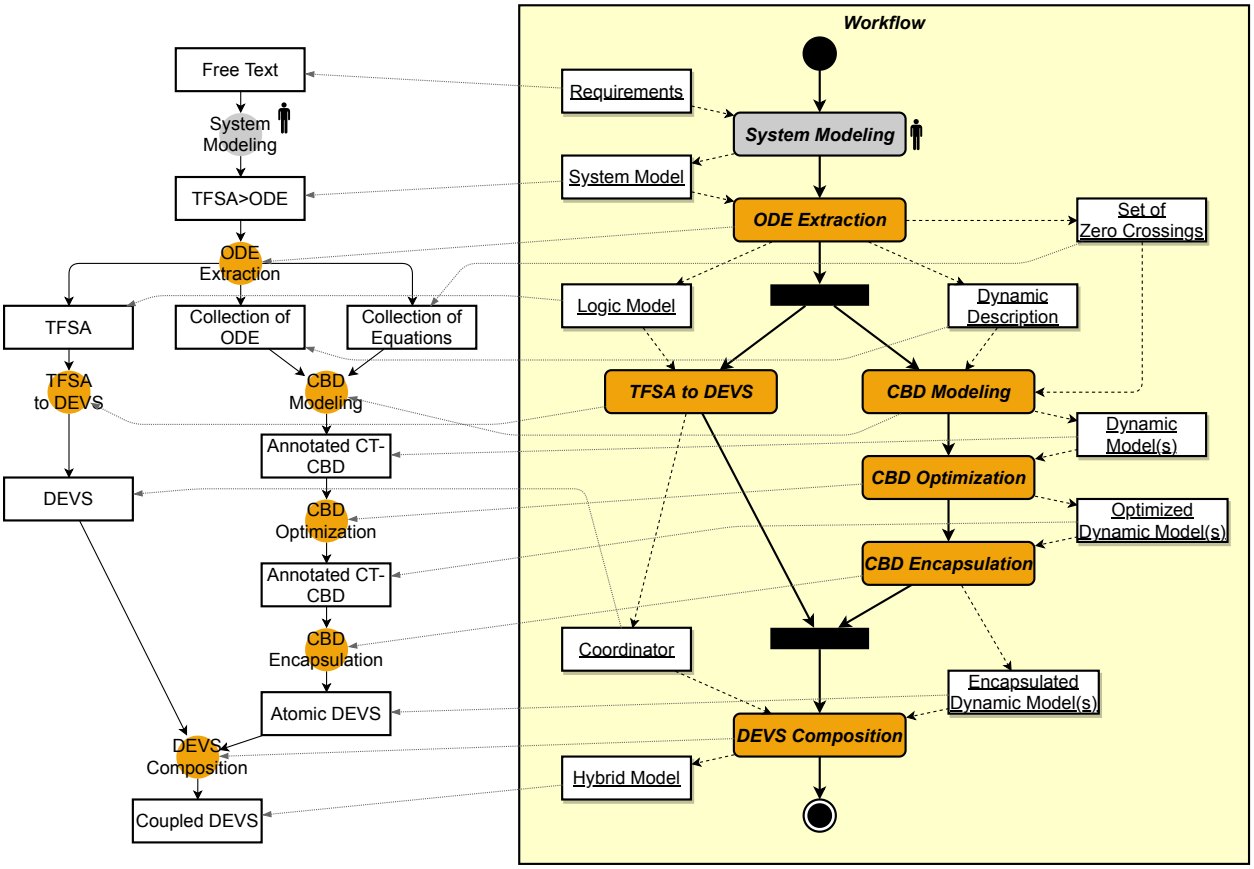

Figure 5: FTG+PM for constructing the DEVS model.

model should fully describe the required components of the system (aspect(s)) under study. In the *ODE Extraction* activity, all ODE states will be converted into empty states, storing the set of equations of the ODE state in an external, yet linked component. Additionally, the StEL description will also be extracted, allowing it to be converted to an event. What remains is a normal TFSA, with explicit links to external ODEs and the corresponding StEL units. This TFSA can easily be translated to an atomic DEVS model, yielding a *coordinator* model. The *CBD Modeling* activity converts all external ODE components to CBD models, which can be optimized and/or simplified by the *CBD Optimization* activity. These optimizations can (but don't have to) include: *RK preprocessing* (converts a fixed step size CBD model into an adaptive step size model, using Runge-Kutta), *constant folding*, *peephole optimizations*...

The *CBD Encapsulation* activity does not *translate* the CBD model to a semantically equivalent DEVS model, but rather *embeds* a CBD simulator in an atomic DEVS model, ensuring the StEL events can be fired. This embedding allows debugging and traceability of the underlying CBD model, as well as *state event location*.

The resulting DEVS models are combined in the *DEVS Composition* activity, as graphically illustrated in Figure 6. Overlined (port) notations identify a (possibly empty) collection of signals/events. The embedded models continuously output the set of computed variables $\overline{v}$ and send them to the coordinator.

In the TFSA>(ODE+StEL), a specific set of equations was only required to be executed in a specific state. Hence, based on the internal state of the coordinator, a specific subset of all encapsulated CBD models may be (re)started, while all others must be stopped.

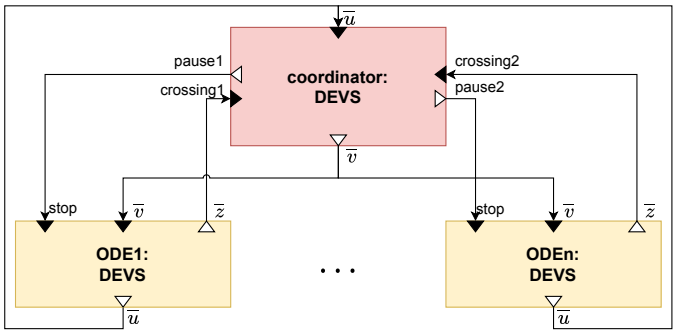

Figure 6: Generic structure of a DEVS model obtained from a hybrid TFSA>(ODE+StEL) model.

3.1 Embedding CBDs In DEVS

Instead of extracting the ODE from the CBD and translating it to DEVS (which is perfectly possible due to the relationship between CBD and ODE), the choice was made to *embed* (or *encapsulate*) the CBD model and its corresponding simulator inside a DEVS model. While computationally less efficient, the internal block structure is maintained and can be used for debugging purposes and traceability. Therefore, this section describes how a CBD *simulator* can be embedded in an atomic DEVS model w.r.t. the original CBD model.

The generic structure of a CBD block, compared to a DEVS model is mostly similar. Both have a set of in- and output ports and an internal state. For CBDs, this state is at least the current iteration index k, (a copy of) the original CBD model (executed up to k) and the corresponding simulation unit.

For CBDs, all iterations k take a certain time h_k to execute. Because of the high correlation with DEVS' ta function, it is useful to ensure ta matches h_k. However, every iteration can be split into three stages. The *waiting stage* introduces the required delay h_k into the system, whereas the *computation stage* does the actual computation of the CBD simulation at iteration k. Here it becomes clear the CBD simulator will be embedded as a whole in a single atomic DEVS. Finally, there is the *output stage* that has to follow the *computation stage*, because it outputs the computed values. Within CBDs, this separation is not entirely necessary, however the DEVS semantics dictate λ to happen before δ_{int}. Hence, the encompassing atomic DEVS needs to sequentially iterate through these stages for every iteration. Therefore, ta becomes 0 for the *computation* and the *output stage*, but it equals h_k in the *waiting stage*. This stage information will also be added to the state of the encapsulating DEVS model.

A major difference between CBD and DEVS is that CBD makes use of *signals*, but DEVS uses *events*. Signals exist in the continuous world and are indicative of a data flow. Events, however, are a discrete notification to another unit of the system. While in a different world, the *zero-order hold* principle allows for a semantic adaptation between the discrete world and the continuous world. This makes the logic for the outputs Y simple: whenever the DEVS model is in the *output stage*, Y equals the last computation of the output signals of the CBD model. In all other stages, λ will not output anything.

The inputs X require more effort. Since the h_k will mostly be hidden for external DEVS models, it is impossible to pinpoint whenever the encompassing DEVS model requires an input. By changing the input ports in the CBD model to Constant Blocks (*i.e.*, blocks that output a constant value), the inputs can also follow the zero-order hold logic. If an event is now received by the encompassing DEVS model, the values of these Constant Blocks will be changed to the new values.

Unfortunately, this technique does not work if the input represents an *initial condition* for the CBD model. Receiving such a value should terminate the simulation and restart it with the new initial values. Luckily, when the input is not an initial condition, the stop/restart logic can still be applied, under the assumption that the simulation clock will be given a (new) start time. Hence, when firing δ_{ext}, the CBD simulation will be terminated, all values of the Constant Blocks will be altered (w.r.t. X) and the simulation will be restarted from the current time.

To validly (re)start a CBD simulation, it is therefore pertinent for the coordinator to send a valid set of initial conditions. They can be sent to a specific CBD model, or to all CBD models at once. However, because this information is sent through multiple connections per model (one for each initial condition), it is perfectly viable to only alter a subset of the required initial conditions.

3.1.1 State Event Location

The heart and soul of hybrid modeling is *State Event Location* (or *zero-crossing detection*). A StEL of $f(\overline{u})$ through y can be written in the TFSA>(ODE+StEL) as $f(\overline{u}) \nearrow y$ or $f(\overline{u}) \searrow y$. By adding an output z for $f(\overline{u})$ to the CBD model, the encompassing DEVS model only needs to keep track of a crossing for z through y. Multiple algorithms and numerical techniques exist to obtain the exact point of crossing t_c. When this point is found, the system should rewind to t_c.

The DEVS formalism does not allow $ta < 0$, hence the problem should be solved within the CBD simulator. While h_k may be negative, this does not undo the values stored in memory blocks (*e.g.*, a delayed signal). Hence, the CBD simulator is given an *undo* operation that also rewinds all memory values. The problem has looped back onto itself: the encompassing DEVS model needs to apply the *undo* operation when the CBD model has experienced a crossing *in the past*.

The three stages mentioned before can happen in two ways:

Falling Edge indicates the order *waiting stage*, *computation stage*, *output stage*.
Rising Edge indicates the order *computation stage*, *waiting stage*, *output stage*.

While the *Falling Edge* order is the usual sequence in CBD contexts, both are semantically equivalent (due to the *output stage* being executed at the same time). When the *Rising Edge* stage order is used, the encapsulating DEVS model has information of the future signal *before* the DEVS clock has advanced. This allows the prediction of a future zero-crossing. Therefore, at time t_k, in the *waiting stage*, ta becomes $\min(h_k, t_c - t_k)$. Additionally, the internal CBD clock also needs to use $t_c - t_k$ if a crossing happened. This was done by altering the original model, such that $\min(h_k, t_c - t_k)$ is also used as to compute the new h_k.

When multiple signals $y_1, y_2, y_3 \ldots$ experience a crossing, respectively at $t_{c,1}, t_{c,2}, t_{c,3} \ldots$, a *conservative* approach can be used, *i.e.*, the smallest $t_{c,i}$-value is used as t_c. This ensures the system to raise an event for all crossings.

In Figure 6, whenever a state event z ($\in \overline{z}$) occurs, this signal will therefore also be sent to the coordinator, who will output the last obtained $\overline{v}$ as new initial conditions $\overline{u}$.

3.2 Framework

A CBD simulator (and framework) has been created in Python, based on some of the provided features available in Python(P)DEVS (Van Tendeloo and Vangheluwe 2015). It provides (soft) real-time simulation, as-fast-as possible simulation, interaction with a user-defined gameloop and coordination w.r.t. the *tkinter* mainloop. CBD models can be constructed using the www.diagrams.net graphical interface, flattened, simplified (*e.g.*, constant folding, peephole optimizations...) and converted to (and from) textual equations (*i.e.*, LaTeX, including a step-by-step solution for educational purposes). Additionally, a set of widely used building blocks is included, as well as a converter that implements the proposed embedding of CBD models in Python(P)DEVS.

3.2.1 Adaptive Step Size

To allow for full flexibility, adaptive step size will be implemented by preprocessing the original CBD model and transforming it into a semantically equivalent CBD model. Internally, the required set of formulas will be constructed. Specifically for the Runge-Kutta algorithm, it is required to evaluate a function f

at a specific time t_n and (set of) value(s) ($\bar{y}_n$). By identifying and constructing a reusable f block from the source CBD model, the function can be obtained. It computes the result of the IVP at t_n and $\bar{y}_n$.

Hence, the model will be pre-processed and converted to a structure that is shown generically in Figure 7. The *RK* block is the implementation of the Runge-Kutta formulas, for a given Butcher Tableau, using the f block. Equations 1-3 that provide the results for the *RK* block, will also be computed using CBD. Due to space constraints, their graphical representations were excluded from this paper. The preprocessed CBD model provides an q-th order approximation of the initial model and alters the CBD clock to allow for adaptive step sizes.

$$\widetilde{y_{n+1}} = \widetilde{y_n} + \sum_{i=1}^{s} b_i k_i \tag{1}$$

$$k_s = h \cdot f\left(t_n + c_s \cdot h, y_n + \sum_{i=1}^{s-1} a_{s,i} k_i\right) \tag{2}$$

$$h_{new} = h \cdot clamp\left(S \cdot \left(\frac{\varepsilon \cdot h}{|\tilde{y}_{n+1}^{(q+1)} - \tilde{y}_{n+1}^{(q)}|}\right)^{\frac{1}{q}}, 0.1, 4.0\right) \tag{3}$$

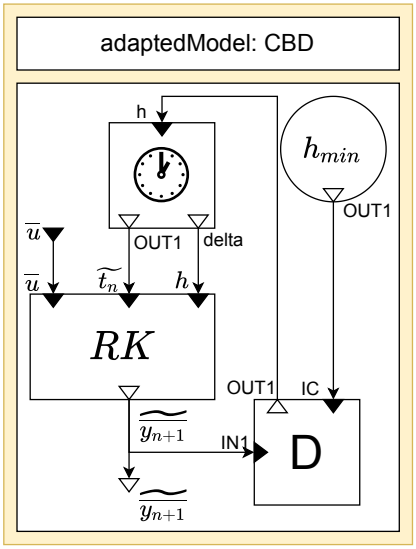

Figure 7: Generic top-level CBD model after Runge-Kutta preprocessing.

3.3 PRT Example

Returning to the PRT rail car example, the previously described techniques can be used to obtain a coupled DEVS model, shown in Figure 8. The simulation traces obtained on that model using our Python(P)DEVS simulator were shown in Figure 1.

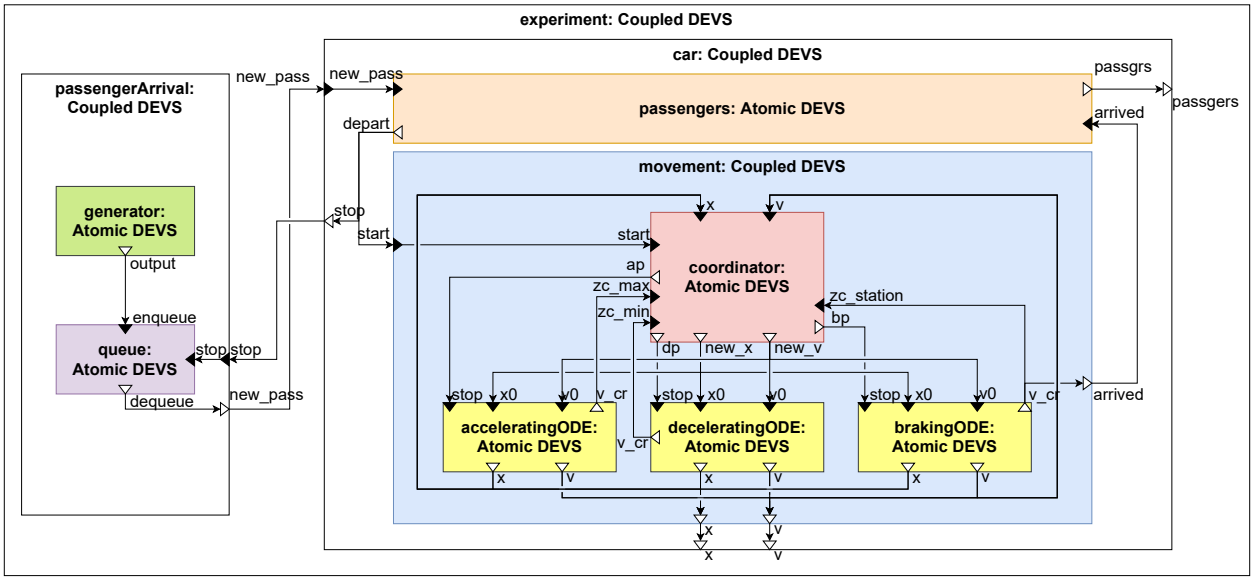

Figure 8: The resulting DEVS model. Distance detection is not shown for readability.

4 RELATED WORK

This paper applies Multi-Paradigm Modeling (Mosterman and Vangheluwe 2004), which advocates explicitly modeling all relevant parts and aspects of a system at the most appropriate level(s) of abstraction, using the most appropriate modeling language(s).

In (Borland and Vangheluwe 2003) and in (Shaikh and Vangheluwe 2011), a translation from StateCharts onto DEVS has been attempted. Similarly, in (Paredis, Van Mierlo, and Vangheluwe 2020) a mapping from GPSS onto DEVS was proposed. All these built on the same concept of DEVS being a "common denominator" for numerous modeling languages (Vangheluwe 2000), to which this paper also contributes.

PowerDEVS (Bergero and Kofman 2011) was created to support hybrid system modeling in DEVS. (D'Abreu and Wainer 2005), introduces M/CD++ for modeling and simulating continuous/hybrid systems, based on DEVS and Modelica by using quantization. A similar technique is used in (Bergero et al. 2013) in the discussion of DEVS parallelization of hybrid systems.

MECSYCO (Camus et al. 2018) is a co-simulation middleware that can be used to co-simulate CBD models using DEVS. They also use an embedding technique for mapping onto DEVS, however focusing on co-simulation and not on hybrid system modeling.

Furthermore, in (Nilsson et al. 2003), non-causal hybrid models are discussed within the context of functional modeling.

5 CONCLUSION AND FUTURE WORK

This paper has shown, starting from a representative rail car example, how Multi-Paradigm Modeling leads to the need to combine formalisms. To give these formalisms a precise semantics and to make them executable, we choose to map them all onto behaviorally equivalent (modulo some level of approximation in the case of continuous formalisms) DEVS models. Our focus and main contribution is the principled combination TFSA>(CBD+StEL) of Timed Finite State Automata and Causal Block Diagrams using a State Event Location "glue" formalism StEL and their mapping onto DEVS.

In the future, we plan to apply optimizations, in particular through symbolic manipulation of the CBD models and their discretization. In our current work, we have focused on switching between continuous models. In the future, we will combine this with interleaving, through co-simulation, of continuous models. We will also allow for the generation and inclusion of FMI 2.0-compliant Functional Mockup Units (FMUs). Finally, we plan to replace TFSA by the more expressive Statecharts formalism, also mapping the latter onto DEVS.

ACKNOWLEDGMENTS

This research was partially supported by Flanders Make, the strategic research center for the manufacturing industry.

REFERENCES

Åström, K. J., H. Elmqvist, and S. E. Mattsson. 1998. "Evolution of Continuous-Time Modeling and Simulation". In *Proceedings of the 12th European Simulation Multiconference, ESM'98*, 9–18. ESM.

Bergero, F., and E. Kofman. 2011. "PowerDEVS: a tool for hybrid system modeling and real-time simulation". *Simulation* 87(1-2):113–132.

Bergero, F., E. Kofman, and F. Cellier. 2013. "A novel parallelization technique for DEVS simulation of continuous and hybrid systems". *Simulation* 89(6):663–683.

Borland, S., and H. Vangheluwe. 2003. "Transforming Statecharts to DEVS". In *Proceedings of the 2003 Summer Simulation Conference*, 154–159. La Jolla, California, USA.

Buchanan, M., J. E. Anderson, G. Tegnér, L. Fabian, and J. Schweizer. 2005. "Emerging Personal Rapid Transit Technologies Introduction, State of the Art, Applications". In *Proceedings of the AATS conference, Bologna, Italy*, 7–8.

Camus, B., T. Paris, J. Vaubourg, Y. Presse, C. Bourjot, L. Ciarletta, and V. Chevrier. 2018. "Co-simulation of cyber-physical systems using a DEVS wrapping strategy in the MECSYCO middleware". *SIMULATION* 94(12):1099–1127.

D'Abreu, M. C., and G. A. Wainer. 2005. "M/CD++: modeling continuous systems using Modelica and DEVS". In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 229–236: Institute of Electrical and Electronics Engineers, Inc.

Gomes, C., J. Denil, and H. Vangheluwe. 2020. *Causal-Block Diagrams: A Family of Languages for Causal Modelling of Cyber-Physical Systems*, Chapter 4, 97–125. Springer International Publishing.

Gomes, C., B. Meyers, J. Denil, C. Thule, K. Lausdahl, H. Vangheluwe, and P. De Meulenaere. 2019. "Semantic adaptation for FMI co-simulation with hierarchical simulators". *Simulation* 95(3):241–269.

Gomes, C., C. Thule, D. Broman, P. Larsen, and H. Vangheluwe. 2018. "Co-Simulation: A Survey". *ACM Computing Surveys (CSUR)* 51(3):49:1–49:33.

Mosterman, P. J., and H. Vangheluwe. 2004, September. "Computer Automated Multi-Paradigm Modeling: An Introduction". *Simulation* 80(9):433–450.

Mustafiz, S., J. Denil, L. Lúcio, and H. Vangheluwe. 2012. "The FTG+PM Framework for Multi-Paradigm Modelling: an Automotive Case Study". In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, 13–18. ACM.

Nilsson, H., J. Peterson, and P. Hudak. 2003. "Functional hybrid modeling". In *International Symposium on Practical Aspects of Declarative Languages*, 376–390. Springer.

Overstreet, C. M., and R. E. Nance. 2004. "Characterizations and Relationships of World Views". In *Proceedings of the 2004 Winter Simulation Conference*, 279–287.

Paredis, R., S. Van Mierlo, and H. Vangheluwe. 2020. "Translating GPSS to DEVS (based on a DEVS Building Block Library)". In *Proceedings of the 2020 Winter Simulation Conference*, edited by K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, 2221–2232. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Shaikh, R., and H. Vangheluwe. 2011. "Transforming UML2.0 Class Diagrams and Statecharts to Atomic DEVS". In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation*, 205–212: SCS.

Van Tendeloo, Y., and H. Vangheluwe. 2015. "PythonPDEVS: a Distributed Parallel DEVS simulator". In *Proceedings of the 2015 Spring Simulation Multiconference*, 844–851. Alexandria, VA, USA.

Vangheluwe, H. 2000. "DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modelling". In *IEEE International Symposium on Computer-Aided Control System Design*, 129–134. Anchorage, AK, USA: Institute of Electrical and Electronics Engineers, Inc.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. New York: Academic Press.

AUTHOR BIOGRAPHIES

RANDY PAREDIS is a Ph.D. student in the Modeling, Simulation and Design Lab (MSDL) at the University of Antwerp. He explores a generic architecture and framework for model-based design of Digital Twins. His e-mail address is randy.paredis@uantwerpen.be.

JOACHIM DENIL is an assistant professor in the Constrained Systems lab of the Electronic Engineering department at the University of Antwerp (Belgium). He is also the core lab manager of the University of Antwerp for Flanders Make, the strategic research center for the Flemish manufacturing industry. His main research interest is modeling and simulation-based design of software-intensive and cyber-physical systems. His e-mail address is joachim.denil@uantwerpen.be.

HANS VANGHELUWE is a Professor and head of the MSDL. He develops modeling and simulation methods, techniques and tools to increase system builders' productivity. He has a long-standing interest in the DEVS formalism and is a contributor to the DEVS community of fundamental and technical research results. His e-mail address is hans.vangheluwe@uantwerpen.be.