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

MODEL TRANSFORMATION ACROSS DEVS AND EVENT GRAPH FORMALISMS

Neal DeBuhr
Hessam S. Sarjoughian

Arizona Center for Integrative Modeling & Simulation
School of Computing and Augmented Intelligence
699 S. Mill Avenue
Arizona State University, Tempe, AZ, 85281, USA

ABSTRACT

This paper develops a model transformation mechanism across the Discrete Event System Specification (DEVS) and Event Graph (EG) modeling formalisms. We detail this cross-formalism model transformation from methodological and software implementation perspectives. By using simple, well-defined, and automated mechanisms of cross-formalism model transformation, modelers establish a plurality of vantage points, from which to understand and communicate model behavior. Model characteristics may be clarified, emphasized, obfuscated, or hidden across these different vantage points. This paper, therefore, serves as a step toward research into better modeling that can improve soft factors such as model reasoning and collaborative model design for developing better simulations.

1 INTRODUCTION

There are three established formalisms for Discrete Event Simulation (DES): Colored Petri Nets (Jensen et al. 2007), Discrete Event System Specification (DEVS) (Zeigler et al. 2000), Event Graphs (EG) (Schruben 1983). Conventional simulation engineering wisdom dictates the selection of a single formalism, based on the nuances of the application domain, the availability of simulation project resources, and the desired outcomes of the simulation project. However, some problem domains, such as those in the hybrid simulation and system-of-systems engineering fields, may require the use of multiple formalisms.

Although many kinds of systems can be simulated using one modeling method, it is common to use a plurality of modeling methods. For example, practitioners develop continuous-time and discrete-time simulations for dynamical systems, when these models can help solve different kinds of problems (e.g., automated navigation of an autonomous vehicle). A single modeling method can be insufficient or otherwise impractical to achieve simulation project objectives. For Discrete Event Simulation, different modeling methods offer concrete technical benefits, such as enabling the use of a wider variety of DES simulators. However, there are also soft factor benefits. Notably, model understanding, communication, and acceptance is facilitated by having multiple vantage points by which to view a model.

In this paper, we investigate a Parallel Discrete Event System Specification (PDEVS) $\rightarrow$ Event Graph (EG) model transformation, from both a theoretical and software implementation standpoint. The PDEVS and EG formalisms are selected based on their substantial history, across both research literature and professional practice, and based on differentiation against existing related works (e.g., (Schruben and Yucesan 1994), (Redjimi and Boukelkoul 2013)). We make no formalism evaluations, of any kind. This formalism pair simply serves as a proof of concept for the broader investigation of cross-formalism model transformation. We consider the PDEVS and EG models to be at the same level of abstraction. In systems theory and DEVS, we might call this level of abstraction I/O System and atomic model, respectively. We defer the larger picture of simulators, coupled model structures, exogenous event systems, higher/lower system specification levels, higher/lower levels of abstraction, and domain-specific contextual factors to future work.

We adopt a two-step strategy for model transformation. The two-step strategy employs the Abstract State Machine (ASM) formalism (Wagner 2017), as an intermediate model representation. ASM conceptually joins the state-variable-centric event routines and strong exogenous vs. endogenous event distinction of DEVS, with the explicit future event list and conditionals-based event stratification of EG - creating a convenient formalism transformation stepping stone. Additionally, ASM provides a direct representation of the event-based simulation foundation of EG. For a conceptually simplified transformation then, we investigate DEVS-to-EG model transformation by first translating the DEVS algebraic structure to the ASM algebraic structure, and then using the ASM algebraic structure as a foundation for EG.

A single example model is used, throughout the paper, to illustrate the model transformation process. We'll call the example model "batcher". A finite capacity batching process begins when the batcher receives a job. It will then accept additional jobs, adding them to a batch with the first job until a max batching time or max batch size is reached - whichever comes first. If the simultaneous arrival of multiple jobs causes the max batch size to be exceeded, then the excess jobs will spillover into the next batch. When the batcher again has no jobs, it will become passive, and wait for a job arrival to initiate another batching.

This paper is organized as follows: Section 2 provides an overview of the relevant DES formalisms. Section 3 defines a step-by-step process for translating the DEVS algebraic structure into an ASM algebraic structure. Section 4 explains the use of ASM event rules to generate an EG model - completing the two step process of DEVS-to-EG model transformation. Section 5 details the end-to-end batcher model transformation. Section 6 summarizes an automated software implementation of the model transformation process. Finally, Section 7 provides the proof-of-concept findings and opportunities for future works. Comprehensive coverage of these topics - the general process, a concrete example, and a software implementation - would result in a long and unfriendly read. Therefore, this paper does not cover in-depth considerations like transformation generality proofs, model-simulator interactions, and edge case properties.

2 FORMALISM OVERVIEW

2.1 Discrete Event System Specification

DEVS, is the starting point for our model transformation process. DEVS has many variations and extensions. Specifically, we leverage the well-established PDEVS formalism (Zeigler et al. 2000). PDEVS atomic model definitions are an algebraic structure $< X^b_M, Y^b_M, S, \delta_{ext}, \delta_{int}, \delta_{con}, \lambda, ta >$. The element definitions from (Zeigler et al. 2000): X_M is the bag of input ports and values, Y_M is the bag of output ports and values, S is the set of sequential states, δ_{ext} is the external state transition function, δ_{int} is the internal state transition function, δ_{con} is the confluent transition function, λ is the output function, and ta is the time advance function.

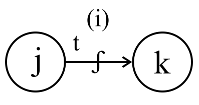
2.2 Abstract State Machine

The Abstract State Machine (Wagner 2017) defines an operational semantics for event-based and object-event transition systems. It is an algebraic structure elements: SV is the set of state variable declarations defining the structure of possible system states, ET is the set of event type definitions, and R is the set of event rules expressed in terms of SV and ET. We do not consider the nuances of ASM; instead, in alignment with the motivation described in Section 1, we use it as an intermediate model representation.

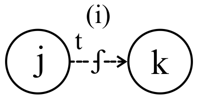
2.3 Event Graph

Our model transformation target formalism is EG (Schruben 1983). Like DEVS and Petri nets, EG has some variations and extensions. We consider the original EG formalism (Schruben 1983). It directly adopts the three basic elements of the event worldview: *state variables* that describe the system, *events* that change the values of state variables, and *relationships* between events. These elements map one-to-one to the ASM's elements of *SV* (state variables), *ET* (event type definitions), and *R* (event rules), respectively.

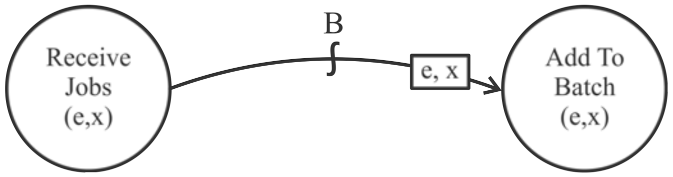
EG builds on these basic event-worldview concepts, with a visual representation founded in graph theory. EG uses vertices to represent events, and associated state variable changes are written under these vertices. Event relationships are represented as directed edges between pairs of vertices. Directed edges with solid lines represent event scheduling. Directed edges with dashed lines represent event canceling. Both types of directed edge can have conditional and temporal expressions that introduce event-scheduling conditionality and time delay. The event-scheduling edge shown in Figure 1a can be interpreted as *if condition i is met, upon execution of event j, then event k will be scheduled for execution with delay t.* The event-cancelling edge shown in Figure 1b can be interpreted as *if condition i is met, upon execution of event j, then after t time, all occurrences of event k in the future event list will be removed.*



(a) Generic event scheduling edge.



(b) Generic event cancelling edge.



(c) Batcher model edge and vertex attributes.

Figure 1: Event graph edges.

Vertex attributes and edge attributes define additional model behavior across events. Vertex attributes designate event parameters, such as *e* (elapsed time) and *x* (input value) in the events of Figure 1c. Edge attributes enable passing event parameters. In the batcher model, the same *e* and *x* values received by the Receive Jobs event are passed to the follow-up Add To Batch event (if the scheduling condition B is met).

Event cancellation edges, vertex attributes, and edge attributes are not strictly necessary for EG Turing completeness, and therefore representation of DEVS models (Savage et al. 2005). However, we leverage all three of these EG conveniences in the model transformation process, as described in Section 3.

3 ALGEBRAIC STRUCTURE TRANSLATION

By translating the PDEVS algebraic structure into an ASM algebraic structure, we create an algebraic definition that encapsulates the basic event-based simulation concepts of EG. This algebraic structure translation constitutes the first of two steps in our model transformation process.

3.1 Translation Strategies

Conditional Expressions: As a prerequisite step, we must first consider the PDEVS model function definitions. By convention, port and phase conditional expressions are often embedded in the function arguments. Each of the external transition function, internal transition function, and/or output function can have multiple functions. Considering the batcher model, We can interpret an δ_{ext} external transition functions as specifying three event rules. We can write the PDEVS functions to have conditional expressions as shown in equation 1. This approach organizes the initial model specification and makes for a simpler transformation process.

1. If the batcher is passive and the new jobs do not fill the batch, begin a new batch
2. If the batcher is already batching and the new jobs do not fill the batch, continue batching
3. Otherwise (if new job arrivals fill the batch or if a release is already triggered), release a batch

PDEVS formalism defines the external and internal state transition functions to be piecewise. For non-trivial state transitions, explicit conditionals are required for behavior changes. Nevertheless, both explicit (i.e., piecewise function cases) and implicit (e.g., $floor(s)$) conditionals can be transformed. The edge case of no explicit conditionals is also supported, and is intuitively transformed as a single conditional covering the entire model's state space and input regime. The modeler can use explicit conditionals when conditionals are substantive, and use implicit conditionals when there is no need. The choice carries through to the EG model, where explicit conditionals will generate graph structures (visually distinctive and emphasized) and implicit conditionals will be incorporated into state transition statements (a compact representation, without visual emphasis). Given the practical limitations associated with infinite graph structures, cases like $floor(s)$ should retain their implicit conditionals representation, as they would in other modeling methods (e.g., PDEVS).

$$
\begin{aligned}
X_M &= \{(p,\ v) \mid p \ \epsilon \ P_{in},\ v \ \epsilon \ X_p\} \\
Y_M &= \{(p,\ v) \mid p \ \epsilon \ P_{out},\ v \ \epsilon \ Y_p\} \\
S &= \{\text{"}passive\text{"},\ \text{"}batching\text{"},\ \text{"}release\text{"}\} \times \mathbb{R}_0^+ \times V^+ \\
\delta_{ext}(phase,\ \sigma,\ queue,\ e,\ x) &= \begin{cases} (\text{"}batching\text{"},\ t_{max},\ \{x_i \mid (\text{"}in\text{"},\ x_i) \ \epsilon \ x\}) & if\ A \\ (\text{"}batching\text{"},\ \sigma - e,\ queue \cup \{x_i \mid (\text{"}in\text{"},\ x_i) \ \epsilon \ x\}) & if\ B \\ (\text{"}release\text{"},\ 0,\ queue \cup \{x_i \mid (\text{"}in\text{"},\ x_i) \ \epsilon \ x\}) & if\ C \end{cases} \qquad (1)
\end{aligned}
$$

$$
\begin{aligned}
&where \\
&A : phase = \text{"}passive\text{"} \wedge \mid queue \cup \{x_i \mid (\text{"}in\text{"},\ x_i) \ \epsilon \ x\} \mid < queue_{max} \\
&B : phase = \text{"}batching\text{"} \wedge \mid queue \cup \{x_i \mid (\text{"}in\text{"},\ x_i) \ \epsilon \ x\} \mid < queue_{max} \\
&C : otherwise
\end{aligned}
$$

Internal Transition and Output Coupling: ASM combines event state transitions and event outputs, within a single event routine. We achieve this by joining the PDEVS internal transition and output functions: generating all combinations of the internal transition function conditional expressions and output function conditional expressions, assigning a $YIELD$ value to each combined conditional case (based on the associated output function), and assigning a state transition to each combined conditional case (based on the associated internal transition function). Considering the batcher model, this combination named is specified as δ'_{int} in equation 2. For brevity the δ_{int} and λ specifications are not provided.

$$
\delta'_{int}(phase,\ \sigma,\ queue) = \begin{cases} YIELD\ \{q_i \mid q_i \ \epsilon \ queue \wedge i \le queue_{max}\} & \\ S = (\text{"}passive\text{"},\ \infty,\ \varnothing) & if\ A \\ \\ YIELD\ \{q_i \mid q_i \ \epsilon \ queue \wedge i \le queue_{max}\} & \\ S = (\text{"}release\text{"},\ 0,\ \{q_i \mid q_i \ \epsilon \ queue \wedge i > queue_{max}\}) & if\ B \\ \\ YIELD\ \{q_i \mid q_i \ \epsilon \ queue \wedge i \le queue_{max}\} & \\ S = (\text{"}batching\text{"},\ t_{max},\ \{q_i \mid q_i \ \epsilon \ queue \wedge i > queue_{max}\}) & if\ C \end{cases} \qquad (2)
$$

$$
\begin{gathered}
where \\
A : \mid queue \mid \le queue_{max} \qquad B : 2 * queue_{max} \le \mid queue \mid \qquad C : otherwise
\end{gathered}
$$

Event Prioritization: The confluent transition function of the PDEVS formalism provides a means for prioritized execution of simultaneous events. In the model translation process, the relative prioritizations are reflected in the ordering of ASM event rules. A simple, yet common, case is a confluent transition function which always assigns priority to either the internal transition function or the external transition function. The batcher model is one such case and uses internal transition prioritization (i.e., $\delta_{con}(s,\ ta(s),\ x) = \delta_{ext}(\delta_{int}(s),\ 0,\ x)$). Therefore, event rules related to the internal transition function consistently come first in the list. In more complicated cases, the prioritizations change throughout the course of a simulation, depending on state variables and event parameters. During execution, the simulation engine must order (and therefore execute (Wagner 2017)) the event rule list based on the confluent transition function.

Conditionality and State Transition Decoupling: There is inconsistency in the approach to conditional state changes across DEVS, ASM, and EG. These differences are summarized in Table 1.

Table 1: Conditional state changes across formalisms.

Formalism	Conditional State Changes
DEVS	Transition functions capture substantive state change conditionality in piecewise cases
ASM	Arbitrary conditional state change logic can be added to the event routines
EG	Conditionality can only be expressed through conditional event scheduling (only unconditional state changes are possible, directly)

In the EG formalism, events are accompanied by state transition functions, and not arbitrary event routines (Schruben 1983). These state transition functions contain no conditional logic. We must therefore decouple conditionality and state changes during the PDEVS $\rightarrow$ ASM algebraic structure translation.

To accomplish conditionality and state change decoupling, we split each possible state transition (as defined in the δ_{ext} and δ'_{int} piecewise cases) into two distinct events. The two events execute sequentially, within a superdense time segment (Pnueli and Manna 1992). We'll conceptualize each of these two-event pairs as a *superdense time trajectory* (Sarjoughian and Sundaramoorthi 2015). The first event contains only conditional event scheduling logic. The second event provides the non-conditional state changes. Together, the first event and immediate follow-up event execute conditional state change logic. However, neither of these events individually contain conditional state changes. Figure 2 illustrates this concept of decoupling the conditionality and state changes, for the batcher model.

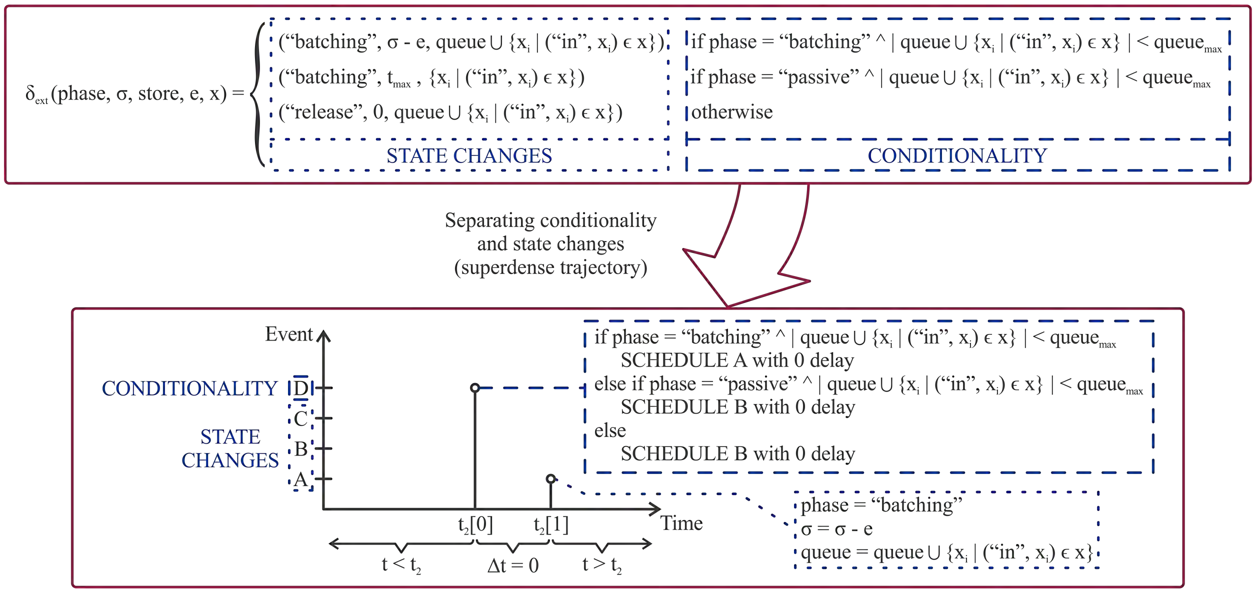

δ_ext(phase, σ, store, e, x) =
(“batching”, σ - e, queue ∪ {x_i | (“in”, x_i) ϵ x})
(“batching”, t_max, {x_i | (“in”, x_i) ϵ x})
(“release”, 0, queue ∪ {x_i | (“in”, x_i) ϵ x})
if phase = “batching” ^ | queue ∪ {x_i | (“in”, x_i) ϵ x} | < queue_max
if phase = “passive” ^ | queue ∪ {x_i | (“in”, x_i) ϵ x} | < queue_max
otherwise
STATE CHANGES
CONDITIONALITY
Separating conditionality and state changes (superdense trajectory)
Event
CONDITIONALITY
STATE CHANGES
D
C
B
A
if phase = “batching” ^ | queue ∪ {x_i | (“in”, x_i) ϵ x} | < queue_max
SCHEDULE A with 0 delay
else if phase = “passive” ^ | queue ∪ {x_i | (“in”, x_i) ϵ x} | < queue_max
SCHEDULE B with 0 delay
else
SCHEDULE B with 0 delay
t_2[0]
t_2[1]
Time
t < t_2
Δt = 0
t > t_2
phase = “batching”
σ = σ - e
queue = queue ∪ {x_i | (“in”, x_i) ϵ x}

Figure 2: Decoupling conditionality from state changes.

Model Outputs: In ASM, model outputs are handled with *RETURN* statements, at the end of event routines (Wagner 2017). However, we expand on this ASM concept through the introduction of a *YIELD* statement, which can be interpreted as *return this value at the end of the routine*. Using a *YIELD* statement, outputs precede state changes - an approach that aligns well with DEVS, where the output function precedes the internal transition state changes. Table 2 contains a modified batcher model event rule with a *RETURN* statement and Table 3 contains a modified batcher model event rule with a *YIELD* statement. Note that these are not equivalent event rules. If a state variable is changed in the Δ state transition function, then yielding that value before the change and returning that value after the change will be different outputs. The output from the Table 2 event rule and Table 3 event rule are the empty set $\varnothing$ and the pre-transition *queue*, respectively.

Table 2: Event routine return statement.

ON (Event Expression)	**DO (Event Routine)**
$ReleaseFullQueue@t$	$\Delta = \{phase = \text{“}passive\text{”},\ \sigma = \infty,\ queue = \varnothing\}$ $RETURN\ queue$

Table 3: Event routine yield statement.

ON (Event Expression)	**DO (Event Routine)**
$ReleaseFullQueue@t$	$YIELD\ queue$ $\Delta = \{phase = \text{“}passive\text{”},\ \sigma = \infty,\ queue = \varnothing\}$

3.2 Generic Translation Steps

We apply the aforementioned translation strategies to generate ASM event rules from the PDEVS definition. We start with a PDEVS model that has conditional expressions only on the right side of the function, in the piecewise conditionals. From there, we take our first step of generating the δ'_{int} function, which will be used throughout the model transformation process. The general case of δ'_{int} preparation is given in (3) and (4). (3) is the δ_{int} and λ definitions before preparation and (4) is the resultant δ'_{int} definition. The internal transition function and output function have conditions $i_A, i_B, \ldots$ and $j_A,\ j_B, \ldots$, respectively. We generate all combinations of these conditionals and use the combinations in the δ'_{int} definition.

$$\delta_{int}(s) = \begin{cases} \delta_{int,A}(s) & if\ i_A \\ \delta_{int,B}(s) & if\ i_B \\ \ldots & \ldots \end{cases} \qquad \lambda(s) = \begin{cases} \lambda_A(s) & if\ j_A \\ \lambda_B(s) & if\ j_B \\ \ldots & \ldots \end{cases} \tag{3}$$

$$\delta'_{int}(s) = \begin{cases} \begin{matrix}\lambda_A(s)\\ \delta_{int,A}(s)\end{matrix} & if\ i_A \wedge j_A \\ \begin{matrix}\lambda_A(s)\\ \delta_{int,B}(s)\end{matrix} & if\ i_B \wedge j_A \\ \ldots & \ldots \end{cases} \tag{4}$$

From the external transition function, we generate a conditional event-scheduling rule, followed by unconditional state transition event rules. We'll designate the first event rule $\delta_{ext}(e,\ x)@t$. All possible $\delta_{ext}@t$ follow-up events occur after a delay of zero, as required by the superdense time trajectory strategy laid out in Section 3.1. We will call the follow-up event rules $\delta_{ext,i}(e,\ x)@t$. In this case, $i = 1..m$, where m is the number of possible follow-up events. The follow-up events contain state changes, but no conditionality. All of these $\delta_{ext,i}(e,\ x)@t$ event rules conclude with a $\delta_{int}@\sigma$ event scheduling. $\delta_{int}@\sigma$ scheduling accommodates the required internal transition dynamics. The resulting event rules are cataloged in Table 4. At this point, we have defined the possible event trajectories, resulting from a model input.

Table 4: Derived external event rules.

ON (Event Expression)	**DO (Event Routine)**
$\delta_{ext}(e,\ x)@t$	// Conditional $\delta_{ext,i}$ event scheduling logic resulting in a one-event FEL $E' = \begin{cases} \{\delta_{ext,1}(e,\ x)@0\} & condition \\ \{\delta_{ext,2}(e,\ x)@0\} & condition \\ ... \end{cases}$
$\delta_{ext,1}(e,\ x)@t$	$\Delta = ...$ // Unconditional state transition function $E' = \{\delta_{int}@\sigma\}$ // Unconditional internal transition event scheduling
$\delta_{ext,2}(e,\ x)@t$	$\Delta = ...$ // Unconditional state transition function $E' = \{\delta_{int}@\sigma\}$ // Unconditional internal transition event scheduling
...	...

With the superdense time trajectories related to the external transition function now implemented in the event rules, we can proceed to the event rules related to the internal transition function. Similar to the external transition case, we begin with an event rule containing purely event scheduling logic, which we will call $\delta_{int}@t$. Following this new event rule are all possible follow-up events that contain state changes, $\delta_{int,j}@t$, where $j = 1..n$ and n is the number of possible follow-up events. A $YIELD$ statement begins the routine, with the $YIELD$ value dictated by δ'_{int}. $\delta_{int}@\sigma$ scheduling, again, accommodates the required internal transition dynamics. Like the preceding step, we've now added a collection of possible event trajectories to the ASM event rules.

In the event routines, note that the future events list E' is set, independent of the previous future events list. That is, the future events list does not *build*, but rather is created anew during each event routine. This is an artifact of the original PDEVS specification, which does not explicitly use an FEL.

4 EVENT GRAPH GENERATION

4.1 Graph Generation Strategy

An EG model can be generated from the ASM event rules, with a small number of special considerations. The model transformation complexity and strategy is accounted for in the translation process of Section 3.

Event Expression Parameters: Elapsed time e and input x must be passed across events, in the δ_{ext}-related superdense time trajectories. These parameters are accommodated by EG edge and vertex attributes (see the batcher model in Figure 4). The boxed e and x on the directed edge is interpreted as *pass these event parameters from the origination event to the follow-up event, during schedulings*. The $(e,\ x)$ expressions in the vertices show the required event parameters for those event types.

Model Outputs: The EG formalism doesn't explicitly support model outputs. We therefore introduce the novel notation of a boxed output expression under the event signature in an EG event. This notation reinforces the conceptualization of events as functions, by including the output in the same visual space as the event name and event parameters - similar to how a function definition in code is typically a function name, function parameters, and an output type all on one line. Further, the boxed notation mirrors that of EG edge attributes, in that boxed values signify outputs. An output of λ_1 is shown in Figure 4.

Future Event List Creation: EG includes the FEL explicitly. The FEL additions and subtractions are the event-scheduling edges and event-cancelling edges, respectively. However, our derived ASM event rules create a new FEL in each event routine. To accommodate this difference, an event-cancelling edge will need to be drawn from the $\delta_{ext}(e,\ x)$ event to the δ_{int} event. Only this single event-cancelling edge is required - an artifact of the PDEVS specification underpinnings. Causal regularity of the model accounts

Table 5: Derived event rules with external transition prioritization.

ON (Event Expression)	**DO (Event Routine)**
$\delta_{ext}(e,\ x)@t$	// Conditional $\delta_{ext,i}$ event scheduling logic resulting in a one-event FEL $E' = \begin{cases} \{\delta_{ext,1}(e,\ x)@0\} & condition \\ \{\delta_{ext,2}(e,\ x)@0\} & condition \\ \ldots \end{cases}$
$\delta_{ext,1}(e,\ x)@t$	$\Delta = \ldots$ // Unconditional state transition function $E' = \{\delta_{int}@\sigma\}$ // Unconditional internal transition event scheduling
$\delta_{ext,2}(e,\ x)@t$	$\Delta = \ldots$ // Unconditional state transition function $E' = \{\delta_{int}@\sigma\}$ // Unconditional internal transition event scheduling
...	...
$\delta_{int}@t$	// Conditional $\delta_{int,j}$ event scheduling logic resulting in a one-event FEL $E' = \begin{cases} \{\delta_{int,1}(e,\ x)@0\} & condition \\ \{\delta_{int,2}(e,\ x)@0\} & condition \\ \ldots \end{cases}$
$\delta_{int,1}@t$	$YIELD$ ... // Yield output values $\Delta = \ldots$ // Unconditional state transition function $E' = \{\delta_{int}@\sigma\}$ // Unconditional internal transition event scheduling
$\delta_{int,2}@t$	$YIELD$ ... // Yield output values $\Delta = \ldots$ // Unconditional state transition function $E' = \{\delta_{int}@\sigma\}$ // Unconditional internal transition event scheduling
...	...

for all events, except for the $\delta_{ext}(e,\ x)@t$ event rule, so $\delta_{ext}(e,\ x)$ is the only event-cancelling edge origin vertex. Meanwhile, non-exogenous events are limited to the internal transition function, so δ_{int} is the single event-cancelling edge destination vertex.

4.2 Generic Graph Generation Steps

We'll start by generating events from the $\delta_{ext}(e,\ x)@t$ and $\delta_{ext,i}(e,\ x)@t$ ASM event rules. The $\delta_{ext}(e,\ x)@t$ event rule is represented with a circle containing the event expression, in the usual EG style. As designed, this event routine contains no state changes; only conditional event scheduling. This conditional event scheduling logic is implemented as conditional event-scheduling edges, terminating at $\delta_{ext,i}(e,\ x)$ events. A delay of zero is assigned to the conditional event-scheduling edges (by convention, no delay is drawn, instead of an explicit 0). The unconditional state changes associated with each possible follow-up event rule are then drawn below the respective event circle. Note that vertex and edge attributes enable the use of e and x across all the $\delta_{ext}(e,\ x)$ and $\delta_{ext,i}(e,\ x)$ events. At this point, the diagram now includes all event rules related to the original PDEVS external transition function - Step 1 in Figure 3.

Next in the EG creation, $\delta_{int}@t$ and $\delta_{int,j}@t$ event rules are added to the graph. A δ_{int} event is drawn, along with all follow-up events $\delta_{int,j}$. Conditional event scheduling edges are used to connect δ_{int} to all of these possible follow-up events, based on the conditional logic in the $\delta_{int}@t$ event routine. These new conditional event transitions all have a delay of zero. The unconditional state changes associated with each follow-up event are drawn below the associated graph vertex. At this point, we have completed Step 1 and Step 2 in Figure 3.

Next, we need to connect the external transition logic and internal transition logic. Specifically, we need to implement the $\delta_{int}@\sigma$ scheduling at the end of every follow-up event rule. As defined in the ASM event rules, these event scheduling edges are unconditional. The resultant graph is the combination of Step 1, Step 2, and Step 3 in Figure 3.

Finally, we add an unconditional event-cancellation edge from δ_{ext} to δ_{int}, with a delay of zero. The motivation for this event-cancelling edge is discussed in Section 4.1. Figure 3 captures the final model representation. EG events may be renamed, for improved interpretability of the final EG model. However, this is not strictly required and does not impact the mechanics of the models. Without event rule renaming, the translation process is direct and automatable. We will review a software implementation of the full model transformation process in Section 6.

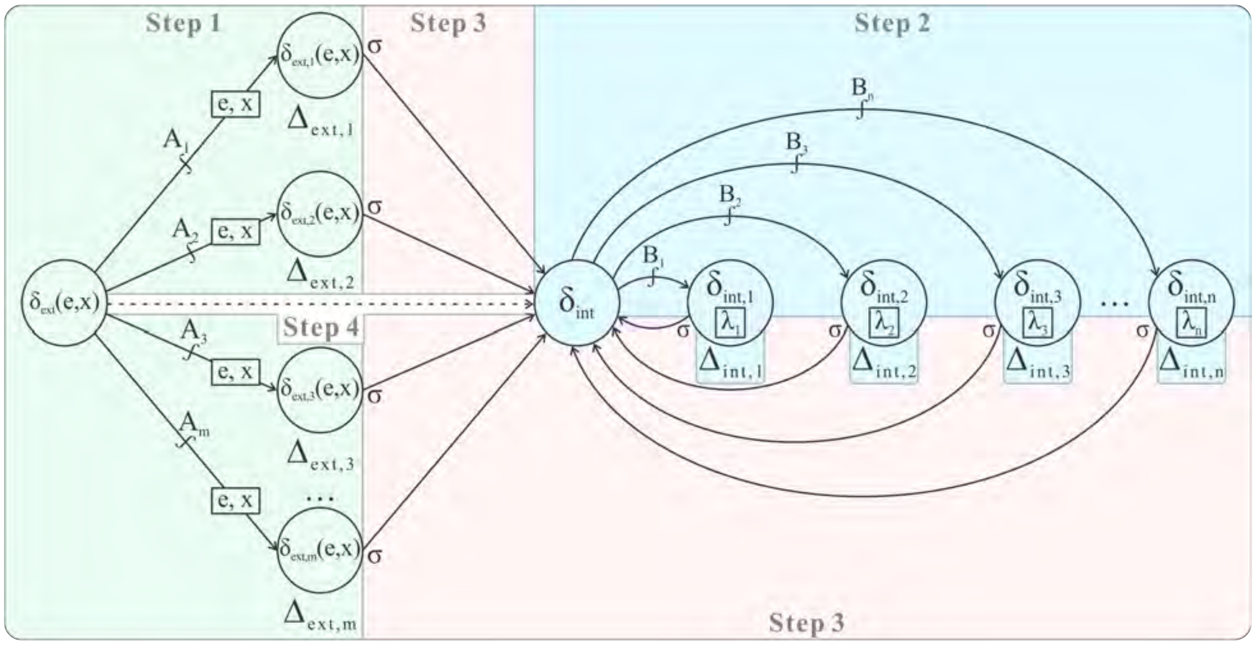


Figure 3: Model transformation - resultant event graph.

5 BATCHER MODEL TRANSFORMATION

We now return our attention to the batcher model, and use it to conceptually solidify the strategies and steps discussed so far. The batcher model specification is provided in equations 1 and 2.

Application of the algebraic structure translation process generates the ASM event rules in Table 6. The three piecewise cases in the PDEVS external transition function generate a single event rule for conditional event scheduling and three event rules for unconditional state changes. The three piecewise cases in the δ'_{int} similarly generate four event rules. With the confluent transition function consistently prioritizing δ_{int} in the PDEVS model, the δ_{int} and $\delta_{int,i}$ events come first in the event rules.

The eight ASM event rules map to eight events in the EG representation. The events, event-scheduling edges, and event-cancelling edges are drawn in accordance with the process detailed in Section 4. We can improve the model intuition by renaming the events - in this case, $\delta_{ext}(e,\ x)$ to *Receive Jobs*, $\delta_{ext,1}(e,\ x)$ to *Start Batch*, $\delta_{ext,2}(e,\ x)$ to *Add To Batch*, $\delta_{ext,3}(e,\ x)$ to *Fill Batch*, δ_{int} to *Prepare Release*, $\delta_{int,1}$ to *Release Full Queue*, $\delta_{int,2}$ to *Release Multiple*, and $\delta_{int,3}$ to *Release Partial Queue* (Figure 4).

6 MODEL TRANSFORMATION IN PRACTICE

The value of a cross-formalism transformation process is compounded by in-practice conveniences, like automation and software tooling. The dominant software implementations of the PDEVS and EG formalisms are, respectively, the DEVS Suite (for Integrative Modeling and Simulation 2021) and SIGMA (Schruben

Table 6: ASM event rules for the Batcher model.

ON **(Event Expression)**	**DO** **(Event Routine)**
$\delta_{int}@t$	$E' = \begin{cases} \{\delta_{int,1}(e,\ x)@0\} & if\ D \\ \{\delta_{int,2}(e,\ x)@0\} & if\ E \\ \{\delta_{int,3}(e,\ x)@0\} & if\ F \end{cases}$
$\delta_{int,1}@t$	$YIELD\ \lambda_1$ $\Delta = \{phase = \text{“}passive\text{”},\ \sigma = \infty,\ queue = \varnothing$ $E' = \{\delta_{int}@\sigma\}$
$\delta_{int,2}@t$	$YIELD\ \lambda_1$ $\Delta = \{phase = \text{“}release\text{”},\ \sigma = 0,\ queue = \{q_i \mid q_i\ \epsilon\ queue\ \wedge\ i > queue_{max}\}\}$ $E' = \{\delta_{int}@\sigma\}$
$\delta_{int,3}@t$	$YIELD\ \lambda_1$ $\Delta = \{phase = \text{“}batching\text{”},\ \sigma = t_{max},\ queue = \{q_i \mid q_i\ \epsilon\ queue\ \wedge\ i > queue_{max}\}\}$ $E' = \{\delta_{int}@\sigma\}$
$\delta_{ext}(e,\ x)@t$	$E' = \begin{cases} \{\delta_{ext,1}(e,\ x)@0\} & if\ A \\ \{\delta_{ext,2}(e,\ x)@0\} & if\ B \\ \{\delta_{ext,3}(e,\ x)@0\} & if\ C \end{cases}$
$\delta_{ext,1}(e,\ x)@t$	$\Delta = \{phase = \text{“}batching\text{”},\ \sigma = t_{max},\ queue = \{x_i \mid (\text{“}in\text{”},\ x_i)\ \epsilon\ x\}$ $E' = \{\delta_{int}@\sigma\}$
$\delta_{ext,2}(e,\ x)@t$	$\Delta = \{phase = \text{“}batching\text{”},\ \sigma = \sigma - e,\ queue = queue\ \cup\ \{x_i \mid (\text{“}in\text{”},\ x_i)\ \epsilon\ x\}\}$ $E' = \{\delta_{int}@\sigma\}$
$\delta_{ext,3}(e,\ x)@t$	$\Delta = \{phase = \text{“}release\text{”},\ \sigma = 0,\ queue = queue\ \cup\ \{x_i \mid (\text{“}in\text{”},\ x_i)\ \epsilon\ x\}\}$ $E' = \{\delta_{int}@\sigma\}$

where

$A : phase = \text{“}passive\text{”}\ \wedge\ \mid queue\ \cup\ \{x_i \mid (\text{“}in\text{”},\ x_i)\ \epsilon\ x\} \mid < queue_{max}$ $\qquad D : \mid queue \mid \leq queue_{max}$

$B : phase = \text{“}batching\text{”}\ \wedge\ \mid queue\ \cup\ \{x_i \mid (\text{“}in\text{”},\ x_i)\ \epsilon\ x\} \mid < queue_{max}$ $\qquad E : 2 * queue_{max} \leq \mid queue \mid$

$C : otherwise$ $\qquad F : otherwise$

$\lambda_1 : \{q_i \mid q_i\ \epsilon\ queue\ \wedge\ i \leq queue_{max}\}$

1992). Neither of these software programs feature model transformation, or have a public roadmap suggesting an intent to develop such features. This is reflective of the broader simulation software ecosystem. To the authors' knowledge, no tool that can support model transformation from PDEVS to EG models exist.

As a novel proof of concept, we introduce a DEVS-to-EG model transformation software, built on the open source SimRS discrete event simulation package [https://github.com/ndebuhr/sim]. This software takes the PDEVS atomic models included in SimRS, and generates their counterpart EG models. The first step of the process creates a JSON-serialized list of ASM event rules, by using procedural macros defined in https://github.com/ndebuhr/simx. The second step of the process creates an event graph image, by using a Python script in the same repository. A bash script connects the Rust and Python programs.

The DEVS-to-EG model transformation program is summarized in Figure 5. In the Rust source code, an attribute procedural macro is added to the PDEVS atomic model – both the model implementation code block and AsModel trait implementation code block. The model implementation defines the model-specific state transformation functions. The AsModel trait implementation defines the PDEVS atomic model interfaces.

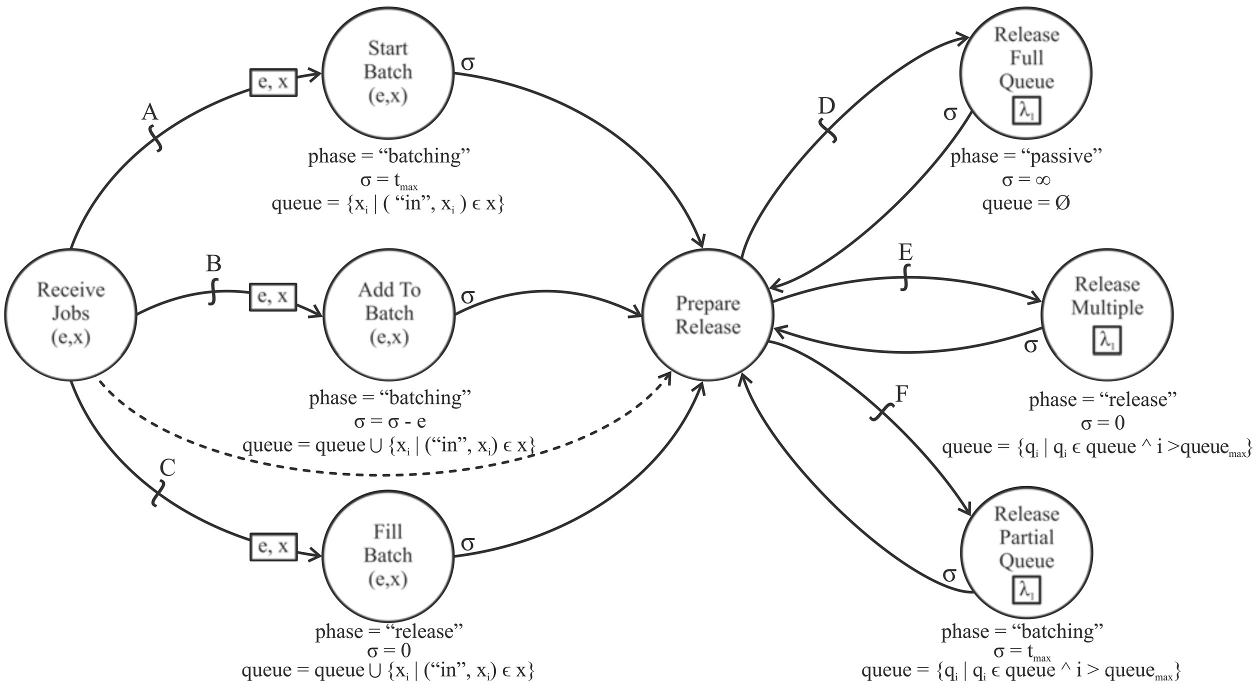


Figure 4: Derived EG model.

Both of these implementation blocks are required for a PDEVS model in SimRS. Therefore, for an existing SimRS PDEVS model, event graph generation is straightforward.

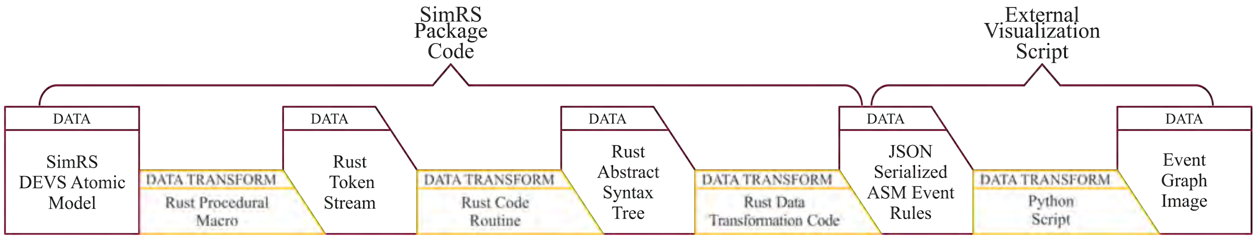


Figure 5: SimRS model transformation.

Procedural macros are a core feature of the Rust programming language, and enable a form of metaprogramming that greatly facilitates the model transformation program. First, the macro accepts the relevant source code block token streams, and transforms these streams into abstract syntax trees. The abstract syntax trees are then parsed, transformed, and serialized to generate a JSON-serialized list of event rules. In alignment with the algebraic structure translation of Section 3, we consider two event rule types – event rules with conditional scheduling and event rules with unconditional state changes. These two event rule types are explicitly differentiated, as named variants, in the data structure. A visualization script, written in Python, creates an EG image from the ASM event rules (JSON-serialized output from the aforementioned Rust program).

The transformation process introduces no new information, and generates an EG representation in an automated fashion. As with the theoretical process, the larger picture of simulators, coupled model structures, exogenous event systems, higher/lower system specification levels, higher/lower levels of abstraction, and domain-specific contextual factors are deferred to future works. Implementation detail is available at https://github.com/ndebuhr/sim and source code and additional documentation can be found at https://github.com/ndebuhr/simx.

7 CONCLUSION

The transformation process proposed in this paper is relatively straightforward – enough that it can be fully automated in software. However, additional work is needed to show how the proven properties and edge case behavior of models can be transformed, not just simple behaviors. This proof-of-concept looked at only the basic cross-formalism *model* transformation. Additional work is required to understand how model transformations of this type can fit into the larger picture of simulators, coupled models, exogenous event systems, multi-level system specification, and domain-specific contextual needs.

Even if transformation artifacts and resultant models are not directly used in a simulation project, they may still provide insights. In particular, model transformation processes and alternative model representations may allow us to see models in new, interesting, and valuable ways. Therefore, a natural extension of this work is to consider how soft factors like model reasoning and collaborative model design are impacted by the ability to generate multiple model representations of the system. Absent of a unified theory of simulation, should such a thing even be possible and desirable, investigations at the intersection of disparate DES modeling formalisms introduce important questions and perhaps useful answers along the way.

ACKNOWLEDGMENTS

We thank the referees whose critiques helped us improve the presentation of our research.

REFERENCES

Arizona Center for Integrative Modeling and Simulation 2021. "DEVS-Suite Simulator". https://acims.asu.edu/software/devs-suite/.

Jensen, K., L. M. Kristensen, and L. Wells. 2007. "Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems". *International Journal on Software Tools for Technology Transfer* 9:213–254.

Pnueli, A., and Z. Manna. 1992. "The temporal logic of reactive and concurrent systems". *Springer* 16:12.

Redjimi, M., and S. Boukelkoul. 2013. "Algorithmic tools for the transformation of Petri nets to DEVS". *Informatica* 37.

Sarjoughian, H. S., and S. Sundaramoorthi. 2015. "Superdense time trajectories for DEVS simulation models". In *Proceedings of the 2015 SpringSim (TMS-DEVS)*, 249–256: Institute of Electrical and Electronics Engineers, Inc.

Savage, E. L., L. W. Schruben, and E. Yücesan. 2005. "On the Generality of Event-Graph Models". *INFORMS Journal on Computing* 17:3.

Schruben, L. 1983. "Simulation modeling with event graphs". *Communications of the ACM* 26:957–963.

Schruben, L., and E. Yucesan. 1994. "Transforming Petri nets into event graph models". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, S. Manivnannan, D. S. Sadowski, and A. F. Seila, 560–565: Institute of Electrical and Electronics Engineers, Inc.

Schruben, L. W. 1992. *SIGMA: Graphical Simulation Modeling*. Thomson South-Western.

Wagner, G. 2017. "An abstract state machine semantics for discrete event simulation". In *Proceedings of the 2017 Winter Simulation Conference*, edited by W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, 762–773: Institute of Electrical and Electronics Engineers, Inc.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation*. Academic Press.

AUTHOR BIOGRAPHIES

NEAL DEBUHR is graduate student at the Arizona Center for Integrative Modeling & Simulation (ACIMS). He earned his M.E. at Arizona State University, with a Modeling and Simulation area of study. His research interests include web-based simulators, modeling methodology, and IT business modeling. His email address is ndebuhr@gmail.com. His website is https://debuhr.me.

HESSAM S. SARJOUGHIAN is an Associate Professor of Computer Science and Computer Engineering in the School of Computing and Augmented Intelligence (SCAI) at Arizona State University (ASU), Tempe, Arizona, and co-director of the Arizona Center for Integrative Modeling & Simulation (https://acims.asu.edu/). He is a core faculty in the School of Complex Adaptive Systems at Arizona State University. His research interests include model theory, poly-formalism modeling, collaborative modeling, simulation for complexity science, and M&S frameworks/tools. He can be contacted at hessam.sarjoughian@asu.edu.