Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

CROSS-FORMALISM DECOMPOSITION OF DEVS COUPLED MODELS

Neal J. DeBuhr
Hessam S. Sarjoughian

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

ABSTRACT

This paper proposes a cross-formalism model decomposition process such that Discrete Event System Specification (DEVS) coupled models can be automatically transformed to event graphs. We approach this process from both methodological and software implementation vantage points. A plurality of system models, from multiple modeling formalisms, may improve simulation project soft factors like collaborative model design and shared system understanding, as well as technical advantages like improved model portability. While additional research is needed to better understand the value of having multiple models, when one might otherwise suffice, well-defined and automated processes for cross-formalism modeling should facilitate the realization of this value. The choice of source and target modeling formalisms reflects the interest of the authors in investigating the role of hierarchy in these cross-formalism simulation problems and processes. Hierarchical modeling has significant overlap in technical and non-technical benefits, so it is an interesting concept to consider alongside cross-formalism modeling.

1 INTRODUCTION

Modeling formalism selection is not mutually exclusive. A researcher can combine multiple modeling formalisms in a single multi-formalism simulation, or can simulate a system multiple times, using a different modeling formalism on each iteration. Yet, the costs associated with modeling and simulating a system using multiple formalisms usually discourages this. One must consider not just the financial costs associated with compute resources, software licenses, and simulation engineer wages - there are also the opportunity costs of typing up resources (financial and otherwise) at the researcher, team, department, and organizational levels. The benefits of applying multiple modeling formalisms to a simulation project are often forgone, based on cost-benefit analyses.

By finding a way to more efficiently leverage multiple modeling formalisms, researchers stand to improve the cost-benefit situation and realize significant value for their simulation projects. Even when one modeling formalism might otherwise suffice, each new perspective has the potential to unlock, hide, emphasize, or obfuscate certain system characteristics. Well-defined and automated mechanisms of cross-formalism model transformation facilitate exactly this expanded perspective and make the process of modeling in multiple formalisms quicker, easier, and more robust.

To this end, we have previously investigated mechanisms for a well-defined and fully-automated model transformation, from Discrete Event System Specification (DEVS) atomic models to event graphs (DeBuhr and Sarjoughian 2021). At this atomic model level (or I/O systems level, from a systems theory perspective), low-level system behaviors can be defined and transformed across formalisms. However, much of the power of DEVS is in hierarchical modeling, componentization, and parallel execution, all of which are lost when

we are confined to the atomic model level. These characteristics of DEVS are particularly useful in the modeling and simulation of high-complexity systems.

Therefore, we now set out to expand on this previous model tranformation work – investigating cross-formalism model decomposition. Specifically, we explore a well-defined and fully-automated generation of flat (non-hierarchical) event graph models, from hierarchical DEVS coupled models. The models on each side of this transformation are rather distinct. For example:

- Hierarchical modeling (DEVS) vs. flat modeling (event graphs)
- Componentization (DEVS) vs. no componentization (event graphs)
- Focus on algebraic specification (DEVS) vs. focus on graphical representation (event graphs)
- Implicit future event list (DEVS) vs. explicit future event list (event graphs)

While this list is not exhaustive, it highlights the substantial differences between hierarchical DEVS modeling and flat event graph modeling. The strong distinction and differentiation of these two formalisms suggests that the models present very different views of the system of interest, a potential significant benefit.

1.1 Hierarchical Modeling

There are several philosophical considerations that form the scaffolding for hierarchical modeling. Firstly, researchers must consider which fundamental concepts and constructs should underpin a hierarchical system's representation. For example, event-focused hierarchical modeling requires a different approach and perspective, compared to state-focused hierarchical modeling. Should time be modeled as a continuous or discrete phenomenon, or should it not be modeled at all? How are inputs and outputs handled - across models, between models and the environment, and across hierarchy levels?

Fortunately, the DEVS formalism encapsulates specific opinions and approaches, with respect to general modeling, as well as hierarchical modeling. These conceptual underpinnings, for atomic models specifically, can be translated to the event-based worldview of event graphs (DeBuhr and Sarjoughian 2021). Therefore, generation of an event graph from a DEVS coupled model is only a matter of layering on coupling and I/O structures transformation. The transformation of model internal dynamics is already well established (DeBuhr and Sarjoughian 2021).

Hierarchical modeling with event graphs is in its infancy, relative to DEVS, so approaching the problem as a cross-formalism decomposition, rather than a cross-formalism hierarchical model transformation, simplifies matters. Hierarchical event graph models remains an interesting area of research, but one that we will not explore in this paper.

We approach the problem of cross-formalism decomposition in three distinct steps:

1. Recursive flattening of all DEVS coupled models, such that all coupled models are iteratively decomposed to atomic models and couplings
2. Transformation of every DEVS atomic model to its respective event graph representation (herein referred to as **event clusters**, to distinguish them from the event graph of the entire system)
3. Transformation of internal (endogenous) and external (exogenous) I/O coupling structures

In DEVS, the well-defined approach to composition and decomposition, the ability to leverage hierarchy-based abstraction mechanisms, and the closure-under-coupling property provide benefits and assurances to the modeler. Even if these benefits and assurances are lost or obfuscated during a model transformation, the derived models may still hold significant value (e.g., a more holistic modeler perspective, improved simulation flexibility, or enriched model reasoning and collaboration). Furthermore, if transformations are direct and automated, then the cost of the derived model is insignificant and the cost-benefit is compelling. Therefore, model transformations can be conceptualized as generating complementary models, perspectives, and insights - and not as substitutional models or competing system representations.

2 DEVS AND EVENT GRAPHS REVIEWS

The formalisms considered in this paper, DEVS and Event Graphs, are among the three well-established discrete event simulation formalisms: DEVS (Zeigler et al. 2000), Event Graphs (Schruben 1983), and Colored Petri Nets (Jensen et al. 2007). These are selected in an effort to build on the researchers' prior work with DEVS and event graph transformation, namely DeBuhr and Sarjoughian (2021).

2.1 Discrete Event System Specification

The DEVS formalism has proven valuable in both academic and professional circles. The rigid formal semantics, composability, and parallel execution, in particular, make DEVS an excellent fit for large and complex simulation problems. DEVS has many variations and extensions. In this paper, we specifically leverage the Parallel DEVS formalism.

Hierarchical modeling with DEVS is carried out using coupled models. Coupled models are an algebraic structure $\langle X,\ Y,\ D,\ EIC,\ EOC,\ IC\rangle$ with the following elements (Zeigler et al. 2000): X is the set of input ports and values, Y is the set of output ports and values, D is the set of components, EIC is the set of external input couplings, EOC is the set of external output couplings, and IC is the set of internal couplings.

Components within a DEVS coupled model can be coupled models themselves or atomic models. Atomic models are defined with an algebraic structure $\langle X_M^b,\ Y_M^b,\ S,\ \delta_{ext},\ \delta_{int},\ \delta_{con},\ \lambda,\ ta\rangle$ and the following elements (Zeigler et al. 2000): X_M^b is the set of input ports and values, Y_M^b is the set 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 confluence transition function, λ is the output function, and ta is the time advance function.

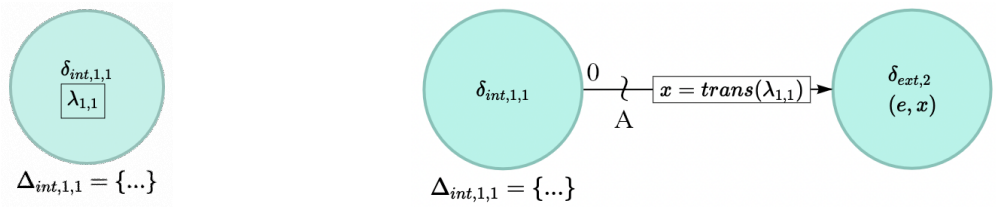
2.2 Event Graphs

Event graphs build on the foundational concepts of the event-based worldview and graph theory. The difference in worldview and nature of model representation makes model transformation across the DEVS and event graph formalisms more complex, but also more valuable - the more distinct the two perspectives are, the more likely they are to be complementary, and not duplicative. The event-based worldview of event graphs has three main components (Schruben 1983):

1. State variables that characterize the system
2. Events that change the value of state variables
3. Relationships between events

The implementation of event graphs includes:

1. Event vertices, which capture:
   (a) Event metadata (e.g., event expressions $\delta_{int,1,1}$ and $\delta_{ext,2}$ in Figure 1).
   (b) State variable changes (e.g., $\Delta_{int,1,1}$ in Figure 1).
   (c) Parameterization via vertex attributes (e.g., $(e,\ x)$ in Figure 1b).
2. Event scheduling edges (e.g., the edge in Figure 1b), which:
   (a) Can be parameterized with edge attributes (e.g., $x = trans(\lambda_{1,1})$ in Figure 1b).
   (b) Have an associated event scheduling delay (e.g., 0 in Figure 1b).
   (c) Are only engaged if a set of conditions are met (e.g., A in Figure 1b).
3. Event canceling edges, which are not required for Turing completeness (Savage et al. 2005), but enable future event list deletions for more expressive and succinct models. These edges are drawn with a dashed line, instead of the solid line used in event scheduling edges.



(a) Outputs to the simulation environment.

(b) Outputs to a coupled model.

Figure 1: Output handling across modeling scenarios.

In this paper, we will regularly use e and x in event and edge attributes. These attributes represent elapsed time and inputs, respectively, as in the DEVS formalism. With respect to event scheduling delays, zero values are often visually omitted – a practice that we will follow throughout the rest of the paper. The non-zero delay of σ, used frequently in this paper, is a transformation artifact and follows the DEVS interpretation of *time until the next event*. Regarding event conditions, a variable may stand in for an explicit listing, such as with the uppercase letter variables used in this paper. The variable values are defined in a legend elsewhere. If there is no conditionality (e.g., $A = True$ in Figure 1b), then the event condition notation is traditionally omitted.

When establishing an atomic model transformation process in previous work, we added one critical element to the original event graph specification (DeBuhr and Sarjoughian 2021). Namely, we introduce the concept of event outputs to the environment. The concept is illustrated in Figure 1, where the boxed $\lambda_{1,1}$ in Figure 1a indicate passing values from an event to the environment, similar to how the standard event graph specification leverages edge attributes to pass values from an event to another event (Figure 1b). This feature addition brings event graphs to parity with DEVS, with respect to exogenous event structures.

3 TRANSFORMATION

In this section the strategies for transforming hierarchical Parallel DEVS models to their event graphs counterparts are described.

3.1 Couplings

The primary challenge when transforming a DEVS coupled model to an event graph is the need to correctly transform couplings. Fortunately, event graphs created from DEVS atomic models have a natural, logical separation of input behaviors and output behaviors (DeBuhr and Sarjoughian 2021). This is an artifact of the logical separation in the original DEVS atomic model specification. Therefore, we can use event scheduling edges to connect the output event trajectories of one event cluster to the input event trajectories of another event cluster, as illustrated in Figure 2. In this figure, δ event expressions, $A \cdots J$ event conditions, e elapsed time parameters, x input parameters, λ event outputs, and Δ state changes are kept generic, to exemplify the general case. The choice of event trajectories count (2 for $\delta_{ext,1}$, 2 for $\delta_{int,1}$, 3 for $\delta_{ext,2}$, and 3 for $\delta_{int,2}$) is motivated purely by illustrative value.

Event scheduling edges between event clusters are largely based on the DEVS coupled model internal couplings, *IC*. The internal couplings transform some event graph outputs (Figure 1a) into event graph edge attributes (Figure 1b). These event graph scheduling edges have no delay, since DEVS does not attribute a time duration to the transmission of messages across component models. Further, these event scheduling edges have no conditional expression, as no construct in DEVS can disrupt a message transmission.

When creating event scheduling edges from the internal couplings, we must account for the possibility that the transmitting port name does not match the receiving port name. The equations below show how a mapping can be constructed, using the information about the transmitting model and associated output (denotation via α subscript), the receiving models (denotation via β subscript), and the internal couplings

IC. A bag of outputs is created by the transmitting model – a set of output port and output value pairs (Equation 1). The internal couplings pair transmitting models and their associated output ports with receiving models and their associated input ports (Equation 2). We then use the transmitting model D_α, the internal couplings *IC*, and the bag of outputs λ_α to calculate the bag of input port and input value pairs x_β for any potential receiving model D_β (Equation 3). We define a function *trans* to represent the transmission in event graphs (Equation 4). When drawing event scheduling edges in event graphs, the D_α and D_β are constant for a specific event scheduling edge and can be directly inferred. D_α and D_β are the originating event and terminating event for the scheduling edge, respectively. Similarly, *IC* is a simulation constant. Therefore, event graph edge attributes can use the shortened notation of Equation 5 (Figure 1b).

$$\lambda_\alpha = \{(p,\ v) \mid p \in P_{\alpha,out},\ v \in Y_\alpha\} \tag{1}$$

$$IC = \{((a,\ p_a),\ (b,\ p_b)) \mid a,b \in D,\ p_a \in P_{a,out},\ p_b \in P_{b,in}\} \tag{2}$$

$$x_\beta = \{(p_x,\ v_x) \mid ((D_\alpha,\ p),(D_\beta,\ p_x)) \in IC,\ (p,\ v_x) \in \lambda_\alpha\} \tag{3}$$

$$x_\beta = trans(D_\alpha,\ D_\beta,\ IC,\ \lambda_\alpha) \tag{4}$$

$$x_\beta = trans(\lambda_\alpha) \tag{5}$$

For a given pair of models, we can use D_α, D_β, *IC*, and λ_α to limit event scheduling edges only to those that could be feasibly executed during a simulation. Doing so avoids a combinatorial explosion of event scheduling edges in cross-formalism model decomposition. Specifically, event scheduling edges only need to be drawn where the input bag space is nonempty, for each output case. That is, an event scheduling edge should only be drawn between the event output case $\lambda_{\alpha,1}$ of model D_α and the input event $\delta_{\beta,ext}$ of model D_β if $x_\beta \neq \varnothing$ in Equation 3.

While the equations for *trans* are non-trivial, the interpretation of *trans* can be simplified. Effectively, *trans* is a port renaming function. When an event cluster has an output, it's in terms of that cluster's port names. However, when the event scheduling edges schedule associated follow-up events, the follow-up events need to understand that output from the perspective of their port names. *trans* handles the n:n mapping of output ports to input ports.

3.2 Simulator-Provided Attributes

The event edge attributes described in Section 3.1 rely on constants and variables that fall outside of the transmitting model and receiving model boundaries, like *IC*. However, these are constant values that can be easily inferred, that are short lived (only being used in the *trans* function call), and that don't convey any meaningful information to the modeler. Therefore, for brevity, we consider these to be inferred arguments to the *trans* function (Equation 5). On the other hand, e (elapsed time) is not a constant and generally essential to model logic. Therefore, we take a different approach around notation for the elapsed time variable. Specifically, we retain the event attribute in the event graph notation, even in the cases where it is undefined (as is the case when executing an internal-coupling-derived scheduling edge). The notation can therefore be interpreted as "when an attribute (either vertex attribute or edge attribute) is undefined in the model, it is to be provided by the simulator". Design and implementation of event graph simulators is outside the scope of this paper.

3.3 Future Event List

When transforming a DEVS atomic model to an event graph, the resultant event graph is opinionated and reflects some characteristics of the original DEVS specification (DeBuhr and Sarjoughian 2021). One such DEVS model characteristic that carries through to the event graph model is that of the future event list. While DEVS atomic models do not have an explicit future event list, they can be understood as having an implicit future event list, which only ever contains one event. Event graphs derived from these atomic models are similarly constrained to one-event future events lists (DeBuhr and Sarjoughian 2021).

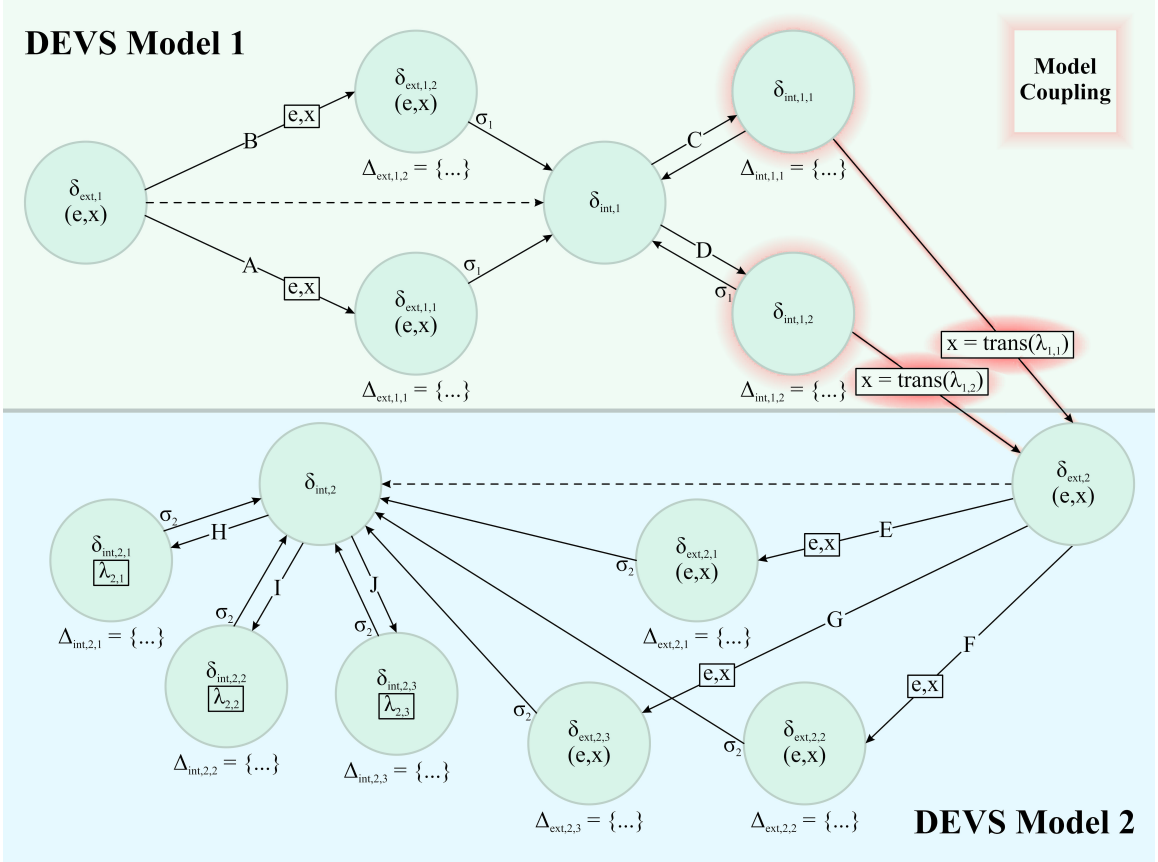


Figure 2: Input-output coupling of two event graphs.

Coupled DEVS models have a future event list with one future event for each atomic model. This is mirrored in coupled-model-derived event graphs, where each atomic-model-derived event cluster contributes one event to the future events list. This is perhaps best understood visually:

1. Event graphs derived from atomic models contain a set of superdense time trajectories (Sarjoughian and Sundaramoorthi 2015). The trajectories are either associated with an external transition (Step 1 in Figure 3) or an internal transition (Step 2 in Figure 3). In either case, the event trajectories are only ever followed by a σ-delayed internal transition scheduling edge (Step 3 in Figure 3). Therefore, execution of any of the superdense time trajectories leaves the future event list at $[\delta_{int}@\sigma]$.
2. Event graphs derived from atomic models already contain a mechanism to maintain the correct single-event future event list, in the case of exogenous events. Specifically, the event graph includes a cancellation edge between δ_{ext} and δ_{int} (Step 4 in Figure 3). Only this single cancellation edge is needed, because exogenous events only ever trigger the δ_{ext} event and a future event list that is consistently $[\delta_{int}@\sigma]$ will always be zeroed by a single δ_{int} cancellation.
3. Coupling DEVS models is strictly a matter of coupling outputs to inputs via ports. Interaction is controlled via these interfaces. Therefore, coupling does not introduce new mechanisms of model interaction, outside of δ_{ext} in either the DEVS or event graph model representation. If δ_{ext} events are already accommodated in atomic-model-derived event cluster (as previously described), then the coupled DEVS model transformation process needs no special concepts, processes, or accounting to correctly maintain the future event list for coupled-model-derived event graphs.

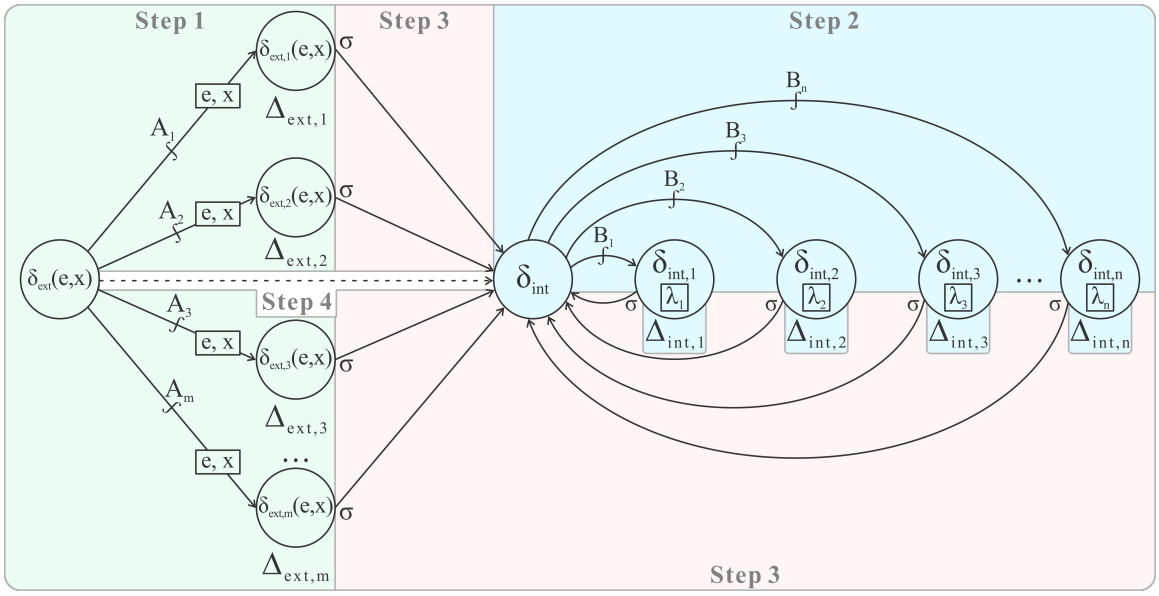


Figure 3: Specifying an event graph derived from an atomic DEVS model (DeBuhr and Sarjoughian 2021).

3.4 Event Graph Refactoring

The transformation of atomic DEVS models can produce large and complex event graphs. Naturally then, combining multiple atomic models into a coupled model can produce even larger and more complex event graphs. To combat this event graph complexity, at least visually, event graph simplification or "refactoring" can become critical. Specifically, it may be possible to reduce the number of events (vertices) and scheduling relationships (edges), so that the event graph is more comprehensible, intuitive, and succinct. This must not change the model in any functional way. There are many mechanisms by which event graphs can be refactored. For example, two events can be joined if all of the following are true:

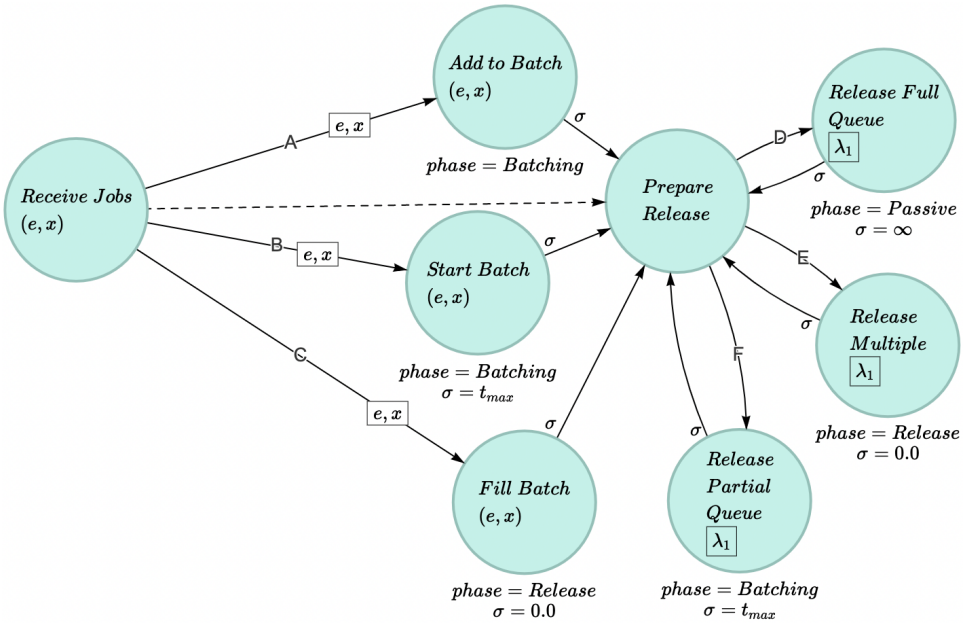
1. An event scheduling edge is the only outgoing edge for the first event and the only incoming edge for the second event
2. There is no delay or conditionality for the edge connecting the two events
3. State variable changes across the two events are commutative

4 BATCHER MODEL EXAMPLE

In previous work, we illustrated cross-formalism model transformation with a "Batcher" DEVS atomic model. The event graph for this model is provided in Figure 4 and the associated transformation-prepared DEVS specification can be found in (DeBuhr and Sarjoughian 2021).

For improved connection to that previous work, we will illustrate the cross-formalism decomposition of a "Batch Generator with Rework" model (Figure 5):

1. The Generator model and Batcher model form a "Batch Generator" coupled model.
   (a) The Generator model generates single jobs, with a consistent period T (Equation 8).
   (b) The Batcher model accepts input jobs and batches them (see DeBuhr and Sarjoughian (2021) for specification). It will place jobs into a batch until a maximum batching time t_{max} or maximum batch size $queue_{max}$ is reached, at which point the Batcher will release the batch.
2. The Rework model queues jobs ($queue \in V^+$), then with no delay passes them (probability G) or feeds them back to the Batcher (probability $1-G$) (Equation 7).



where

A : $phase = \text{“}batching\text{”} \wedge \mid queue \cup \{x_i \mid (\text{“}jobs\text{”}, x_i) \in x\} \mid < queue_{max}$

B : $phase = \text{“}passive\text{”} \wedge \mid queue \cup \{x_i \mid (\text{“}jobs\text{”}, x_i) \in x\} \mid < queue_{max}$

C : $otherwise$

D : $\mid queue \mid \leq queue_{max}$

E : $2 * queue_{max} \leq \mid queue \mid$

F : $otherwise$

λ_1 : $\{(\text{“}batches\text{”}, q_i) \mid q_i \in queue \wedge i \leq queue_{max}\}$

Figure 4: Event graph derived from the Batcher DEVS model.

$$X_M = \{(\text{“}batches\text{”}, v) \mid v \in X_p\}$$

$$Y_M = \{(p, v) \mid p \in \{\text{“}good\text{”}, \text{“}bad\text{”}\}, v \in Y_p\}$$

$$S = \{\text{“}passive\text{”}, \text{“}inspecting\text{”}\} \times \mathbb{R}_0^+ \times V^+$$

$$\delta_{ext}(phase, \sigma, queue, e, x) = (\text{“}inspecting\text{”}, 0, queue \cup \{x_i \mid (\text{“}batches\text{”}, x_i) \in x\})$$

$$\delta'_{int}(phase, \sigma, queue) = \begin{cases} YIELD\ \{(\text{“}good\text{”}, q_0)\} \\ S = (\text{“}inspecting\text{”}, 0, \{ q_i \mid q_i \in queue \wedge i > 0\}) & if\ A \\ \\ YIELD\ \{(\text{“}good\text{”}, q_0)\} \\ S = (\text{“}passive\text{”}, \infty, \emptyset) & if\ B \\ \\ YIELD\ \{(\text{“}bad\text{”}, q_0)\} \\ S = (\text{“}inspecting\text{”}, 0, \{ q_i \mid q_i \in queue \wedge i > 0\}) & if\ C \\ \\ YIELD\ \{(\text{“}bad\text{”}, q_0)\} \\ S = (\text{“}passive\text{”}, \infty, \emptyset) & if\ D \end{cases} \tag{6}$$

$$ta(phase, \sigma, queue) = \sigma$$

where

$$\begin{aligned} &A: UNIFORM(0,1) \leq G \ \wedge \ |queue| > 1 \quad B: UNIFORM(0,1) \leq G \ \wedge \ |queue| \leq 1 \\ &C: UNIFORM(0,1) > G \ \wedge \ |queue| > 1 \quad D: otherwise \end{aligned} \tag{7}$$

For brevity, we have not included the DEVS coupled model specifications. Note that the Generator model does not have input ports, so an external transition function δ_{ext} is not applicable. The Rework model uses a uniform random number generator $UNIFORM(0,1)$, which is then compared to the pass rate G to determine if a job is passed or failed. Failed jobs, where $UNIFORM(0,1) > G$, are sent back to the Batcher model "jobs" input port.

$$\begin{aligned} X_M = \emptyset; Y_M = \{(\text{“}jobs\text{”}, 1)\}; S = \{\text{“}passive\text{”}, \ \text{“}active\text{”}\} \ \times \ \mathbb{R}_0^+ \\ \delta'_{int}(phase, \ \sigma) = \begin{cases} YIELD \ \{(\text{“}jobs\text{”}, \ 1)\} \\ S = (\text{“}active\text{”}, \ T) \end{cases} \\ ta(phase, \ \sigma) = \sigma \end{aligned} \tag{8}$$

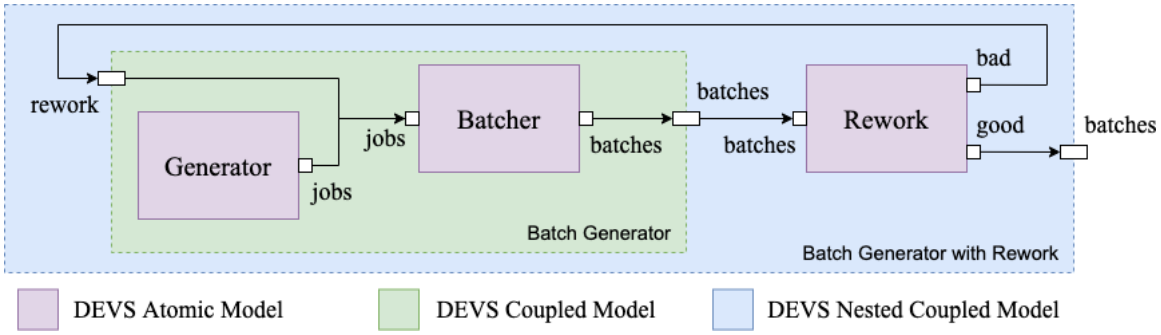


Figure 5: Hierarchical modeling of the Batch Generator with Rework.

The proposed cross-formalism model decomposition is executed across three steps (see Section 1.1): coupled models are recursively decomposed (Figure 6), event clusters are generated, and I/O coupling structures are transformed (Figure 7). For brevity, event scheduling edge conditions and event outputs are defined once (Equation 9) which can then be composed into the event graph shown in Figure 7.

$$\begin{aligned} &A: phase = \text{“}batching\text{”} \ \wedge \ | \ queue \ \cup \ \{x_i \mid (\text{“}jobs\text{”}, \ x_i) \ \in \ x\} \ | < queue_{max} \\ &B: phase = \text{“}passive\text{”} \ \wedge \ | \ queue \ \cup \ \{x_i \mid (\text{“}jobs\text{”}, \ x_i) \ \in \ x\} \ | < queue_{max} \\ &C: otherwise \\ &D: \ | \ queue \ | \leq queue_{max} \\ &E: 2 * queue_{max} \leq | \ queue \ | \\ &F: otherwise \\ &G: UNIFORM(0,1) \leq G \ \wedge \ |queue| > 1 \\ &H: UNIFORM(0,1) \leq G \ \wedge \ |queue| \leq 1 \\ &I: UNIFORM(0,1) > G \ \wedge \ |queue| > 1 \\ &J: otherwise \\ &\lambda_1: \{(\text{“}batches\text{”}, \ q_i) \mid q_i \ \in \ queue \ \wedge \ i \leq queue_{max}\} \\ &\lambda_2: \{(\text{“}jobs\text{”}, \ 1)\}; \lambda_{3,1}: \ \{(\text{“}good\text{”}, \ q_0)\}; \lambda_{3,2}: \ \{(\text{“}bad\text{”}, \ q_0)\} \end{aligned} \tag{9}$$

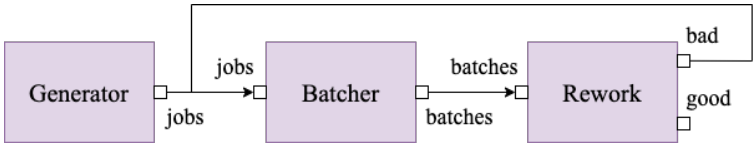


Figure 6: Recursively decomposed Batch Generator with Rework model.

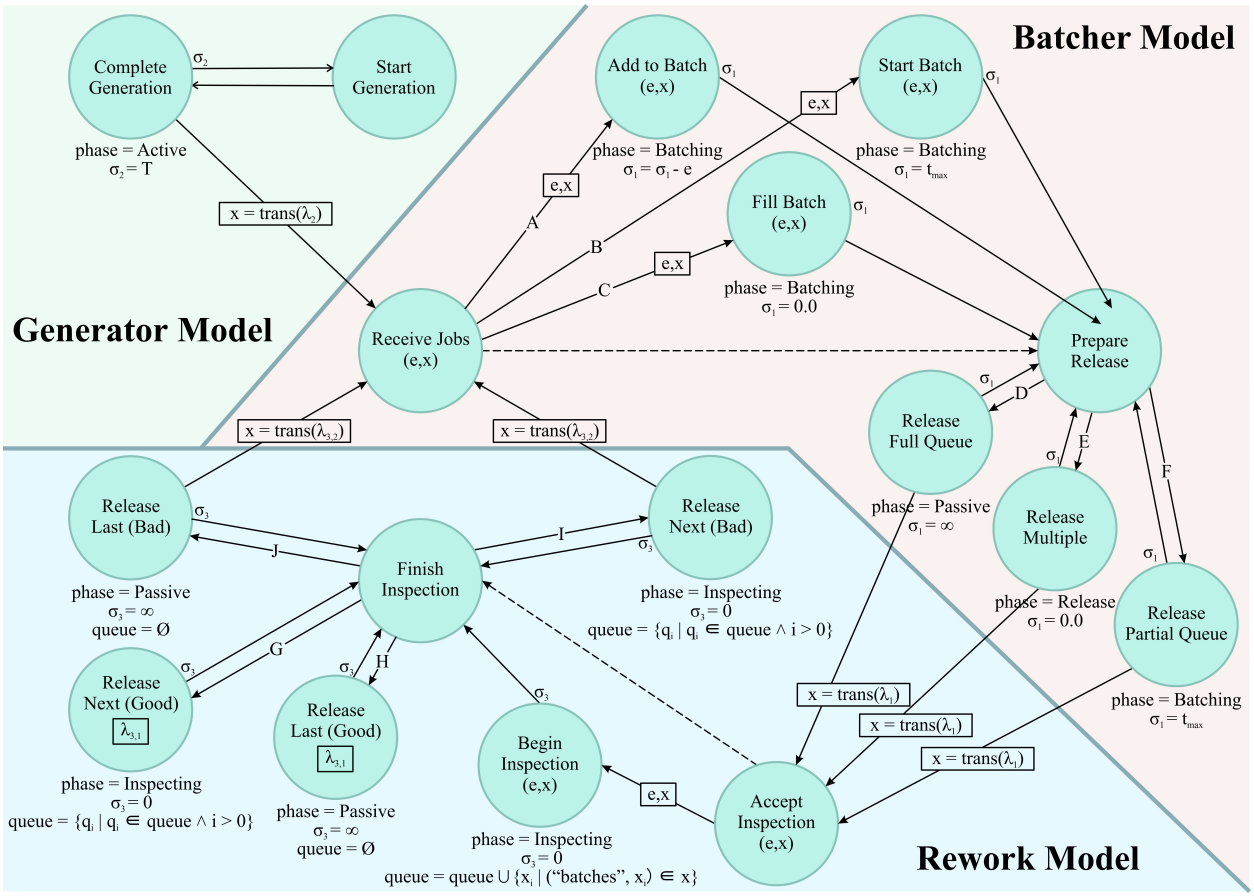


Figure 7: Transformed Batch Generator with Rework coupled model.

5 SOFTWARE IMPLEMENTATION

While methodological advances in well-defined and automated cross-formalism modeling are valuable in their own right, implementation of these advances in software is particularly useful for real-world simulation projects. To this end, we have developed cross-formalism decomposition software, within the open source simulation library SimRS. SimRS is a simulation library and engine, written in Rust. It was developed as a part of previous cross-formalism modeling research, and contains an atomic model transformation implementation (DeBuhr and Sarjoughian 2021). Expanding on this previous work, SimRS now supports the generation of event graphs from SimRS (DEVS) coupled models. This new capability required some fundamentally new approaches and programming patterns.

The existing atomic model transformation code in SimRS relies completely on meta-programming and compile-time event graph creation. This is possible, because the structure of different SimRS atomic models is programmed (not instantiated/initialized). Runtime specification of random variable distributions

and other model parameters is possible, but the structure of the model is completely defined before code compilation. This is true for both out-of-the-box and custom models within the SimRS framework. However, coupled models can be created as a runtime instantiation. That is, the models can be instantiated with a specification of component models, internal couplings, external input couplings, and external output couplings - no coding is required. Therefore, coupled model event graph generation requires a combination of both meta-programming (compile-time Rust macros for atomic model event graph generation) and regular programming (runtime Rust code for handling component interfaces and couplings). Interested readers are encouraged to explore the code repository at https://github.com/ndebuhr/sim.

6 RELATED WORKS

The concept of hierarchy for developing event graphs is noted to be important for developing reusable and scalable models (Schruben 1995). One approach toward this goal is the use of object-orientation with the listener design pattern (Buss and Sanchez 2002). A simulator called SimKit was developed for education and used for select systems (Stork 2011). The idea rests on encapsulating individual event graphs into components, each having its own data and function. Given two event graph components, one component becomes a listener to another component. When one component has an event, its listener receives the event generated by another component. The link between these event graphs is supported with a bridge, an encapsulated event graph without transitions. A bridge has an event that is registered to be listened to by another event. It maps the sender's event name to the receiver's event name. The interface concept defined in the UML class diagram is used for modeling hierarchical event graphs. The interface acts as input and output to other components. A cascade of event graph components has two components – one listens to external events and another for sending external events. The essential event scheduling with timing and multiple input and output events are not described. Furthermore, for some systems, events should be specified as primitive/complex objects, not names. From a high-level view, this approach and the event clusters approach use the encapsulation and coupling concepts. However, compared to the coupling of event clusters presented in this paper, combining event graphs using object-orientation and the listener pattern lacks formal structural and behavioral specifications.

A formal framework for specifying hierarchical event graphs supported with a set of rules for event ordering and execution with sufficiency but not always necessary conditions is developed (Som and Sargent 1989). It introduces the concepts of expanded event graphs using subgraphs. Each expanded event graph has a unique event vertex called root. Only the root vertex can have incoming edges or canceling edges. The rules must be followed to transform an event graph into its expanded form. This approach is akin to system theory in that subgraphs are unique at each level and can be hierarchically constructed. The input variables of an event (which may not belong to the future event list) and the output variables of an event (which may belong to the future event list) are connected through identity functions. A procedure supported with a set of provable properties is provided to construct and validate expanded event graphs. The concept of super events supports defining hierarchical event graphs with grouping events. Every expanded event graph has a strict tree hierarchy with a single root vertex at the highest level. This can ensure that event ordering in individual event graphs is correct in view of other event graphs. Event ordering priorities should be defined by the modeler. Defining ordering priorities grows proportionally relative to the number of expanded event graphs and their hierarchical structure. These hierarchical event graph models are akin to the classic DEVS formalism. In contrast, the coupling of event clusters is grounded in Parallel DEVS and thus benefits from multi-set event scheduling and concurrency. In other words, the execution of coupled event clusters is grounded in the Parallel DEVS abstract simulator protocol.

7 CONCLUSION

A plurality of perspectives enables rich understanding - whether that's multiple models for a system of interest or multiple researchers for a given simulation project. From a cost-benefit perspective, modeling a

system through two different modeling formalisms, when one would otherwise suffice, can initially appear unattractive. However, well-defined and automated mechanisms of generating new models greatly improves the practicality of cross-formalism modeling. In this paper, we've taken a step towards a richer framework and knowledge base, for practical cross-formalism modeling.

Hierarchical modeling introduces new complexities and risks but also opportunities to cross-formalism modeling. Hierarchical modeling's utility in DEVS overlaps with some of the possible benefits of cross-formalism modeling. For example, both may ease collaborative model design, improve model reasoning, enable a holistic systems perspective, and streamline model design. Leveraging both hierarchical and cross-formalism modeling may be a powerful way to facilitate challenging, high-complexity simulation projects – particularly when simulation project soft factors are considered. This paper brings together hierarchical modeling concepts with cross-formalism modeling concepts, in an effort to initiate more discussions and future research.

ACKNOWLEDGMENTS

We are thankful to the anonymous referees for their helpful reviews of an earlier version of this paper.

REFERENCES

Buss, A. H., and P. J. Sanchez. 2002. "Selecting Input Models". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. L. S. J. M. C. E. Yücesan, C. H. Chen, 732 – 736. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

DeBuhr, N. J., and H. S. Sarjoughian. 2021. "Model Transformation Across DEVS and Event Graph Formalisms". In *Proceedings of the 2021 Winter Simulation Conference*, edited by S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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.

Sarjoughian, H. S., and S. Sundaramoorthi. 2015. "Superdense time trajectories for DEVS simulation models". In *Proceedings of the 2015 SpringSim (TMS-DEVS)*, 249–256. Piscataway, New Jersey: 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(1):3–9.

Schruben, L. 1983. "Simulation modeling with event graphs". *Communications of the ACM* 26(11):957–963.

Schruben, L. W. 1995. "Building reusable simulators using hierarchical event graphs". In *Proceedings of the 1995 Winter Simulation Conference*, edited by C. Alexopoulos, J. Kang, W. R. Lilegdon, and D. Goldsman, 472–475. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Som, T. K., and R. G. Sargent. 1989. "A formal development of event graphs as an aid to structured and efficient simulation programs". *ORSA Journal on Computing* 1(2):107–125.

K. Stork 2011. "SimKit". https://wiki.nps.edu/display/~kastork/Simkit, accessed 02.15.2022.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation*. 2nd ed. London: Academic Press.

AUTHOR BIOGRAPHIES

NEAL J. 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.