*Methodology*

*Simulation*

*Simulation: Transactions of the Society for Modeling and Simulation International*
2015, Vol. 91(8) 715–734

DOI: 10.1177/0037549715598570
sim.sagepub.com
SAGE SCS

# Designing DEVS visual interfaces for end-user programmers

**Maryam Maleki[1], Robert Woodbury[1], Rhys Goldstein[2], Simon Breslav[2] and Azam Khan[2]**

**Abstract**

Although the Discrete Event System specification (DEVS) has over recent decades provided systems engineers with a scalable approach to modeling and simulation, the formalism has seen little uptake in many other disciplines where it could be equally useful. Our observations of end-user programmers confronted with DEVS theory or software suggest that learning barriers are largely responsible for this lack of utilization. To address these barriers, we apply ideas from human–computer interaction to the design of visual interfaces intended to promote their users' effective knowledge of essential DEVS concepts. The first step is to propose a set of names that make these concepts easier to learn. We then design and provide rationale for visual interfaces for interacting with various elements of DEVS models and simulation runs. Both the names and interface designs are evaluated using the Cognitive Dimensions of Notations framework, which emphasizes trade-offs between 14 aspects of information artifacts. As a whole, this work illustrates a generally applicable design process for the development of interactive formalism-based simulation environments that are learnable and usable to those who are not experts in simulation formalisms.

**Keywords**

Discrete-event simulation, end-user programming, visual programming, cognitive dimensions

## 1. Introduction

The Discrete Event System specification (DEVS) provides a general, rigorous formalism for describing discrete-event simulations. It is used widely by simulation researchers and systems engineers but has seen less uptake in other disciplines where it could be extremely useful. DEVS is frequently adopted as a basis for software libraries and simulation environments. Most of these tools to date have attempted to closely map their objects to the mathematical constructs of the formalism. This approach has led to features that make DEVS difficult for domain experts (who mostly have limited knowledge of simulation theory) to understand. For example, names such as ''internal transition function'' carry little meaning to those who have not studied DEVS theory, whereas the word ''model'' may cause confusion by evoking different interpretations for different users. Another source of difficulty is the formalism's separation of a simulation program into a model-independent simulator and various modeling elements. This separation provides numerous benefits, yet contrasts sharply with the imperative style of programming taught first to many potential DEVS users.

Software itself presents new opportunities in learning any formalism. Through careful interface design, support for exploratory coding strategies such as *copy-and-modify*,[1] freely available examples,[2] and features such as command-line completion, templates, and contextual help, software can assist new users in understanding a system. With the right design, software enables learning by doing. Yet traditionally, systems engineers study DEVS theory and then learn DEVS tools. A different approach is needed for domain experts who prioritize the objective of their code over its systematic development.[3] To attract a larger following, DEVS tools should assume the role of explaining the formalism and take advantage of lessons learned in end-user and visual programming.

A multitude of science, engineering, and design disciplines now have computational sub-disciplines that leverage state-of-the-art computing methods and technology to address the needs of their area. These computational communities have become large as both system capabilities

[1]School of Interactive Arts and Technology, Simon Fraser University, Canada
[2]Autodesk Research, Canada

**Corresponding author:**
Robert Woodbury, School of Interactive Arts and Technology, Simon Fraser University, 250-13450 102 Avenue, Surrey, BC, Canada V3T 0A3.
Email: robw@sfu.ca

and demand for these capabilities have grown. While domain experts within these communities know little of simulation theory, they make use of procedural scripting, are familiar to some degree with object-oriented concepts, and tend to have considerable experience with visual programming. In particular, they tend to be comfortable with dataflow visual programming, in which data flows from one entity to another, and the program is represented by a directed graph.[4] Dataflow systems are both widely used and actively developed in the architectural domain, where among other benefits the paradigm helps designers parameterize building geometry.[1] In the domain of life sciences, dataflow programming aids in the specification and execution of scientific workflows.[5]

The challenge of expanding DEVS utilization in the computational communities of various domains can be addressed by focusing on three objectives. First, change the way DEVS theory is expressed to make it easier to learn from a user's perspective. Second, make software consistent with the re-expressed DEVS theory. Third, design visual interfaces that emphasize and relate the key DEVS concepts associated with every element of the users' models and simulation runs. In the very early stages of software design, before interfaces exist to facilitate evaluation through user studies, a set of qualitative guidelines is needed to inform design decisions. We therefore propose the use of a framework known as Cognitive Dimensions of Notations,[6] and apply it to all three objectives.

In this paper we pursue the design process outlined above by first proposing a set of names to make DEVS more approachable, and then presenting a set of visual interfaces intended to inform and remind users of the formalism's essential concepts. These interfaces exhibit novel features, some prominent and others subtle, that we hope will inspire simulation tool developers. Most importantly, the rationale for all decisions is given, which will help DEVS experts re-imagine their own simulation environments and attract not only systems engineers but also end-user programmers from a wide range of disciplines. We present the concepts discussed here through the canonical simulation scenario of a bank.

## 2. Background and related work

### 2.1. Human–computer interaction

Researchers in the large and rapidly growing field of human–computer interaction have explored a number of topics relevant to the software-supported learning of formalisms and programming tools. Three topics in particular have influenced our approach to visual interface design.

**End-User Programming** This provides insight into the motivations of simulation practitioners from a range of disciplines. Writing and maintaining software is a *professional programmer's* primary job. An *end-user programmer* however, has a different primary job: he/she writes a program only to achieve a goal. This goal is not the program itself, but something related to his/her job or hobby. For example, a school teacher may program a spreadsheet to calculate class grades, or an architect may write a simulation to analyze the energy performance of a building. Simulation tools have both types of users, and both groups face barriers in authoring simulation models. The difference lies in the fact that end-user programmers are motivated by their domain and not by the merit of producing code, and therefore perceive these barriers as distractions[7] that require cognitive attention.

In the attention investment model,[8] the amount of attention a person spends on a task is considered as a currency. So for each programming task, there is a certain amount of attention *''cost''* that needs to be spent in order to receive a *''payoff''* from the program (for example, time saved by automating a task), with the *''risk''* of receiving no payoff or incurring additional costs. Users take into account the *''perceived''* costs and risks of a programming task and its payoff in their decision making process. For a simulation system to be widely used, the cost and risk of creating, editing, and using simulations must be low. In addition, the *perceived* cost of these tasks must be low and in line with the actual cost. In other words, the system should not *seem* more difficult to learn and use than it actually is.

An increase in the utilization of DEVS-based software will be accompanied by a shift in user demographics away from systems engineers and towards experts in other domains. Both barriers and attention investment are particularly salient for domain experts; hence the visual interfaces we design target end-user programmers. In general, when we use the term *user*, we mean ''end-user programmer''.

**Visual programming** Visual programming, where more than one dimension is used to convey semantics,[9] shows promise in reducing both barriers and perceived cost. *Visual programming languages* use visual expression as part of their syntax. *Visual programming environments* use traditional textual syntax with visual elements in the environment to aid in creating or editing code. There have been several studies on the effects of using visual programming techniques on program comprehension, program creation and program debugging. The results for program comprehension are generally mixed and depend on the user and the type of task. More significant improvements have been reported for program creation, as well as program debugging.[10] Visual programming has become widely accepted among end-user programmers.[11] The goal of visual programming research is not to eliminate textual code, but to improve programming languages and environments and their ease of use, as well as users' accuracy and speed of programming. These goals are pursued using the following strategies:.[9]

- *Concreteness*: expressing aspects of the program using instances.

- *Directness*: reducing the distance between the user's goal and the actions required to achieve that goal, giving the user the feeling of directly manipulating the object.
- *Explicitness*: making some aspect of the semantics (such as dataflow or relationships) explicit in the environment.
- *Immediate visual feedback*: automatically displaying the effects of program edits.

Spreadsheets are among the most successful visual programming environments. They allow the user to work on concrete data in the cells, give them the impression that they are directly manipulating the data, make the relationship between data cells explicit, and give immediate feedback on edits. Another widely used visual programming language is Rhino3D's Grasshopper add-on. Grasshopper, a favorite among architectural designers, uses nodes and links to visualize dataflow and parametric relationships between geometric elements, with the result rendered immediately in Rhino's three-dimensional model viewer. An effective visual programming environment for DEVS would employ all four strategies while amplifying the formalism's support for modular model design. This is analogous to the objective of the ConMan system,[12] an early example of visual programming involving communication between modular components.

**Cognitive Dimensions of Notations** Cognitive Dimensions of Notations is a framework for evaluating the usability of information artifacts, including both mathematical formalisms and software interfaces. The framework, first introduced by Green[6] and later elaborated by Green et al.,[13] turns the knowledge of psychology of programming into a form that is usable for non-psychologists. It gives names to concepts that may seem obvious, but significantly impact design decisions and trade-offs. The framework comprises 14 dimensions, which we mark in this paper with special typography.

- `Role-expressiveness`$^{(cd)}$: The purpose of an entity is readily inferred.
- `Hidden dependencies`$^{(cd)}$: Important links between entities are not visible.
- `Consistency`$^{(cd)}$: Similar semantics are expressed in similar syntactic forms.
- `Diffuseness`$^{(cd)}$: The notation is verbose.
- `Closeness of mapping`$^{(cd)}$: The notation corresponds closely to the problem world.
- `Hard mental operations`$^{(cd)}$: Use places high demand on cognitive resources.
- `Visibility`$^{(cd)}$: Components can be viewed and juxtaposed easily.
- `Abstraction`$^{(cd)}$: The notation provides adequate levels of abstraction.
- `Progressive evaluation`$^{(cd)}$: Work-to-date can be checked at any time.
- `Juxtaposibility`$^{(cd)}$: Different parts of the interface can be viewed and compared side-by-side.
- `Error-proneness`$^{(cd)}$: The notation invites mistakes and the system gives little protection.
- `Secondary notation`$^{(cd)}$: Notation can carry extra information by means of layout, color or other cues.
- `Viscosity`$^{(cd)}$: Much effort is required to make a change to a program.
- `Premature commitment`$^{(cd)}$: Decisions must be made before necessary information is available.

The Cognitive Dimensions of Notations framework is used in formative evaluation of software systems during their design process. For example, Jones et al.[14] use the dimensions as guides in creating the Forms/3 spreadsheet system. The framework is also used to evaluate existing systems, such as Green and Petre's[15] evaluation of several visual programming languages and environments, including LabView and Prograph. It can also serve as a feedback and evaluation method for users of a system, using the Cognitive Dimensions of Notations questionnaire.[16]

Depending on the nature of the system and the type of task it supports, some dimensions may be more important than others and trade-offs may have to be made. A system designer may try to increase the usability of the notation in one dimension, keep the usability in a second dimension constant, and inevitably reduce the usability in a third dimension. We use the framework to analyze existing systems and to design new visual interfaces, and in doing so, some dimensions came up more often than others. For example, considering the target users of our system who are end-user programmers, it is important for the notation to show the purpose of the entities as clearly and readily as possible (`role-expressiveness`$^{(cd)}$) and to reveal the dependencies between those entities (`hidden dependencies`$^{(cd)}$). Another example of a dimension that came up frequently in our design discussions was `diffuseness`$^{(cd)}$, not so much as a goal we were after, but a consequence of our design decision to use visual elements to express the notation. The list of dimensions above is ordered based on how often they came up: the dimensions that we used most frequently, such as `role-expressiveness`$^{(cd)}$ and `hidden dependencies`$^{(cd)}$, are at the top of the list.

## 2.2. Approachability of DEVS

The DEVS formalism allows essentially all simulation models to be expressed in a common form. A model is either *atomic* or *coupled*, and is specified by defining each mathematical element in the corresponding tuple in equation (1). The role of each element is described in *Theory of*

*Modeling and Simulation*,[17] the latest edition of the book which introduced DEVS in 1976. The elements are evaluated by a model-independent simulation procedure, which can be found in the same book or in Algorithm 1 of this paper's Appendix.

$$\begin{aligned} &\text{Atomic Models} \\ &\langle X, Y, S, \delta_{\text{ext}}, \delta_{\text{int}}, \lambda, ta \rangle \\ &\text{Coupled Models} \\ &\langle X, Y, D, \{M_d\}, EIC, EOC, IC, Select \rangle \end{aligned} \tag{1}$$

DEVS has two key properties that facilitate a scalable approach to the design and application of simulation models. First, given a set of models implemented in a common form based on the theory, a single simulator may perform simulation runs using any of them. The concept of a reusable simulator is not unique to DEVS, but an atomic model's relationship with certain time durations gives the formalism a high degree of generality which makes the separation of model and simulator particularly appealing. Second, DEVS models of either type can be combined to represent increasingly complex real-world systems. It is true that object-oriented programming and numerous modeling paradigms address complexity using similar compositional strategies. However, a DEVS coupled model is particularly modular in that its component models do not reference one another.

It is a testament to the generality of DEVS that models of numerous other formalisms can be formally mapped onto DEVS models.[18] This fact, combined with the decades of systems engineering research in the area, should convince simulation users in other fields that DEVS can accommodate their modeling objectives without restricting them to a specific paradigm. Nevertheless, our observations suggest that DEVS remains unused by many communities of simulation practitioners in need of a common yet flexible strategy to facilitate collaboration. We do not believe this reflects any fundamental limitation of the theory. Based on our own early encounters with the approach, we believe DEVS could achieve more widespread adoption if it entailed less effort, or rather a lower perceived cost in attention, to learn. In other words, we believe that a lack of approachability is the key factor impeding the use of DEVS.

The approachability of DEVS relates in part to its history. The formalism was largely inspired by earlier work in systems theory, notably that described in the book *A Mathematical Theory of Systems Engineering*.[19] This deeply rooted mathematical perspective contributes to a number of practical benefits for expert users. For example, there is a formula that expresses the ''legitimacy'' of a model specification. In simplified language, *legitimacy* indicates that a model will allow a simulator to progress through a simulation run without getting stuck. Many systems engineers have learned DEVS by studying these theories. However, the effort it takes to interpret mathematical descriptions of the formalism can become a deterrent for simulation users in other fields.

One must also acknowledge that DEVS requires users to develop a mental model of the simulation procedure that may be at odds with their past programming experience. Many potential DEVS users will have first learned the imperative style of programming, which represents computations in the context of control flow. Given an imperative program, a programmer can readily trace the possible execution paths from one line to the next. Control flow is somewhat hidden with DEVS, as an atomic model defines the functions $\delta_{\text{ext}}$, $\delta_{\text{int}}$, $\lambda$, and $ta$ but does not express the flow of execution between them. This separation of the simulation procedure from the models is what enables the development of reusable simulators and the rapid integration of simulation models. It makes DEVS more useful, but unfortunately less approachable.

Past work has addressed the approachability of DEVS through various types of contributions, the most obvious of which is the writing of accessible sections, chapters, and books introducing the theory. The textbook by Wainer[20] is notable in that its detailed description of a DEVS simulator, a seemly fundamental topic, is deferred until seven chapters on domain-specific applications have been presented.

DEVS can also be made more approachable by re-expressing parts of the theory. Presenting the DEVS++ library, for example, Hwang[21] re-expresses atomic models by merging the internal transition function $\delta_{\text{int}}$ with the output function $\lambda$. The resulting function is denoted $\delta_y$ to emphasize its relationship with the output set $Y$, and similarly the external transition function $\delta_{\text{ext}}$ is denoted $\delta_x$ to relate it to the input set $X$. Goldstein et al.[22] argue that similar deviations from the theory can be found wherever DEVS models are mapped from mathematical specifications onto computer implementations. The authors proceed to recommend a number of changes to the original formalism specifically intended for DEVS-based tools. Some of these changes promote computational efficiency but others might make the theory more approachable.

Helping programmers approach the formalism outlined in (1), which is known as Classic DEVS, is only one of the objectives of most introductory material on the subject. Typically there is also an intention to make a vast amount of related systems engineering literature more accessible. This literature includes numerous extensions to the theory, technological contributions, applications, and variants of the formalism. The best known variant is the Parallel DEVS formalism,[23] which better exploits parallel computing technology for isolated simulation runs. A notable variant named EasyDEVS aims to make the theory more approachable by unifying Classic DEVS and Parallel DEVS into a single formalism.[24]

When introducing DEVS with the objective of making the entire literature more accessible, it is clearly advantageous to adhere to established naming and modeling conventions. All works cited above use traditional nomenclature to a large extent. Nevertheless, it is interesting to consider whether simulation users from other communities might find DEVS more approachable if it were explained using familiar terms and with only minimally necessary links to systems engineering. Efforts have been made to exploit established conventions within particular areas of expertise. For example, Mooney and Sarjoughian[25] combine DEVS with the Unified Modeling Language (UML), which is popular among software engineers. The DEVS/UML combination is clearly useful, but does not align with our user communities, whose members understand mainly basic imperative languages and visual programming interfaces. While interpreting DEVS through UML makes DEVS accessible to the software engineering community, our work further expands this to communities that practice more basic programming skills and techniques.

### 2.3. Existing visual interfaces for DEVS

To make DEVS more approachable through the design of visual interfaces, one may build upon previous work on domain-independent DEVS-based simulation environments. Several of these tools feature interfaces that lower the perceived cost, and in many cases the actual cost, of learning and applying various aspects of DEVS theory.

Visual node editing interfaces provide a means of viewing and editing coupled models in DEVS-based simulation environments such as PowerDEVS,[26] CD++ Builder,[27] and GVLE.[28] Such interfaces alleviate the need for textual specifications of coupled model components and the links between them. They help to distance users from the mathematical elements of the formalism, in some cases for atomic models as well as coupled models. For example, PowerDEVS provides a library of pre-implemented atomic models, and a user new to DEVS can prepare a simulation by simply creating instances of these atomic models and visually connecting them. Graphical editing of compositional models is very common in non-DEVS simulation environments such as Simulink,[29] Xcos,[30] Ptolemy II,[31] and Dymola,[32] to name a few. The discrete-event simulation environment OMNeT++[33] allows both textual and graphical editing of coupled models, and editing in either interface causes the other to be updated automatically.

Visual interfaces that show the transfer of information between events serve at least two important roles. First, they reveal to a novice user both correct and false assumptions in their understanding of the simulation procedure. Second, they assist all users in debugging their models. Existing simulation environments demonstrate multiple ways to represent this dynamic information. A timeline interface such as the Sequence Chart in OMNeT++ plots events in order of occurrence, along with arrows representing messages between events. Both OMNeT++ and the MS4 Me simulation environment[34] feature an alternative style of interface in which messages are not plotted but rather animated in sequence. By animating data transfer along the links of the node editing interface, a user may more easily relate occurrences in the simulation run to elements of the coupled model. On the other hand, the timeline's ability to simultaneously display events and messages over a period of simulated time may be useful for observing certain patterns. Our interest is in achieving the benefits of both styles of interface by designing timelines exhibiting visual similarities with model representations.

To simplify programming of the atomic models, DEVS-based simulation environments support templating in various forms. For example, CD++ Builder and MS4 Me can generate class definitions with all the required member functions, so that the programmer need only provide the body of each function. PowerDEVS separates atomic model functions into different tabs in the interface, hiding the visual complexity of an object-oriented class. These templating techniques allow users to focus their concentration on the logic of their models rather than tool-specific syntactical conventions. That said, the user must still have a solid understanding of DEVS at the outset, since otherwise he/she will not know when each function is invoked nor how data is transferred from function to function. The challenge for future interfaces is to continue to guide the user in authoring a model while also clarifying the simulation procedure and the role of each function with respect to data.

A number of existing DEVS-based tools allow users to approach DEVS indirectly by starting with fully diagrammatic modeling paradigms. For example, both MS4 Me and CD++ Builder include State Diagram editors in which nodes represent assignments to state variables and links describe transitions between these assignments. State Diagrams and other fully diagrammatic models can be automatically converted into DEVS models, so in a sense their authors are using DEVS. However, our interest lies in visual interfaces that not only produce DEVS models but also communicate the distinguishing aspects DEVS theory.

## 3. Re-expressing DEVS theory

The primary objective of this work is to design visual interfaces that help end-user programmers arrive at some reasonable interpretation of the formalism, and it is important to acknowledge that a wide range of interpretations are possible. We must first develop a clear picture of how we wish users to interpret the formalism, as only then can the visual interfaces be designed accordingly.

If our goal were to teach users the mathematics of the formalism, the visual interfaces might display symbols such as $\delta_{int}$, and the associated functions would be labeled

with ''internal transition function'' and other well-established names. We decided to take a different approach and aim to help users bridge from their general and domain knowledge to a qualitative but effective understanding of how to use DEVS. Because we are tailoring our interfaces not for systems engineers but for programmers in other domains, we take great liberties in proposing our own names to make DEVS more approachable. We explain each departure from the literature.

We begin with the word ''model'', which evokes different interpretations for experts in different domains. Consider for example a simulation of heat transfer through a building. To the DEVS expert, the relevant ''model'' is that of the heat transfer process, and, if it is an executable model, it will require building geometry as initial input data. To an architect, by contrast, the heat transfer process is part of a solver while the building geometry is the ''model''. Thus this single word may become an early source of confusion when DEVS is first introduced to an architect, and the same can be said of many other types of simulation users.

To address the inevitable overloading of the word ''model'', we advocate the use of more specific terms wherever possible. Consider the four statements below.

1. *In the proposed* ***model****, the withdrawal request must be rejected if amount* $>$ *balance.*
2. *The bank machine* ***model*** *is defined as* $\langle X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta \rangle$, *where* $X = \{(p, v) \mid p =$ ''Arriving Customer$''\wedge\ldots$
3. *Open the atomic* ***model*** *by selecting the file ''bank_machine.java''.*
4. *When the* ***model*** *receives an ''Arriving Customer'' message, it computes the service duration.*

The first model is likely a *conceptual model*. The second is a *specification* that may formalize the conceptual model. The third is an executable description of a bank machine, or rather of a *class* of bank machines that can be instantiated by supplying parameters. The executable description may implement the specification. The fourth ''model'' represents an individual bank machine in the context of some scenario; it may be an *instance* of the executable description. When explaining DEVS to a broad audience, we would minimize the use of ''model'' by favoring ''conceptual model'', ''DEVS model specification'', ''DEVS class'', or ''DEVS instance'' in situations corresponding to those above.

We acknowledge that these conventions are a departure from the literature. Systems engineers will point out that ''a DEVS model specification'' may simply be called ''a DEVS'', but we expect this abbreviation to cause confusion for some users. Instead, the phrase ''a specification'' may be adopted when ''DEVS model'' is implied by the surrounding context. A more significant departure from the literature is our use of ''class'' and ''instance'', which suggest a strong relationship to the similarly named object-oriented concepts. We believe this analogy is appropriate and will be helpful to experts in the computational sub-disciplines of their domains. In fact, DEVS models are often implemented as object-oriented classes, and in these cases the relationship is more than just an analogy. We note the word ''model'' has been identified as a source of confusion in non-DEVS modeling frameworks, and this confusion has even been linked to costly user errors in managing model libraries.[35]

We turn our attention to the term ''coupled model'', which follows from the *coupling of systems* concept that predates DEVS. This well-establish terminology reflects a bottom-up perspective on system composition, where pairwise connections between interacting subsystems are introduced one at a time until the desired level of complexity is achieved. Unfortunately, a ''couple'' implies two of something, whereas a ''coupled model'' can in fact have any positive number of components. We therefore find ''composite model'', as used in the DEVS-based CoSMoS environment,[36] more approachable. In most cases we would avoid ''model'' and explain to users that a DEVS class is either an *atomic class* or a *composite class*. Similarly, a DEVS instance is either an *atomic instance* or a *composite instance*. Professional programmers will be reminded of the *Composite Design Pattern*,[37] which simplifies the representation of hierarchical structures in a manner closely related to the DEVS principle of *closure under coupling*.

The word ''component'' is retained, but we remind ourselves that a *component* is strictly an element of a composite class. A component represents a DEVS instance, but the terms are not interchangeable. In a simulation involving one atomic class, there is one atomic instance but no components. In a simulation involving one composite class with four components assigned atomic classes, there are four atomic instances plus one composite instance.

In the context of the theory, it is reasonable to refer to simply ''an input'', since DEVS models are associated with only one form of influencing data: a value, affixed with a port name, received at some point in time. In practice, a *parameter* assigned at the beginning of a simulation run might also be considered ''an input''. Similarly, ''an output'' could be interpreted as a timed (port name, value) pair as in DEVS theory, but it could also refer to a *statistic* calculated at the end of a simulation run. We therefore avoid the use of ''input'' and ''output'' as countable nouns describing data, at least where the names ''parameter'', ''message'', and ''statistic'' allow us to be more specific. Note that a *message* is the value that is actually transferred between DEVS instances, without any affixed port name.

Because parameters, messages, and statistics all pertain to information transferred to or from a DEVS instance, it is reasonable to organize all three types of data using named *ports*. Both atomic and composite classes are permitted to

have any number of any type of port. The four types of ports are listed below along with the associated type of data and the relevant phase of a simulation run.

- *Initialization ports* represent parameters whose values are provided to the DEVS instance during the *initialization phase* that starts a simulation run.
- *Simulation input ports* receive messages during the *simulation phase* of a simulation run.
- *Simulation output ports* send messages during the *simulation phase* of a simulation run.
- *Finalization ports* represent statistics whose values are assigned by the DEVS instance during the *finalization phase* that concludes a simulation run.

The qualifiers ''initialization'', ''simulation'', and ''finalization'' allow several concepts to be introduced with a great deal of symmetry. For example, there are *initialization links, simulation links*, and *finalization links*. Each link may only connect to the similarly named type of port. Each link directs its associated type of data, either a parameter value, a sequence of messages or a statistic value, during the similarly named phase of a simulation run. One possible objection to this scheme is the use of ''simulation'' to refer to only one of the three phases. The justification is that the other two phases, initialization and finalization, rely only on familiar programming concepts. DEVS can therefore be described as building upon a programmer's preexisting knowledge through the incorporation of a simulation phase and its associated time-related elements: messages, simulation ports, simulation links, and simulation events.

A *simulation event* represents a self-contained set of computations associated with a particular DEVS instance and a single point in simulated time. For an atomic instance, every simulation event is associated with an invocation of either an internal or external state transition function. We observe that the name ''internal transition function'' can be regarded as an abbreviation of ''internally triggered state transition function''. Although the full 12-syllable phrase is prohibitively long, the omission of ''state'' conceals the type of object that is transitioning, while the omission of ''triggered'' leads to ambiguity. If a user assumes that ''internal'' and ''external'' refer to the scope of a transition's effects, he/she might incorrectly associate internal and external transitions with, respectively, the input and output of messages.

After searching for relatively short names that unambiguously differentiate the two types of simulation events, we settled on ''planned event'' and ''unplanned event''. A *planned event* is an internally triggered event that was at one time scheduled, but at some point actually occurred. Note that in a discrete time simulation, all events are ''planned'' in the sense that their timing is known at the outset; appropriately, such simulations have only planned events. A *unplanned event* is an externally triggered event that effectively cancels the upcoming planned event due to the ''unplanned'' arrival of a message.

The functions associated with planned and unplanned events are, respectively, the *planned event handler* and the *unplanned event handler*. We use ''handler'' in place of ''function'' to distinguish these atomic class elements from the more general functions with which all programmers are familiar. Event handlers differ from regular functions primarily in their relationship with time. The value known as the ''elapsed time'' is preserved, but we refer to it as the *elapsed duration*. The word ''duration'' better reflects the fact that this quantity is measured relative to a previous event, as opposed to the beginning of the simulation. What is known as the *lifetime of the state*, or formally $ta(s)$, can instead be referred to as the *planned duration*. The name ''planned duration'' exhibits a helpful symmetry with ''elapsed duration'', and also emphasizes the fact that a ''planned'' event will occur should this duration of time elapse.

Having proposed a reasonably comprehensive set of names to describe DEVS to an end-user programmer, we are now ready to apply these phrases in the design of visual interfaces. Beforehand, let us briefly reflect on several principles from the cognitive dimensions of notations framework that provide a general rationale for the new names.

When choosing a name, `closeness of mapping`$^{(cd)}$ suggests that its meaning in a DEVS context should map closely to its meaning in a context already familiar to the user. This affirms our restrictions on the use of ''model'', which carries numerous domain-specific meanings, as well as our avoidance of ''coupled'' so as not to imply ''a pair''. We believe that ''class'', ''instance'', and ''composite'' avoid misleading associations and map closely to object-oriented concepts that are growing in popularity even among novice programmers.

When a name is used in a DEVS-related context, it should always carry the same meaning. Our names promote this type of `consistency`$^{(cd)}$ in a preemptive fashion. For example, it is likely that, at various points in a user's experience with DEVS, ''input'' will be applied to some sort of initial value as opposed to a value received at a point in simulated time. Such inconsistencies are difficult to avoid, unless one actively promotes the use of more specific terms such as ''parameter'' and ''message''.

It is common to encounter trade-offs between cognitive dimensions. A clear example of this is the widespread avoidance of ''internally triggered state transition function'' in favor of ''internal transition function''. The first phrase clearly suffers from `diffuseness`$^{(cd)}$ due to its length, but the latter gives up `role-expressiveness`$^{(cd)}$ to some degree by omitting two words that help describe the element. We consider ''planned event handler'' to be a sufficiently short and reasonably

role-expressive alternative. The phrase ''lifetime of the state'' and the semantically equivalent expression ''*ta*(*s*)'' exhibit the same trade-off, and for that reason we prefer ''planned duration''.

## 4. Designing visual interfaces

### 4.1. Atomic class interface

Unlike their composite counterparts, atomic classes are structurally similar to one another. They all have the same four event handlers, and each handler is always linked with a particular one of the four types of ports. To mirror this structural consistency in interactive `consistency`$^{(cd)}$, a visual atomic class interface should show all ports and handlers of an atomic class in one place, and allow a user to design a new class by essentially ''filling in the blanks''. We acknowledge that this level of assistance may not be necessary for a systems engineer already familiar with DEVS theory. For an end-user programmer, however, a comprehensive, template-like visual interface may act as a valuable learning aid as well as a practical programming tool.

Figure 1(a) illustrates our proposed atomic class interface as it might appear when exploring or enhancing a hypothetical ''Bank Machine'' class. The fact that there are four types of ports conveniently allows them to be grouped at the corners of the interface. By scanning the corners in left-to-right, top-to-bottom order, we see quickly that the Bank Machine has three initialization ports representing parameters, two simulation input ports for receiving messages, four simulation output ports for sending messages, and two finalization ports representing statistics.

Arranged in a diamond-shaped configuration, the four event handlers exhibit the following symmetries:

- each row of handlers is associated with a particular phase: initialization, simulation, finalization;
- the upper three handlers compute planned durations (arrows lead to the clock symbol);
- the lower three handlers observe elapsed durations (arrows lead from the clock symbol).

In order to reduce the number of handlers to the four shown, a number of conventions were adopted from existing DEVS tools. The time advance function *ta* is absorbed into the upper three handlers, as is done in DEVSJAVA[38] and CD++.[20] Users of this interface will therefore learn about ''planned durations'' in place of *ta*. Also, the output function $\lambda$ is absorbed into the planned event handler (i.e. internal transition function). As in DEVS++,[21] this deviation from the mathematical formalism emphasizes the symmetry between unplanned and planned events with respect to simulation input and output ports. As recommended by Goldstein et al.,[22] a single planned event may produce multiple messages. This is convenient in the case of the Bank Machine class, assuming the departure of a ''Served Customer'' is modeled to coincide with an ''Account Change'' message communicating the result of a withdrawal or deposit transaction. The Appendix provides a mapping between this adaptation of DEVS and the classic version of the formalism, showing that they are essentially equivalent.

Having identified the most prominent elements of the atomic class interface, we turn our attention to more specific design decisions and the cognitive dimensions they promote. These decisions address various user needs we consider at high risk of being neglected in DEVS-based software.

**A user needs to understand the simulation procedure as they develop an atomic class**. The *separation of model and simulator* is an essential principle of DEVS, but it is important that ''separation'' be interpreted as ''semantically distinct and independent'' as opposed to ''visually separated''. It is common in DEVS tools for the simulation procedure to be hidden from users, or shown only in the documentation. This lack of visibility forces the users to imagine the simulation procedure in their head, or to switch their attention back and forth between the documentation and the model interface in an effort to determine the order of events. The timing of events may also be unclear to an inexperienced user. He/she may discover from the code how the planned duration is calculated, yet fail to understand when the simulator actually applies this duration and advances the simulation time.

To increase the `visibility`$^{(cd)}$ of the simulation procedure, and to reduce `hard mental operations`$^{(cd)}$ in determining the order of events, we use arrows, decision points, and loops to show the flow of program execution. The decision points in Figure 1(a) indicate that if the ''*Planned duration…*'' has ''*elapsed*'', a planned event occurs; if it is ''*Interrupted by…*'' an ''*incoming message*'', an unplanned event occurs; if it is ''*Interrupted by…*'' the ''*end of simulation*'', finalization occurs. Observe that the point in the procedure where time advances is marked by a clock symbol, promoting `closeness of mapping`$^{(cd)}$ between simulation time and a user's preexisting knowledge of time.

The interface in Figure 1(a) combines model and simulator elements from a visual standpoint, yet keeps the model and simulator semantically separate. To maintain this separation, the key principle is that the only elements of the interface that may change are those related to the model: the number of ports, the names of the ports, and the code within each handler. The thick gray arrows, loops, decision points, and clock symbol are simulator elements that appear the same for all models and remain unchanged as models are developed. The distinction between editable model elements and non-editable simulator elements, which can be further emphasized by highlighting editable

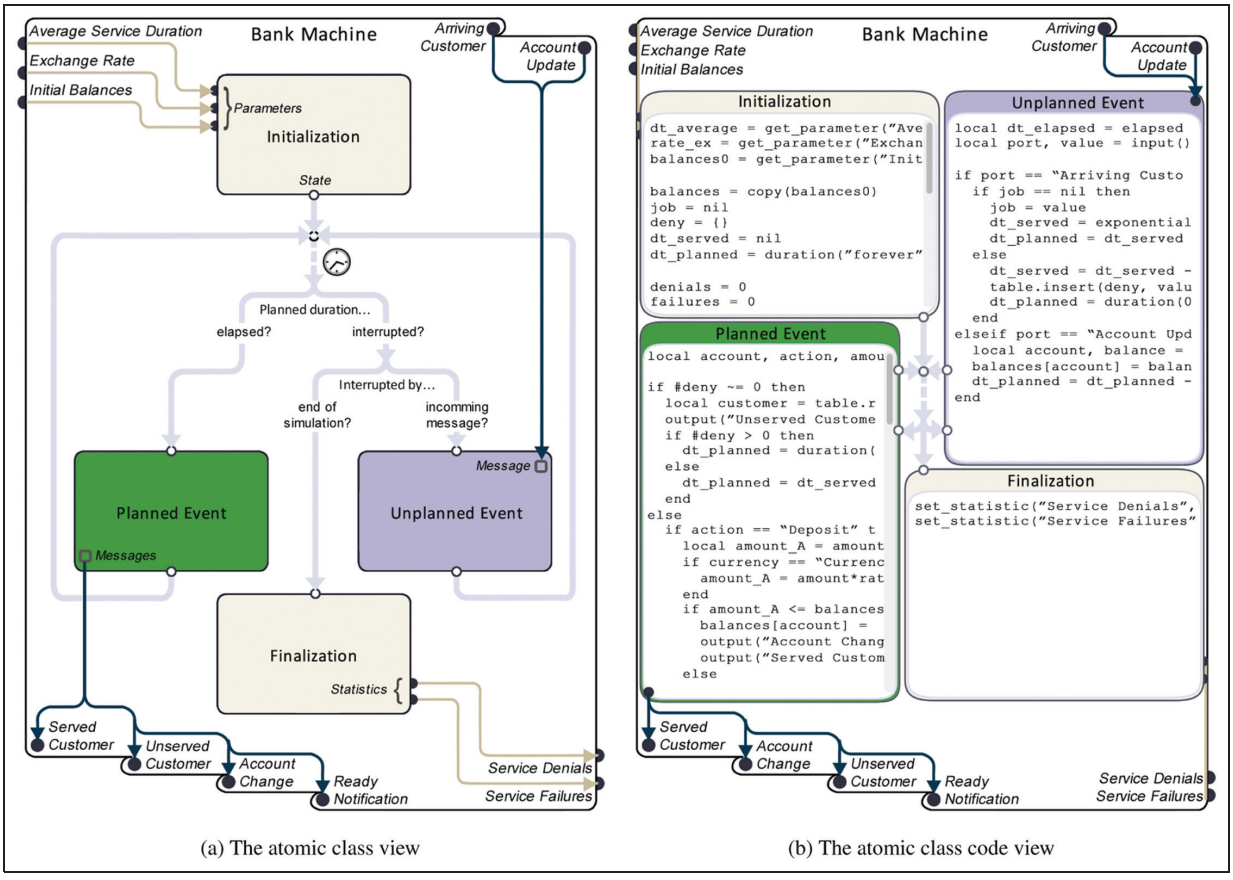


**Figure 1.** The visual interface of the atomic class with handlers collapsed (a) and expanded (b).

objects when hovered over, should help the user discover that with DEVS he/she has neither the ability nor the need to alter the simulation procedure. In fact this fundamental concept can be explicitly stated in a dialog box that would appear if a user attempts to alter a simulator element by clicking on it. Users will be familiar with the locking of cells in an Excel spreadsheet: one of many examples of editable and non-editable objects juxtaposed in a helpful way.

**A user needs to understand the role of each handler in observing and computing data**. In DEVS theory, the input and output of every function is expressed mathematically with statements such as $\delta_{\text{ext}} : Q \times X \rightarrow S$. Systems engineers typically learn these statements prior to using DEVS in a programming environment, but end-user programmers need the software to first inform and later remind them of the role of each handler with respect to data.

To indicate the flow of data into, out of, and among the four event handlers, our atomic class interface combines the procedural representations mentioned above with elements from dataflow programming. This increases the `role-expressiveness`$^{(cd)}$ of each handler. As shown in Figure 1(a), the visual interface also uses shapes, colors and orientations to distinguish between different types of data and their respective ports. Initialization and finalization links are pale gold in color, and flow in a left-to-right direction. Simulation links are blue with a top-down direction, carrying messages from the simulation input ports to the unplanned event handler and from the planned event handler to the simulation output ports. State links, shown in gray, serve three purposes. First, as previously discussed, they play a key role in illustrating the simulation procedure. Second, they clarify the role of the handlers in observing and modifying the state of an atomic instance. Third, they draw attention to possible `hidden dependencies`$^{(cd)}$ between handlers: a state change produced by one handler may influence computations in any downstream handler.

**A user needs to relate sections of code to one another and to other atomic class elements**. When the DEVS formalism is used to present an atomic model specification, each function is defined by a separate formula identified by a variable: $\delta_{\text{ext}}$, $\delta_{\text{int}}$, $\lambda$, or *ta*. Similarly, when a visual interface is used to program an atomic class, each handler may be coded in a separate text area identified by a label: ‘‘Initialization’’, ‘‘Unplanned Event’’, ‘‘Planned Event’’, or ‘‘Finalization’’. But whereas examples of

atomic model specifications often fit on a page, we cannot expect all of an atomic class's code to fit on a computer screen. An obvious way to address screen size limitations is to place each handler's code in a scrollable text area on its own tab. Unfortunately, this design may prevent a user from simultaneously viewing related sections of code in different handlers. Another option is to dedicate a separate floating window to each hander. Floating windows may provide too much flexibility, however, making it hard to remember which handler was placed where. Aside from the issue of displaying multiple sections of code, it important for an end-user programmer to relate the code to the simulation procedure and data flow logic.

To help users view sections of code in a helpful context, we envision the use of animated transitions to expose handler code in two stages. In the first stage, the four handler compartments in Figure 1(a) expand, revolve around their common center by roughly 45°, and acquire scrollable text areas as shown in Figure 1(b). Juxtaposing the four event handlers in this fashion reveals `hidden dependencies`(cd) between them at the level of the code. Where possible, the ports and links of Figure 1(a) are retained in Figure 1(b) to remind users of each handler's relationship with the simulation procedure and various types of data. Port styles and link colors are maintained, promoting `consistency`(cd). In the second stage, a user may focus on a single handler by expanding the selected compartment in Figure 1(b) to completely cover the other three compartments.

**A user needs to correct and/or confirm his/her understanding of the simulation procedure and atomic class code**. *Understanding barriers*, according to Ko et al.,[39] emerge when end-user programmers cannot evaluate the behavior of the program relative to their expectations. In other words, even when they think they understand the simulation procedure, the program does something they did not expect. Ko et al.[39] also report *information barriers* occurring when users catch an error or unexpected behavior, but do not know what causes it and how to go about fixing it. Increasing the `visibility`(cd) of the simulation procedure in the context of specific runs, and maintaining its connection with the atomic class code, help in lowering understanding and information barriers.

Figure 2 shows the visual design of the atomic class debugging tool, which consists of the atomic class code view on the left and a timeline interface on the right. When a simulation is run, unplanned events (purple bars near the top of the interface) and planned events (green bars near the bottom) appear on the timeline. There they exhibit the same colors as their corresponding handlers in the atomic

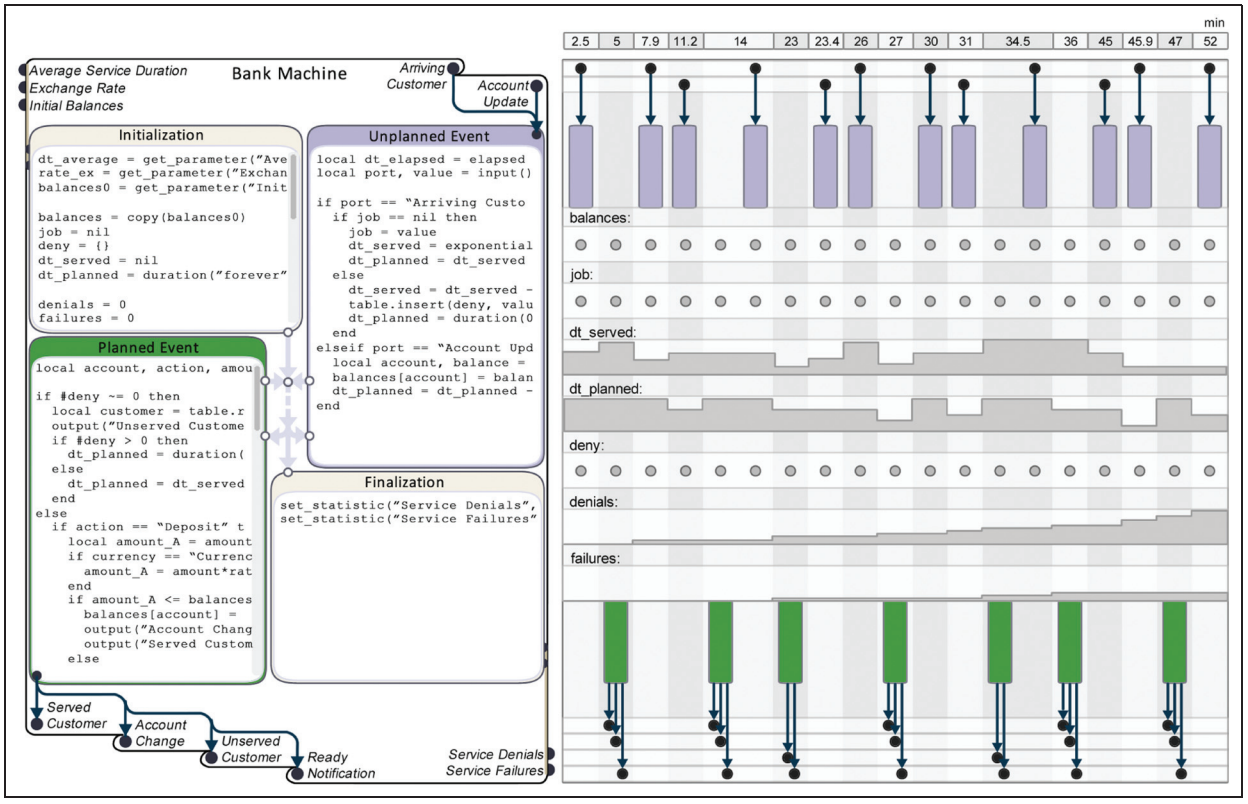


**Figure 2.** The atomic class debugging tool, consisting of the code view and the visualization of events with messages and state variables displayed in the timeline on demand.

class interface to maintain consistency(cd). Ports and events in the timeline are aligned with the code view to reduce hard mental operations(cd) involved in relating the two views. To further emphasize related elements, hovering over an object in one view may produce highlighting in both views.

The debugging tool allows the user to *watch* variables and messages to aid in finding the source of errors and fixing them. The horizontal alignment of variables and messages along a time axis is common practice for static diagrams and is also seen in the DEVS-Suite simulation tool.[40] A feature more unique to the proposed interface is the vertical alignment of ports between views, as well as the alignment of the upper/lower boundaries of unplanned/planned events and their respective handlers. This arrangement of elements gives rise to a large region in the center of the timeline where state variables can be visualized. Seven variables are shown in the middle of the timeline in Figure 2. The timeline navigation tool at the top allows the user to go back and forth in time and track these variables over time, reducing the need for *breakpoints* to stop the process before it crashes and *println* statements to observe past values of variables.[41] We acknowledge that the elements in the timeline will have to change dynamically as the user zooms out to reveal thousands of events. Multiscale widgets such as FacetZoom[42] may help the debugging tool scale to accommodate complex atomic classes and simulation runs.

## 4.2. Composite class interface

As observed in Section 2.3, visual node editing interfaces are a common alternative to textual interfaces for the implementation of composite models. This style of interface is arguably the most prominent instance of visual programming applied in simulation environments, and it serves as a starting point for our own composite class interface. One unusual feature of our interface is that the components are always placed in a cascading vertical arrangement. This layout is shown on the left of Figure 3 for a ''Bank'' model consisting of a lineup for a bank machine, the machine itself, a lineup for a bank teller, and

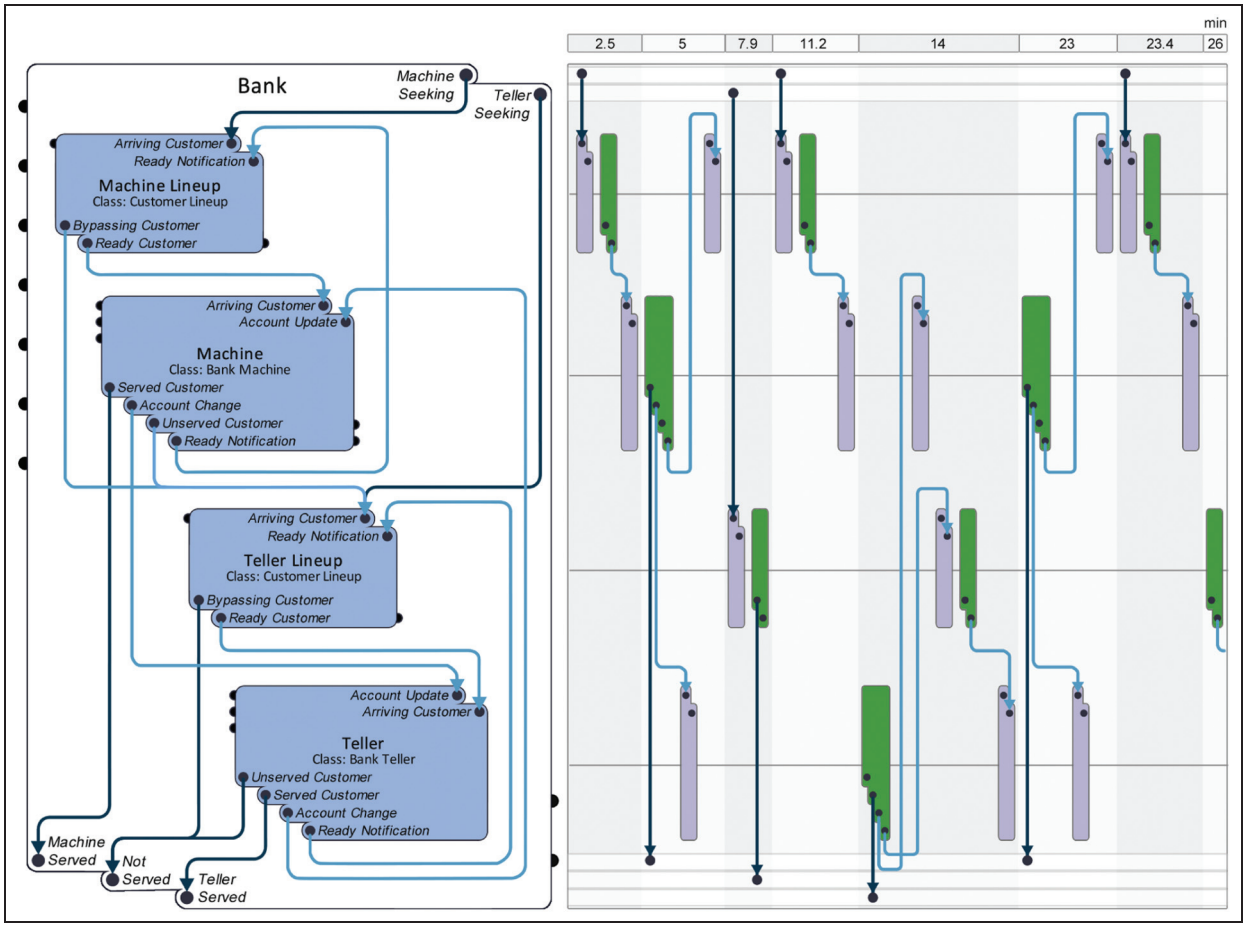


**Figure 3.** The visual interface of the composite class on the left, which is accompanied by the visualization of events on the right in the composite class debugging tool.

the teller itself. The user may reorder the components, in which case the affected links should be re-positioned automatically. Observe that the cascading of the components reduces the number of necessary link turns, lowering diffuseness(cd). Also note that our preference for a familiar top-to-bottom, left-to-right cascade is one of the justifications for placing the simulation input and output ports on the top-right and bottom-left corners. We explain how this vertical layout and other design decisions address the needs of end-user programmers.

**A user needs to understand the relationship between atomic classes, composite classes, and components**. Atomic and composite model specifications differ significantly in structure and appearance. It is therefore expected that a typical DEVS-based tool will provide two very different-looking interfaces: one or more code editors for an atomic class, and a node editor for a composite class. Unfortunately, this inconsistency in visual representation means that users may not necessarily transfer what they learn from one interface to the other. For example, a user may be slow to recognize that both types of classes feature the same types of ports, communicate using the same types of data during the same phases of a simulation run, and can be assigned to components of a composite class.

The similarities between atomic and composite classes can be emphasized wherever appropriate by maximizing consistency(cd) between the visual interfaces of atomic and composite models. First, the outermost compartment that represents an entire class is the same shape for the atomic class in Figure 1(a) and the composite class on the left of Figure 3, with ports placed in the same locations and drawn with the same styles. Observe that the same style of compartment is also used for the four components, suggesting they are associated with classes of either type. Finally, although they were not required by DEVS theory, we included links in the atomic class interface to help users identify the same types of data in the composite class interface. In both interfaces, messages are directed by blue simulation links that connect to ports in a downward direction.

**A user needs to relate the elements of a composite class to events and messages in a simulation run**. Like its atomic counterpart, a composite class consists of static elements representing dynamic behavior, and a user may have difficulty understanding this model-simulation relationship. In the case of a composite class, it is particularly difficult to explain how time advances without showing case-specific sequences of events, as well as the transfer of information from one event to another. Such visualizations are therefore essential for helping end-user programmers learn DEVS, and may also serve as a valuable debugging aid for all users. The challenge for interface designers is to show the sequence and timing of events, the flow of data, and the composite model itself such that all elements can be related to one another.

When a simulation is run, the composite class interface may be accompanied by a timeline of events and messages as illustrated in Figure 3. The vertical layout of components in the class interface allows them to be aligned with the timeline events of the corresponding DEVS instances. This eliminates the need for additional labels on the timeline, and promotes juxtaposibility(cd) with regard to modeling and simulation elements. The ports are also aligned between the class interface and timeline, reducing hard mental operations(cd) in determining which link is responsible for a given transfer of data. For the sake of consistency(cd), the colors of planned and unplanned events in the timeline match those of the corresponding handlers in the atomic class interface.

**A user needs to be mindful of the ordering of simultaneous planned events**. In theory, this is determined by the tie-breaking function *Select*. In practice, most simulators prioritize the component closest to the front of a user-defined list, and the manner in which this list is visually represented can affect the likelihood of a false assumption about which messages may preempt which events. For example, unless the list is an integral part of the interface, the user may simply forget how the components are ordered. A more subtle issue is whether a user understands that simulation links have slightly different roles depending on whether they approach higher- or lower-priority components.

In addition to promoting alignment with the timeline, the vertical layout of our composite class interface implies an ordering of components. The higher a component is located in the diagram, the higher its priority. This concept should be easily understood due to its closeness of mapping(cd) with a user's mental model of organizational hierarchies: higher positions are generally associated with higher priorities. The convention also produces a visual distinction between upward and downward simulation links. This promotes role-expressiveness(cd) once the user understands that only a downward link, which approaches a lower-priority component, is able to preempt an imminent planned event by triggering an unplanned event instead.

**A user needs to manage data dependencies during initialization and finalization**. The DEVS formalism does not provide mathematical elements for initialization and finalization, and we note that a number of specification properties such as legitimacy can be formally analyzed without this information. In DEVS-based tools it is common practice to include parameters and initialization functions in atomic classes, but not composite classes. The reason is that the user is generally permitted to initialize a composite instance by assigning parameters directly to each component. Unfortunately, this functionality may hide dependencies(cd) among component parameters. For example, suppose that both the Bank Machine and Bank Teller atomic classes have an Exchange Rate

parameter. When these classes are assigned to components of the Bank composite class, the need to keep the Exchange Rate parameters consistent imposes an uncomfortable burden on the user.

Our composite class interface can be expanded horizontally to reveal initialization and finalization elements, as illustrated in Figure 4. This view is heavily influenced by dataflow visual programming, which is suitable for the task of initializing component parameters as it reduces `hidden dependencies`$^{(cd)}$ on the parameters of the composite class. These dependencies may take several forms: a one-to-one mapping, as demonstrated by the Machine Lineup Capacity parameter; a one-to-many mapping, as demonstrated by the Exchange Rate parameter; and a computation, as demonstrated by the Accounts initialization node that generates an initial set of customer balances for both the Machine and Teller components. As is typical in dataflow programming, the initialization and finalization links form a directed acyclic graph. Note that the color of these links, their left-to-right orientation at port connections, and the style and placement of the initialization and finalization ports distinguishes parameter- and statistic-related data from message data, promoting `consistency`$^{(cd)}$ with the atomic class interface.

Our target community is accustomed to using dataflow programming to both specify and transform objects in their domain. In dataflow programming, a node represents a function that may be evaluated once values become available on all of its input ports. We include examples of such user-defined functions in both the initialization and finalization steps to demonstrate that users can encapsulate these functions in the places where they will be used. Initialization and finalization functions, such as those represented by the Accounts and Waiting Customers nodes, do not contradict DEVS since they are invoked only before and after the simulation process where time advances. In other words, they are external to the execution of a DEVS-based simulation. We argue that these dataflow programming elements will help users better connect the simulation to the domain models that are the users' primary focus.

We believe there is a strong case for incorporating elements of dataflow visual programming at least for the initialization of component parameters. Providing similar elements for finalization, as shown on the right-hand side of Figure 4, completes what we intend to be a very symmetrical mental model of DEVS. Experts may argue that the processing of statistics belongs in a separate *transducer* class associated with the *experimental frame* as opposed to

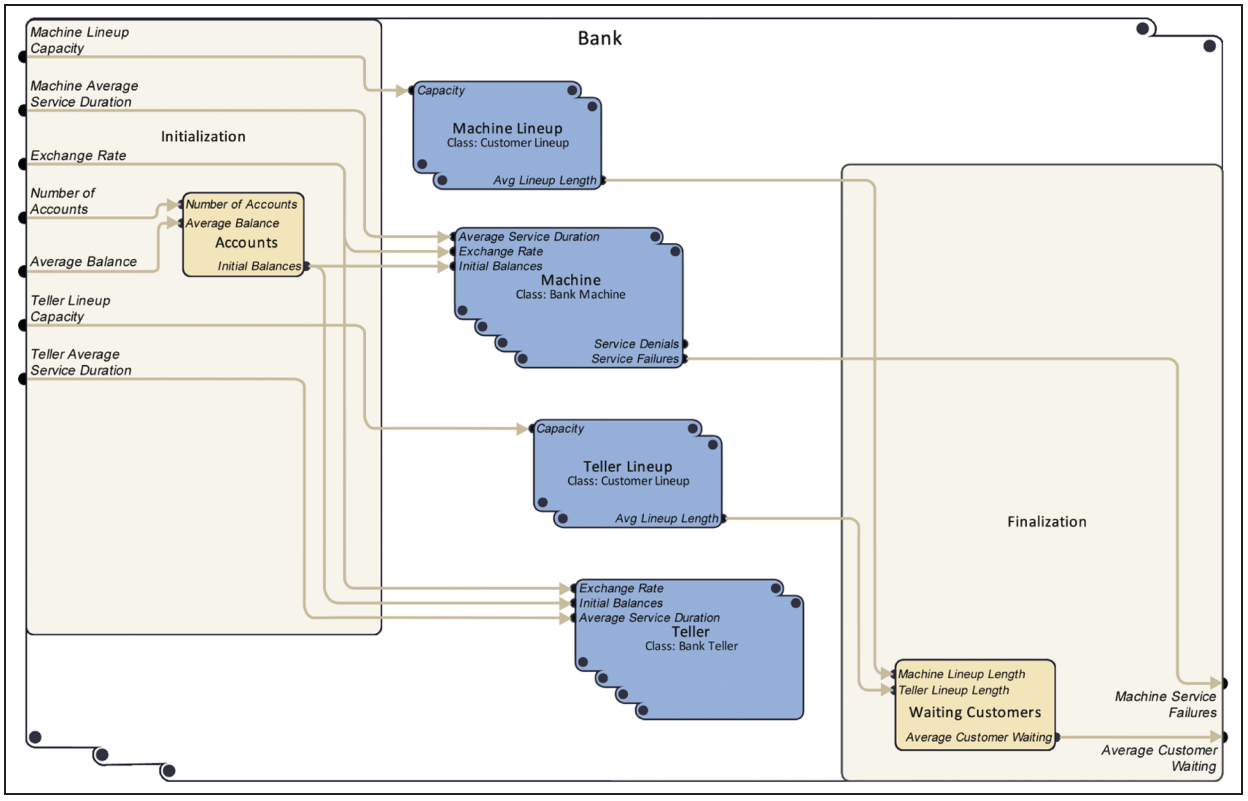


**Figure 4.** The initialization and finalization view of the composite class interface, featuring dataflow programming elements such as the Accounts and Waiting Customers nodes.

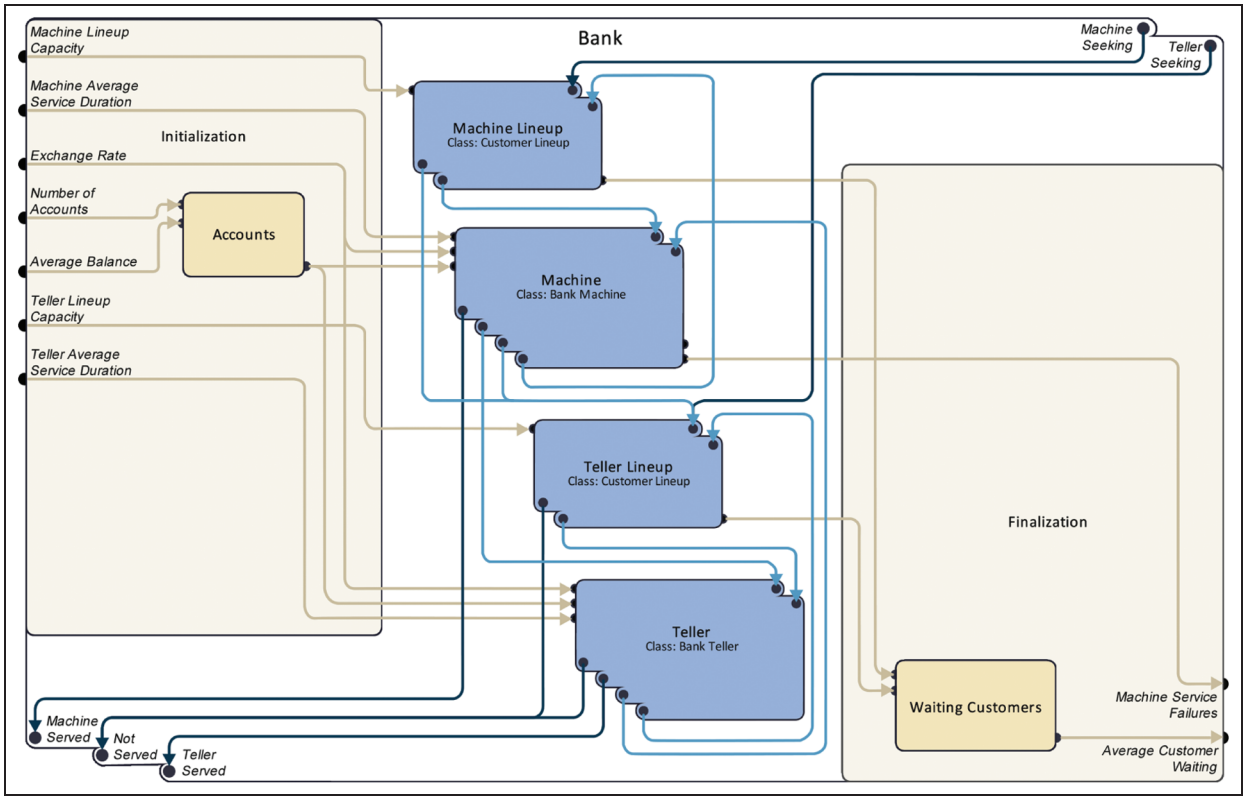


**Figure 5.** The overall view of the composite class interface

the representation of a real-world system. Without taking a firm stand on this issue, we suggest that our finalization elements provide a mechanism for processing statistics in any class. Once this mechanism is understood by the user, it is possible to recommend a set of best practices which may include the consolidation of statistics in transducer classes. For the sake of `role-expressiveness`$^{(cd)}$, we would prefer not to use the term ''transducer'' but rather ''analysis class''.

Figure 5 shows the full view of the composite class, where message links and initialization and finalization elements are overlaid. It is up to the user to choose among the three composite class views based on what data is relevant for the task at hand. We envision an environment that complements these views with an interface that allows the user to manually rearrange the nodes. Manual layouts common to existing simulation tools remain beneficial for two reasons: to minimize link overlap or to use the alternative layout as a form of `secondary notation`$^{(cd)}$. Ultimately, we want to be able to combine the system's automatic layout and the user's manual layout in a meaningful way.

## 5. Discussion and conclusion

We view this paper more as demonstrating an approach to making DEVS easier to learn and use for a potentially large community of users who know much about the phenomenon they want to simulate but little about DEVS, and less as a demonstration of a new interface for DEVS. To this end we have presented both design decisions and rationale, the latter using concepts from end-user programming and the cognitive dimensions of notations.

Any reliable evaluation of our interfaces requires implementation and user studies. Even the names underlying an interface, such as the names proposed in Section 3, lend themselves to systematic investigation once they are embedded in software. One methodology for investigating naming conventions involves the recording and collaborative examination of programming-related conversations, as demonstrated by Katzenberg and Piela.[35] In the results of user studies on vocabulary and visual interfaces, we can expect both confirmation of our approach and evidence of trade-offs that must be made. Cognitive dimensions theory predicts that improvements in one dimension may be linked to problems created for others. In the following, we predict several such trade-off effects.

The presented interfaces will increase overall `diffuseness`$^{(cd)}$ when compared with a standard text editor interface to the simulation code. It is very difficult to improve on the compactness of text, especially when a programmer takes advantage of functions for reusable elements and modern programming language features such as inheritance. On the other hand, these `abstraction`$^{(cd)}$ mechanisms tend to increase `hidden dependencies`$^{(cd)}$, which

should be addressed in a visual interface. Still with the issue of functions and inheritance, the absence of devices for these in the interface will make it less abstract(cd). However, composite classes are an abstraction(cd) mechanism, whose purpose is to improve role-expressiveness(cd) and closeness of mapping(cd), but require devices such as careful link layout to reduce hard mental operations(cd). Of course, further design informed by cognitive dimensions may improve these trade-off situations.

As a consequence of increased diffuseness(cd), our visual interfaces and others like them will not scale easily as DEVS classes grow in size. Atomic classes grow primarily in the internal complexity of their four main user-defined parts: the handlers for initialization, finalization, planned events and unplanned events. However, these structures can and do become complex, requiring additional user-defined functions. Composite models may present larger problems, particularly if they grow by accretion in an iterative process of simulation development. We should expect to see this problem addressed in several ways: through deepening of the model hierarchy (extracting subsets of the components into new composite classes), through *semantic zoom* in which the visualization of a model changes with its size (the smallest possible visualization being a pixel and the largest one that fully displays all parts of a model); model compilation in which models become *black boxes* to a user; and by the brute force method of increasing display size (the cost of which is falling dramatically, but which is limited by human vision and attention constraints).

Further delving into composite classes reveals both benefits and trade-offs in other cognitive dimensions. A composite class comprising only atomic components could quickly become extremely diffuse(cd). Extracting multiple atomic (or composite) components and encapsulating them within a new composite class presents a single concept to a user. This may improve role-expressiveness(cd) in cases where the grouped components collectively perform a well-defined function in the context of a system, or it may promote closeness of mapping(cd) if the new group reflects a familiar real-world object. However, when expressed as a single node, a composite class increases hidden dependencies(cd): what is the relation between incoming and outgoing messages? Also, which component addresses a particular aspect of a simulation? This problem increases sharply when other composite classes occur as components. Fully answering the above questions may involve fully expanding the composite class to reveal all atomic classes, which would dramatically increase diffuseness(cd) and may well decrease role-expressiveness(cd) as the sheer complexity of the display may make it difficult to figure out what individual components actually do. The design presented here is not alone in facing this problem: handling such compositions is one of the primary issues for all visual programming languages. However, the devices of vertical ordering of components and port alignment present a novel approach to this enduring challenge.

Visual programming interfaces present new opportunities for debugging, some of which we take advantage of in our current design and others we leave for future work. For example, juxtaposing(cd) the debugging tool with the simulation interface and aligning their ports and nodes, liveness of the entire interface with highlighting of components across views[43], and the ability to watch the progress through the code line-by-line and through time event-by-event in one interface, all increase visibility(cd) and role-expressiveness(cd) and decrease error-proneness(cd) and hard mental operations(cd) of the system and encourage progressive evaluation(cd). These end-user programming ideas might be combined with emerging research in formalism-based simulation debugging, such as the recent work by Mierlo et al.[44]

We argue that the interface designs presented here may help DEVS tool designers improve the reception of their work. For example, by critically examining established naming conventions from an end-user perspective, designers may improve both learnability and ease of use once learned. By using cognitive dimensions analysis during design, designers may discover not only issues, problems and trade-offs (for example, the trade-off between supporting abstraction(cd) and reducing hidden dependencies(cd) that occurs in composite classes), but also motivate and recognize potentially useful new design elements (such as the component cascade in the composite class interface). By adapting our approach to other modeling formalisms, such as Parallel DEVS, and other paradigms, such as agent-directed simulation, designers may not only improve the user reception of an entire class of tools, but also better reveal commonalities among these diverse approaches. Clearly, we have only touched the surface here. It seems entirely safe to claim that there is plenty of opportunity to make DEVS-based simulation more accessible to non-DEVS end-user programmers who are experts in the domain being modeled.

## Acknowledgements

We would like to thank Jeremy Mogk for his advice on readability of our figures by the colorblind. Of course, any errors in the color scheme are ours alone. Gabriel Wainer provided much useful advice on DEVS, for which we are grateful.

## Funding

This work was supported by the Natural Sciences and Engineering Research Council of Canada under an NSERC Engage Grant.

## References


1. Woodbury R. *Elements of Parametric Design*. London: Taylor and Francis, 2010.
2. Reas C and Fry B. *Processing: a programming handbook for visual designers and artists*. Cambridge, MA: MIT Press, 2007.
3. Ko AJ, Abraham R, Beckwith L, et al. The state of the art in end-user software engineering. *ACM Comput Surv* 2011; 43(3): 21:1–21:44.
4. Davis AL and Keller RM. Data flow program graphs. *Computer* 1982; 15(2): 26–41.
5. Oinn T, Greenwood M, Addis M, et al. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience* 2006; 18(10): 1067–1100.
6. Green TRG. Cognitive dimensions of notations. In *People and Computers V*. Cambridge: Cambridge University Press, pp. 443–460.
7. Ko AJ. Barriers to successful end-user programming. In Burnett MH, Engels G, Myers BA, et al. (eds) *End-User Software Engineering*. Number 07081 in Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.
8. Blackwell AF. First steps in programming: a rationale for attention investment models. In *Proceedings IEEE 2002 symposia on human centric computing languages and environments*, Arlington, VA, pp. 2–10.
9. Burnett MM. Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*. New York: Wiley, 1999.
10. Whitley KN. Visual programming languages and the empirical evidence for and against. *J Visual Languages Comput* 1997; 8(1): 109–142.
11. Johnston WM, Hanna JRP and Millar RJ. Advances in dataflow programming languages. *ACM Comput Surv* 2004; 36: 1–34.
12. Haeberli PE. ConMan: a visual programming language for interactive graphics. In *Proceedings of the 15th annual conference on computer graphics and interactive techniques (SIGGRAPH '88)*. New York: ACM Press, pp. 103–111. DOI:10.1145/54852.378494.
13. Green TRG, Blandford AE, Church L, et al. Cognitive dimensions: Achievements, new directions, and open questions. *J Visual Languages Comput* 2006; 17(4): 328–365.
14. Jones SP, Blackwell A and Burnett M. A user-centred approach to functions in excel. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, Uppsala, Sweden. New York: ACM Press, pp. 165–176.
15. Green TRG and Petre M. Usability analysis of visual programming environments: A 'Cognitive dimensions' framework. *J Visual Languages Comput* 1996; 7(2): 131–174.
16. Blackwell AF and Green TRG. A cognitive dimensions questionnaire optimised for users. In *12th workshop of the psychology of programming interest group*.
17. Zeigler BP, Praehofer H and Kim TG. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. San Diego, CA: Academic Press, 2000.
18. Vangheluwe HLM. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE international symposium on computer-aided control system design, 2000 (CACSD 2000)*, pp. 129–134.
19. Wymore AW. *Mathematical Theory of Systems Engineering*, 1st edn. New York: John Wiley & Sons Ltd, 1967.
20. Wainer GA. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca Raton, FL: CRC Press, 2008.
21. Hwang MH. DEVS++: C++ open source library of DEVS formalism, 2009. Available at: http://odevspp.sourceforge.net/
22. Goldstein R, Breslav S and Khan A. Informal DEVS conventions motivated by practical considerations (WIP). In *Proceedings of the symposium on theory of modeling & simulation*. San Diego, CA: Society for Computer Simulation International, pp. 10:1–10:6.
23. Chow AC and Zeigler BP. Parallel DEVS: a parallel, hierarchical, modular modeling formalism. In *Proceedings of the 1994 Winter simulation conference (WSC)* pp. 716–722.
24. Traoré MK. Easy DEVS. In *Proceedings of the 2007 Spring simulation multiconference (SpringSim '07)*, Vol. 2. San Diego, CA: Society for Computer Simulation International, pp. 214–216.
25. Mooney J and Sarjoughian H. A framework for executable UML models. In *Proceedings of the 2009 Spring simulation multiconference (SpringSim '09)*. San Diego, CA: Society for Computer Simulation International, pp. 160:1–160:8.
26. Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87(1–2): 113–132.
27. Bonaventura M, Wainer GA and Castro R. Graphical modeling and simulation of discrete-event systems with CD++Builder. *Simulation* 2013; 89(1): 4–27.
28. Quesnel G, Duboz R and Ramat E. The Virtual Laboratory Environment – An operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Sim Modell Practice Theory* 2009; 17(4): 641–653.
29. MathWorks. *Simulink: Dynamic System Simulation for MATLAB*, 2000.
30. Scilab Enterprises. Xcos for very beginners, 2013.
31. Ptolemaeus C (ed.) *System Design, Modeling, and Simulation using Ptolemy II*, 2014. Available at: `http://ptomlemy.org/systems`.
32. Brück D, Elmqvist H, Mattsson SE, et al. Dymola for multi-engineering modeling and simulation. In *International Modelica conference*.
33. Varga A and Hornig R. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems and Workshops (SIMUTools '08)*. Brussels: ICST, 2008, pp. 60:1–60:10.
34. Seo C, Zeigler BP, Coop R, et al. DEVS modeling and simulation methodology with MS4 Me software tool. In *Proceedings of the symposium on theory of modeling & simulation*. San Diego, CA: Society for Computer Simulation International, pp. 33:1–33:8.
35. Katzenberg B and Piela P. Work language analysis and the naming problem. *Commun ACM* 1993; 36(6): 86–92.

36. Sarjoughian HS and Elamvazhuthi V. CoSMoS: a visual environment for component-based modeling, experimental design, and simulation. In *Proceedings of the international conference on simulation tools and techniques (SIMUTools)*. Brussels: ICST, 2009, pp. 59:1–59:9.
37. Gamma E, Helm R, Johnson R, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
38. Zeigler BP and Sarjoughian HS. Introduction to DEVS modeling and simulation with JAVA: developing component-based simulation models. *Technical Document*, University of Arizona, 2003.
39. Ko AJ, Myers BA and Aung HH. Six learning barriers in end-user programming systems. In *Proceedings of IEEE symposium on visual languages and human centric computing*, pp. 199–206.
40. Zengin A and Sarjoughian H. DEVS-Suite simulator: A tool teaching network protocols. In *Proceedings of the 2010 Winter simulation conference (WSC)*, pp. 2947–2957.
41. Lewis B. Debugging backwards in time. arXiv preprint cs/0310016, 2003.
42. Dachselt R, Frisch M and Weiland M. FacetZoom: a continuous multi-scale widget for navigating hierarchical metadata. In *Proceedings of the SIGCHI conference on human factors in computing systems (CHI'08)*. New York: ACM Press, 2008, pp. 1353–1356.
43. Maleki MM, Woodbury RF and Neustaedter C. Liveness, localization and lookahead: Interaction elements for parametric design. In *Proceedings of the 2014 conference on designing interactive systems (DIS'14)*, pp. 805–814.
44. Mierlo SV, Tendeloo YV, Mustafiz S, et al. Explicit modelling of a Parallel DEVS experimentation environment. In *Proceedings of the symposium on theory of modeling and simulation*. Alexandria, VA: Society for Computer Simulation International.

### Author biographies

**Maryam Maleki** is a recent PhD graduate in human–computer interaction from Simon Fraser University. She is a user experience researcher and designer and is interested in end-user computing in expert systems, design support tools, and computational design.

**Robert Woodbury** is a University Professor of Interactive Arts and Technology at Simon Fraser University. He was founding Chair of the Graduate Program in the School of Interactive Arts and Technology at SFU and a Director of two pan-Canadian research networks. His main interest is augmenting professional work by supporting state-space search in problem spaces. To parametric design he has contributed a book, *Elements of Parametric Design*, and numerous articles.

**Rhys Goldstein** is a Principal Research Scientist at Autodesk Research. His work explores discrete-event simulation and its use in the architectural, building science, and biological domains. He received the Best Paper award at the 2010 IBPSA-USA Conference for a method to customize simulated human behavior.

**Simon Breslav** is a Principal Research Scientist at Autodesk Research. Simon completed a MSc in computer science (specializing in computer graphics) from the University of Toronto. His current research interests include information visualization, human–computer interaction, user interfaces for simulation, and simulation in the environmental and ergonomics contexts.

**Azam Khan** is the Head of the Environment and Ergonomics Research Group at Autodesk Research. He has published over 60 papers in simulation-aided design, human–computer interaction, and sustainability. He has also founded several inter-organizational initiatives such as the Parametric Human Project Consortium, the Symposium on Simulation for Architecture and Urban Design (SimAUD), and the CHI Sustainability Community.

## Appendix

With the objective of promoting DEVS utilization in domains outside of simulation research and systems engineering, the Introduction outlined the following strategy. *First, change the way DEVS theory is expressed to make it easier to learn from a user's perspective. Second, make software consistent with the re-expressed DEVS theory. Third, design visual interfaces that emphasize and relate the key DEVS concepts associated with every element of the users' models and simulation runs*. Having pursued this strategy throughout the paper, we find ourselves with an adapted version of DEVS. Here we show that this adaptation remains essentially equivalent to the classic version of the formalism.

We begin by comparing the simulation procedures associated with Classic DEVS, the mathematical elements of which are listed in Section 2.2, and our adapted version of DEVS, illustrated in Figure 1(a). To simplify the comparison, we omit the classic sets $X$, $Y$, $S$ that contain the permissible values of input messages, output messages, and instance states. These sets provide both a form of documentation and a degree of reliability. However, their realization in DEVS-based software depends on the programming language used, which is beyond the scope of this paper. We therefore focus on the four classic functions $\delta_{ext}$, $\delta_{int}$, $\lambda$, and $ta$, as well as the four proposed event handlers that take the place of these functions.

The pseudocode listed in Algorithm 1 describes the basic simulation procedure for Classic DEVS. The model and initial state $s_0$ are known at the outset. We assume that all input messages are initially available in the form of a time series $ts_{in}$ containing time ($t$), port (*port*), value (*msg*) entries of the

**Algorithm 1.** Classic DEVS simulation

```
 1: function SIMULATE(model, s_0, ts_in, t_end)
 2:     δ_ext, δ_int, λ, ta ← model
 3:     s ← s_0                          {current state}
 4:     t ← 0                            {current time}
 5:     Δt_e = 0                         {elapsed duration}
 6:     i_in ← 0                         {input time series index}
 7:     n_in ← #ts_in                    {input time series size}
 8:     ts_out ← [ ]                     {output time series}
 9:     done ← ⊥                         {termination flag}
10:     while ¬done do                   {simulation loop}
11:         t_ext, x ← ∞, ∅
12:         if i_in < n_in then          {more messages?}
13:             t_ext, x ← ts_in[i_in]
14:         end if
15:         t_int ← t + ta(s)
16:         t_prev ← t
17:         t ← min(t_ext, t_int)        {time advancement}
18:         Δt_e ← t − t_prev
19:         if t ≥ t_end then            {finished?}
20:             done = ⊤
21:         else if t = t_ext then       {external event?}
22:             s ← δ_ext(s, Δt_e, x)
23:             i_in ← i_in + 1
24:         else if t = t_int then       {internal event?}
25:             y, s ← λ(s), δ_int(s)
26:             if y ≠ ∅ then
27:                 ts_out ← ts_out ‖ [ [t, y] ]
28:             end if
29:         end if
30:     end while
31:     return ts_out, s, Δt_e
32: end function
```

**Algorithm 2.** Adapted DEVS simulation

```
 1: function SIMULATE(model, vals_in, ts_in, t_end)
 2:     f_init, f_u, f_p, f_final ← model
 3:     s, Δt_p ← f_init(vals_in)        {initialization event}
 4:     t ← 0                            {current time}
 5:     Δt_e = 0                         {elapsed duration}
 6:     i_in ← 0                         {input time series index}
 7:     n_in ← #ts_in                    {input time series size}
 8:     ts_out ← [ ]                     {output time series}
 9:     done ← ⊥                         {termination flag}
10:     while ¬done do                   {simulation loop}
11:         t_u, x ← ∞, ∅
12:         if i_in < n_in then          {more messages?}
13:             t_u, x ← ts_in[i_in]
14:         end if
15:         t_p ← t + Δt_p
16:         t_prev ← t
17:         t ← min(t_u, t_p)            {time advancement}
18:         Δt_e ← t − t_prev
19:         if t ≥ t_end then            {finished?}
20:             done = ⊤
21:         else if t = t_u then         {unplanned event?}
22:             s, Δt_p ← f_u(s, Δt_e, x)
23:             i_in ← i_in + 1
24:         else if t = t_p then         {planned event?}
25:             s, Δt_p, list_y ← f_p(s, Δt_e)
26:             ts_out ← ts_out ‖ [ [t, list_y] ]
27:         end if
28:     end while
29:     vals_out ← f_final(s, Δt_e)      {finalization event}
30:     return ts_out, vals_out
31: end function
```

form [*t*, [*port*, *msg*]]. In contexts involving interactivity or real-time control, these messages become available only as the simulation is underway; nevertheless, the pseudocode provides useful guidance as to how the messages should influence a simulation. We also assume that a simulated time point $t_{\text{end}}$ is supplied, and that the simulation ends as soon as the current time $t$ reaches this point. Other terminating mechanisms can be used with minimal change to the overall procedure. The result of the algorithm is the time series of output messages $ts_{\text{out}}$, the final state $s$, and the time elapsed in the final state $\Delta t_e$. One may modify the algorithm to also record some or all of the intermediate states that occur as the simulation progresses. It is remarkable that, given suitable definitions for the functions $\delta_{\text{ext}}$, $\delta_{\text{int}}$, $\lambda$, and $ta$, this algorithm can describe essentially any simulation run regardless of its domain or the timing patterns that emerge between its events.

As with most computer programs, certain manipulations can be performed with minimal or zero effect on the nature of the algorithm. An example of a minimal change is the encapsulation of state within the simulation procedure. Instead of requiring the initial state $s_0$ directly, we require parameter values $vals_{\text{in}}$ from which the initial state can be computed. It is actually common practice to obtain initial states in this fashion. The parameter-to-state transformation is performed by the initialization event handler, which we represent using the mathematical function $f_{\text{init}}$. To fully encapsulate state, an analogous change must be made at the end of the algorithm. Instead of delivering the final state $s$ and corresponding elapsed duration $\Delta t_e$, these values are supplied to the finalization event handler represented by $f_{\text{final}}$. This yields the statistics values $vals_{\text{out}}$, which are delivered along with the output time series as the simulation results.

An example of a program manipulation with no semantic effect whatsoever is the relocation of the computation $ta(s)$. In Algorithm 1, the computation is found on line 15. Nothing is changed if we instead apply $ta$ immediately after every state change; in other words, we extract $ta(s)$

from line 15 and insert it after lines 3, 22, and 25. Line 3 is where we will invoke the initialization event handler $f_{\text{init}}$, and this function can absorb the $ta(s)$ computation. Line 22 is where the unplanned event handler $f_u$ will be invoked, and again it will absorb $ta(s)$. Finally, line 25 will become part of the planned event handler $f_p$, along with the $ta(s)$ computation. Note that wherever $ta(s)$ is computed, the result can be stored in a planned duration variable $\Delta t_p$ so that the value can be still be used on line 15.

These largely superficial changes result in Algorithm 2, the simulation procedure for the adapted version of DEVS involving the four event handlers $f_{\text{init}}$, $f_u$, $f_p$, and $f_{\text{final}}$. For all intents and purposes, the algorithms are the same, and accordingly the argument *model* can be interpreted as a DEVS model in both cases. The difference is that Algorithm 2 better supports the paper's visual interfaces designed for end-user programmers.

We now explore the question of whether the visual interfaces are appropriate only for end-user programmers, or if they also support the needs of those who prefer to begin with DEVS model specifications in the conventional form. It turns out that if one has defined $\delta_{\text{ext}}$, $\delta_{\text{int}}$, $\lambda$, and $ta$, there is a simple mapping of these functions onto $f_{\text{init}}$, $f_u$, $f_p$, and $f_{\text{final}}$.

$$f_{\text{init}}(vals_{\text{in}}) = s_0, \Delta t_p \quad \text{where} \quad \Delta t_p = ta(s_0)$$

$$f_u(s, \Delta t_e, x) = s', \Delta t_p \quad \text{where} \quad s' = \delta_{\text{ext}}(s, \Delta t_e, x), \; \Delta t_p = ta(s')$$

$$f_p(s, \Delta t_e) = s', \Delta t_p, list_y \quad \text{where} \quad s' = \delta_{\text{int}}(s), \; \Delta t_p = ta(s'), \; list_y = [\lambda(s)]$$

$$f_{\text{final}}(s, \Delta t_e) = vals_{\text{out}}$$

The mapping is efficient in that, when the above definitions of $f_{\text{init}}$, $f_u$, $f_p$, and $f_{\text{final}}$ are substituted into Algorithm 2, there are no redundant invocations of $\delta_{\text{ext}}$, $\delta_{\text{int}}$, $\lambda$, or $ta$. A DEVS expert could therefore specify his/her model in the traditional mathematical form, yet implement it using a simulation tool featuring the proposed visual interfaces.

To complete the comparison of the classic and adapted versions of DEVS, there is one remaining difference to explore. In Classic DEVS, each internal event is associated with exactly zero or one output message (in Algorithm 1, line 25, see $y$ which may be $\emptyset$ to indicate no message). In the adaptation, where there are ''planned events'' instead of ''internal events'', each planned event is associated with a list of zero or more output messages (see $list_y$ on line 25 of Algorithm 2). This departure from Classic DEVS is not critical to our work; in other words, our visual interface designs support either one optional message or a list of messages. If a message list is to be supported, a small change must be made in the simulation of composite models.

An elegant way to describe simulations involving composite models is as follows. First, regard any composite model as an atomic model known as the *resultant*, which is possible thanks to the concept known as closure under coupling. Second, because a composite model is essentially a special case of an atomic model, Algorithms 1 and 2 can be used to simulate any model. To apply this technique to Classic DEVS, we must define the resultant functions $\delta_{\text{ext}}$, $\delta_{\text{int}}$, $\lambda$, $ta$ using the composite model elements below

| | |
|---|---|
| $[\ldots, [\langle\delta_{\text{ext}}\rangle_d, \langle\delta_{\text{int}}\rangle_d, \lambda_d, ta_d], \ldots]$ | {components} |
| *EIC* | {external input couplings} |
| *EOC* | {external output couplings} |
| *IC* | {internal couplings} |
| *Select* | {tie-breaking function} |

Let us focus specifically on the resultant functions RESULTANT_LAMBDA($\lambda$) and RESULTANT_DELTA_INT($\delta_{\text{int}}$), both associated with internal events. The pseudocode is given in Algorithm 3. Both functions select the event-triggering component using the same procedure, which is encapsulated in the CALL_SELECT function.

The pseudocode for the resultant planned event handler is given in Algorithm 4, and the procedure is similar to that of the internal events in Algorithm 3. The first step in both cases is to determine the index $d^*$ of the event-triggering component. In Algorithm 3 this is done by the CALL_SELECT function, whereas in Algorithm 4 the corresponding code appears on lines 3 through 10. The Classic function RESULTANT_LAMBDA contains a **for** loop on line 19, which tracks messages from the $d^*$ component to the output ports of the composite model; the same loop appears on line 16 of Algorithm 4. The Classic function RESULTANT_DELTA_INT contains a **for** loop on line 38, which tracks messages from the $d^*$ component to the input ports of other components; the same loop appears on line 21 of Algorithm 4. The RESULTANT_DELTA_INT function is also responsible for updating the state of the $d^*$ component (line 34) and affected components (line 41); the corresponding instructions can be found on lines 11 and 24 of Algorithm 4.

For the version of DEVS adapted for visual interfaces, composite models include the following elements.

| | |
|---|---|
| $[\ldots, [\langle f_{\text{init}}\rangle_d, \langle f_u\rangle_d, \langle f_p\rangle_d, \langle f_{\text{final}}\rangle_d], \ldots]$ | {components} |
| *EIC* | {external input couplings} |
| *EOC* | {external output couplings} |
| *IC* | {internal couplings} |

Though largely similar, there are a few notable differences between Algorithms 3 and 4. For the code at the beginning that selects the event-triggering component, the main semantic difference is that Classic DEVS relies on a *Select* function whereas the adapted version orders events by component index. Most DEVS-based tools feature this

**Algorithm 3.** Resultant internal event

```
 1: function CALL_SELECT(s)
 2:     [..., [s_d, ⟨Δt_e⟩_d], ...] ← s
 3:     Δt_e ← min(..., ta_d(s_d) − ⟨Δt_e⟩_d, ...)
 4:     imminent ← [ ]
 5:     for d : [0, ..., #s − 1] do
 6:         if ta_d(s_d) − ⟨Δt_e⟩_d = Δt_e then
 7:             imminent ← imminent ‖ [d]
 8:         end if
 9:     end for
10:     return Select(imminent), Δt_e
11: end function
12: function RESULTANT_LAMBDA(s)
13:     [..., [s_d, ⟨Δt_e⟩_d], ...] ← s
14:     d*, Δt_e ← CALL_SELECT(s)
15:     y ← ∅
16:     y_{d*} ← λ_{d*}(s_{d*})
17:     if y_{d*} ≠ ∅ then
18:         port_{d*}, msg ← y_{d*}
19:         for [d†, port_{d†}], port : EOC do
20:             if d† = d* ∧ port_{d†} = port_{d*} then
21:                 y ← [port, msg]
22:             end if
23:         end for
24:     end if
25:     return y
26: end function
27: function RESULTANT_DELTA_INT(s)
28:     [..., [s_d, ⟨Δt_e⟩_d], ...] ← s
29:     d*, Δt_e ← CALL_SELECT(s)
30:     for d : [0, ..., #s − 1] do
31:         ⟨Δt_e⟩_d ← ⟨Δt_e⟩_d + Δt_e
32:     end for
33:     y_{d*} ← λ_{d*}(s_{d*})
34:     s_{d*} ← ⟨δ_int⟩_{d*}(s_{d*})
35:     ⟨Δt_e⟩_{d*} ← 0
36:     if y_{d*} ≠ ∅ then
37:         port_{d*}, msg ← y_{d*}
38:         for [d†, port_{d†}], [d, port_d] : IC do
39:             if d† = d* ∧ port_{d†} = port_{d*} then
40:                 x_d ← [port_d, msg]
41:                 s_d ← ⟨δ_ext⟩_d(s_d, ⟨Δt_e⟩_d, x_d)
42:                 ⟨Δt_e⟩_d ← 0
43:             end if
44:         end for
45:     end if
46:     return [..., [s_d, ⟨Δt_e⟩_d], ...]
47: end function
```

**Algorithm 4.** Resultant internal event

```
 1: function RESULTANT_F_P(s, Δt_e)
 2:     [..., [s_d, ⟨Δt_e⟩_d, ⟨Δt_p⟩_d], ...] ← s
 3:     d* ← ∅
 4:     for d : [0, ..., #s − 1] do
 5:         ⟨Δt_e⟩_d ← ⟨Δt_e⟩_d + Δt_e
 6:         ⟨Δt_p⟩_d ← ⟨Δt_p⟩_d − Δt_e
 7:         if d* = ∅ ∧ ⟨Δt_p⟩_d = 0 then
 8:             d* ← d
 9:         end if
10:     end for
11:     s_{d*}, ⟨Δt_p⟩_{d*}, ⟨list_y⟩_{d*} ← ⟨f_p⟩_{d*}(s_{d*}, ⟨Δt_e⟩_{d*})
12:     ⟨Δt_e⟩_{d*} ← 0
13:     list_y ← [ ]
14:     for y_{d*} : ⟨list_y⟩_{d*} do
15:         port_{d*}, msg ← y_{d*}
16:         for [d†, port_{d†}], port : EOC do
17:             if d† = d* ∧ port_{d†} = port_{d*} then
18:                 list_y ← list_y ‖ [ [port, msg] ]
19:             end if
20:         end for
21:         for [d†, port_{d†}], [d, port_d] : IC do
22:             if d† = d* ∧ port_{d†} = port_{d*} then
23:                 x_d ← [port_d, msg]
24:                 s_d, ⟨Δt_p⟩_d ← ⟨f_u⟩_d(s_d, ⟨Δt_e⟩_d, x_d)
25:                 ⟨Δt_e⟩_d ← 0
26:             end if
27:         end for
28:     end for
29:     Δt_p ← min(..., ⟨Δt_p⟩_d, ...)
30:     return [..., [s_d, ⟨Δt_e⟩_d, ⟨Δt_p⟩_d] ...], Δt_p
31: end function
```

simplification. Another meaningful difference is that, whereas the Classic DEVS code handles one potentially empty output $y_{d^*}$, the adaptation code deals with a list of messages $\langle list_y \rangle_{d^*}$. Accordingly, the **if** statements on lines 17 and 36 of Algorithm 3 are replaced with the **for** loop on line 14 of Algorithm 4. A final difference is that whereas both versions include component states $s_d$ and component elapsed durations $\langle \Delta t_e \rangle_d$ in the state $s$, Algorithm 4 also includes component planned durations $\langle \Delta t_p \rangle_d$. Regardless of how DEVS is adapted, these planned durations should be stored in one form or another to avoid redundant invocations of the $ta_d$ functions. Arguably, the storage of $\langle \Delta t_p \rangle_d$ values makes Algorithm 4 easier to implement in an efficient manner.

In summary, the adapted version of DEVS remains similar to the original version of the formalism. There is a direct mapping from Classic DEVS elements to the newly proposed event handlers, and the corresponding simulation algorithms are semantically equivalent aside from a few small differences. Minor departures from the theory can be found in all DEVS-based tools: a result of the fundamental limitations of computer technology.