A Highly Extensible Simulation Framework for Domain-Specific Architectures

George Edwards and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{gedwards,neno}@usc.edu

Abstract

Discrete event simulation is a powerful and flexible mechanism for analyzing both the functional characteristics and quality properties of software design models. However, software engineers do not utilize simulations in many situations where it has significant potential benefits because (1) the programming model used by existing discrete event simulators is not well-suited for capturing domain-specific software designs, and (2) customizing and optimizing core simulation algorithms based on domain-specific requirements and constraints is difficult and, in some cases, not possible. In this paper, we describe a new approach to constructing discrete event simulations of software systems that removes both of these obstacles. First, our simulation approach supports a software architecture-based programming model and allows software engineers to define domain-specific modeling constructs. Second, our simulation approach allows domain-specific optimization of simulation engine functions through modification or replacement of core framework components. To demonstrate the implementation of our approach, we present XDEVS, is a highly extensible discrete event simulation framework that provides both of the above capabilities. We evaluated the utility and efficiency of XDEVS using a large-scale simulation of a volunteer computing system, and this evaluation confirmed that XDEVS represents an improvement over the current state-of-the-art.

1. Introduction

Organizations that develop software-intensive systems with rigorous quality requirements, such as aerospace and defense systems, sensor network applications, and enterprise services, often employ automated system analysis tools to assess key quality criteria, such as dependability and performance. While program analysis techniques that examine system code are extremely useful, they can only be applied once a system has been implemented. At this late stage of development, it is frequently difficult and costly to change the fundamental design decisions made during the early stages of development [2]. These design decisions — such as the basic organization of functionality into components and the interaction patterns used among components — have a profound impact on system quality and are commonly referred to as the system's *architecture*. System analysis techniques that operate on design models, on the other hand, can be leveraged prior to system implementation, when changing architectural decisions is relatively easy. For this reason, industry adoption of software design modeling and analysis technologies is accelerating [7].

Discrete event simulation is a highly flexible technique for analyzing the behavior of complex systems [1]. A discrete event simulation consists of a state-based model, which represents the behavior of the system, and a discrete event simulation engine, which executes the model. Scientists and engineers have used discrete event simulations for a wide range of applications, such as studying biological and chemical systems [8], evaluating military tactics, and planning transportation facilities. In computer science, discrete event simulations have been used extensively in the design of network protocols and hardware architectures.

Discrete event simulation is also a promising approach to evaluating software design models. First, discrete event systems can flexibly simulate the behavior of a variety of other types of formal models that are commonly used to represent the behavior of software systems, such as Markov models, queuing networks, and Petri nets. Since numerous techniques exist for analyzing these types of models with respect to quality metrics, simulation supports a range of different types of analysis. Second, simulation is less computationally-intensive than mathematical methods that solve a model analytically or search the state space of the model exhaustively. As a result, simulation can be applied to larger, more complex systems.

The construction of discrete event simulations of software systems can be difficult, however. Specifically, current discrete event simulation technologies have two critical shortcomings:

1) **Programming model.** The discrete event modeling formalism only defines, and most discrete event simulation engines only support, a fixed set of primitive modeling types. The programming model used to describe the behavior of these types is not well-suited for describing software architectures. Consequently, mapping software architecture models that are expressed using high-level, domain-specific design abstractions to discrete event models requires implementing a complex model trans-

formation from one language to the other. Analysis engineers want simulation engines that natively handle the naturally occurring abstractions in the target domain.

2) **Customization and optimization.** To realize the full potential of a discrete event simulation, it is frequently necessary to customize the simulation engine based on the requirements and constraints of the target domain. In particular, optimizations that take advantage of domain knowledge to improve the efficiency and scalability of the simulation often require modifications to core simulation engine functions, such as scheduling resources and routing messages. Off-the-shelf discrete event simulators are not intended to be modified by the end-user and are difficult to customize.

In this paper, we describe a new approach to constructing discrete event simulations of software systems that addresses the above challenges. The central strategy underlying our approach is to create an extensible and modular simulation framework that:

1) supports a software architecture-based programming model as well as the addition of new, domain-specific modeling constructs to the simulation modeling language through a language extension mechanism, and
2) allows domain-specific optimization of simulation engine functions through modification or replacement of core framework components.

The contributions of this paper are therefore:

- codification of design principles and solutions for model-driven simulation frameworks, including a characterization of three distinct types of framework extensibility, and
- a new simulation framework, called XDEVS, for domain-specific architectures that implements our approach, including support for all three types of extensibility.

In order to demonstrate and evaluate the utility of the XDEVS simulation framework, we designed XDEVS to be an integral component of XTEAM [4], our model-driven design, analysis, and synthesis tool-chain. XTEAM provides facilities for constructing domain-specific languages and models that automatically embed partial semantics within domain-specific types. As we show in this paper, this unique feature of XTEAM models is leveraged by XDEVS to drastically decrease the effort required by engineers to create simulations of domain-specific architectural models. While the seamless integration of XTEAM and XDEVS brings together two complementary sets of features, we also believe XDEVS stands on its own as a novel and useful platform.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of the key aspects of discrete event simulation. Section 3 highlights the important differences between the approach described in this paper and related work in discrete event simulation. In Section 4, we describe the design of the XDEVS simulation framework, including its support for an architecture-based programming model and customization of core framework components.

2. Background

A discrete event simulation consists of a *network* of *nodes* that interact by exchanging *messages*. A *simple* node is an active entity with state, while a *compound* node is an instantiation of a group of simple and compound nodes and their interconnections. Compound nodes can be constructed in a hierarchy of arbitrary depth. Nodes send and receive messages via *ports* which are attached to the ports of other nodes via *links*. Compound nodes transparently pass messages between their internal nodes and external nodes by mapping their external ports to the ports of internal nodes. Every node state transition and message exchange is an *event* that occurs at a discrete time step during the simulation execution.

When applied to software systems, the nodes in a discrete event simulation correspond to computational resources that request services from each other. Each node queues requests, spends some time processing each request, updates its state, and produces output. The behavior of nodes, such as the frequency with which they request services and the amount of time they spend processing requests, is often modeled stochastically. The emergent behavior of the system results from resource contention, node interaction patterns, and other factors.

Discrete event simulation engines may be designed in a variety of ways, but always include a core set of capabilities. A *scheduler* determines the order in which events occur. Whenever a node in the simulation sends a message, a *router* delivers it to the intended recipients. A *handler* determines what state transitions or new messages result from each event. Discrete event simulators often also include random number generators and statistical utilities for conveniently creating stochastic behavior and analyzing simulation data.

One of the most important features of discrete event simulations is their flexibility. Discrete event simulations have been used to analyze the performance, dependability, resource consumption, and other properties of software systems. Discrete event simulations can also model highly dynamic systems with entities that come and go and adapt their behavior based on their environment. Finally, discrete event simulations can be combined with models of continuous systems, such as the physical environment, to produce hybrid models with both discrete and continuous elements.

3. Related Work

A wide variety of commercial and academic discrete event simulators are available. Existing simulators differ from the XDEVS simulation framework we describe in this paper in two important ways: the *programming model* used to describe systems and the *customizability* of the simulation engine.

3.1. Programming Models

Existing simulators require engineers to use either a *routine-based* or *event-based* programming model for describing the

behavior of simple nodes. In a routine-based model, the logic describing the behavior of each node runs in its own thread which is scheduled by the simulation engine. The node receives input and output by invoking *send* and *receive* methods. In an event-based model, logic than handles input messages is invoked by the simulation engine whenever a message arrives. The input-handling logic produces output by invoking a send method.

While the routine-based and event-based programming models are simple and intuitive for describing many types of systems, they are not well-suited for describing complex software architectures because...

As we describe in detail in Section 4.1, the XDEVS simulation framework uses a software architecture-based programming model. XDEVS still executes a model composed of a hierarchy of simple and compound nodes, preserving the properties and features of the discrete event formalism, but these types only exist “under the hood” of the simulation. Engineers describe their systems in terms of components, interfaces, shared resources, and other architectural types. The interfaces of system services, the logic that describes how those services are implemented, and the resources that execute that logic are totally decoupled.

3.2. Customizability

Existing simulators permit only limited types of customization to the simulation infrastructure. Commercial simulators, such as OPNET [] and Simulink [3], are closed-source and, while plug-in interfaces are available to add on certain types of functionality, the core simulation algorithms cannot be modified. Open-source simulators can, in theory, be modified arbitrarily, but they are not designed (or documented) for easy extensibility or customization. Open-source simulators, such as adevs [6] and OMNeT++ [9], provide a simulation kernel in which functionality for scheduling, routing, dispatching, and event handling is tightly coupled. The rationale behind this design is to improve performance, but the result is that it is difficult to modify or replace individual simulator services or reuse custom algorithms across multiple systems.

In Section 4.2, we show how the XDEVS simulation framework provides independent components and interfaces for different simulation services. The structure of the core framework components and their interactions are based on design patterns that have been successfully applied to other types of application frameworks. A very small simulation kernel only implements sufficient functionality to ensure that the constraints of the discrete event formalism are enforced. Engineers are able to modify the core framework components (without needing to consider possible side-effects), mix-and-match simulation algorithms, and reuse custom-built components within different projects. This architecture can actually improve performance in many cases because the simulation algorithms can be customized to take advantage of opportunities for domain-specific optimization.

4. XDEVS Simulation Framework Architecture

In order to effectively apply discrete event simulations to software design models and address the above discussed deficiencies of existing solutions, an architecture-based *programming model* and *customizability* of a simulation engine are essential. A programming model tailored to a given domain (in our case, software architecture) will provide the types of abstractions needed to describe simply and intuitively the targeted entities, their inter-relationships, and the behaviors associated with them; by contrast, a generic programming model will require constant, likely complex transformations between the simulation and domain entities. Likewise, a customizable simulation engine can leverage the assumptions and constraints of the target domain to improve the performance of the simulation and scale to large, complex systems; by contrast, discrete event simulations that fix the scheduling, routing, and event handling algorithms are very likely to result in suboptimal, “one size fits all” solutions. In order to address these two shortcomings of existing discrete event simulation solutions, we have formulated three overarching objectives a simulation framework should satisfy:

1) ensuring separation of modeling and analysis concerns,
2) ensuring availability of appropriate abstractions, and
3) ensuring enforcement of fundamental relationships.

In this section, we describe the architecture of XDEVS, an extensible and modular simulation framework that addresses the two major shortcomings of existing solutions and meets the three key objectives stated above. A high-level view of XDEVS is shown in Figure 1. XDEVS is explicitly designed to enable straightforward integration with software architecture and design modeling environments. Such a front-end can optionally be used to specify domain-specific modeling types and architectural models. When a modeling front-end is used, XDEVS type extensions and simulation models can be automatically generated; however, engineers may forgo the use of a front end and specify type extensions and simulation models directly in a programming language (C++ in the case of XDEVS). We have designed and implemented such a modeling front-end, which is described in our previous work [4]. Even though it is depicted in Figure 1 for completeness, the modeling front-end is not the focus of this paper. To achieve its objectives, the XDEVS framework supports both built-in architectural types and domain-specific extension types (described in Section 4.1) and a set of core modules that can be customized and optimized by engineers (described in Section 4.2).

4.1. Programming Model

The XDEVS simulation framework supports the design abstractions commonly used in architectural modeling languages. We refer to these abstractions as *metatypes* because they are instantiated by an engineer to define the system types

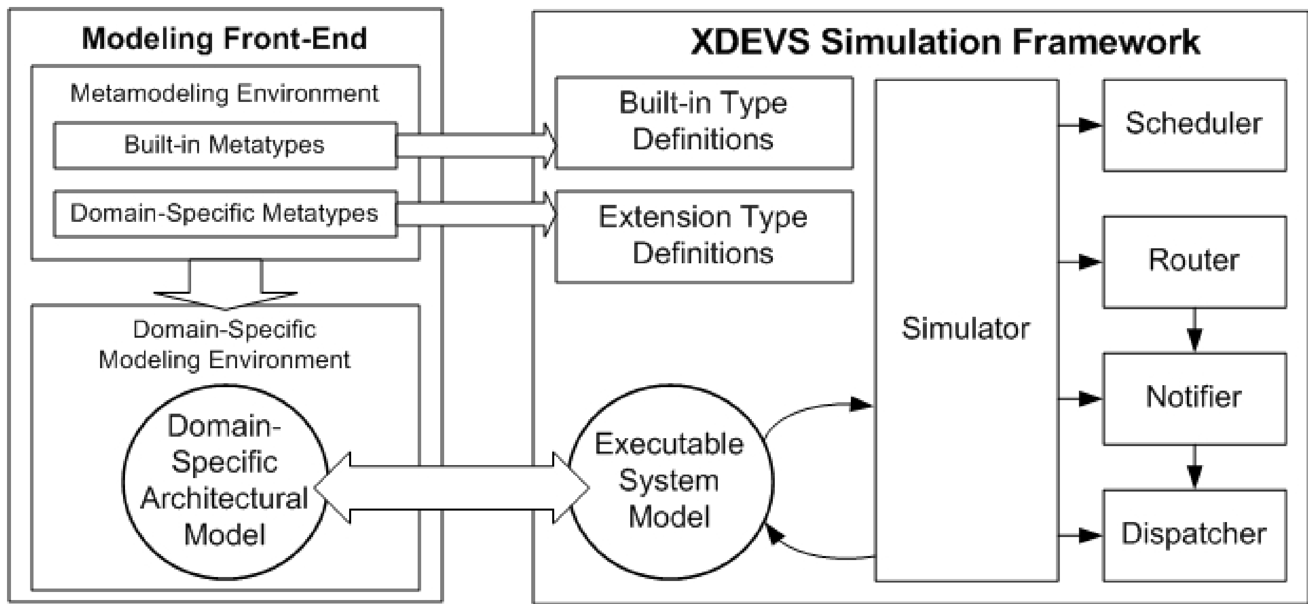

Fig. 1. High-level depiction of the XDEVS simulation framework.

that are used in a given system model. These system types are then instantiated within a simulation. For example, an engineer might define a `web server` component type that is an instance of the `component` metatype, and then create multiple instances of the `web server` in the simulation.

The three-tiered metamodeling approach (metatypes – system types – instances) (1) ensures separation of concerns by explicitly defining the capabilities and properties of each type, (2) ensures availability of appropriate abstractions by maximizing the reuse of common type definitions and providing the basis for the XDEVS language extensions mechanisms, and (3) ensures enforcement of relationships by explicitly constraining how types may interact. Note that such a three-tiered typing system has been employed previously [5], although we are the first to apply this approach to discrete event simulation.

We do not make a formal argument of the expressiveness or completeness of the XDEVS metatypes in this paper. However, in regard to *expressiveness*, the XDEVS metatypes are constructed to capture the consensus of the software architecture community as to the canonical set of architectural abstractions. Section 4.1.1 describes the built-in XDEVS metatypes and their implementation within the simulation framework. In regard to *completeness*, we explicitly designed XDEVS to support language extensions that define new domain-specific metatypes. Thus, the utility of the XDEVS framework does not rely on the completeness of the metatypes, and we do not aspire to make the set of built-in metatypes complete.

4.1.1. Built-in Metatypes. Figure 2 summarizes the XDEVS metatypes and their semantics, while Figure 3 illustrates the relationships among XDEVS metatypes. Due to space constraints, we describe only a subset of these types in detail here.

Architecture. The `architecture` metatype defines an execution environment for component and connector instances. As such, an architecture defines the logical connections among component and connector interfaces, and provides services and resources required by components and connectors. The instantiation and connection of components and connectors can be parameterized and modified dynamically. Architecture types (that is, instances of the architecture metatype) are implemented in XDEVS as subtypes of the `Architecture` class. Every architecture must define methods for instantiating components and connectors, connecting the interfaces of components and connectors, and retrieving references to all the resources it provides, such as threads and files.

In contrast to the above modeling approach, in existing simulation environments, an architecture may be modeled as a compound node. Using this approach, an architecture essentially becomes a “cardboard box” for a set of component and connector instances, mapping the interfaces of internal elements to external links. However, this only fulfills part of the role required of an architecture: it does not define a complete execution environment that includes platform services and resources, such as access to persistent storage and network interfaces. To provide this capability, a more complex and non-obvious mapping to a set of compound as well as simple nodes must be implemented.

By supporting the architecture metatype, XDEVS avoids a convoluted mapping between types. This is an example of how providing appropriate abstractions (recall Objective 2) simplifies simulation design and development. Moreover, our approach flexibly allows a single resource to be provided by multiple architectures, equivalent resources to be easily substituted, and platform services to be readily differentiated from application services. None of these are possible with the traditional approach mentioned above.

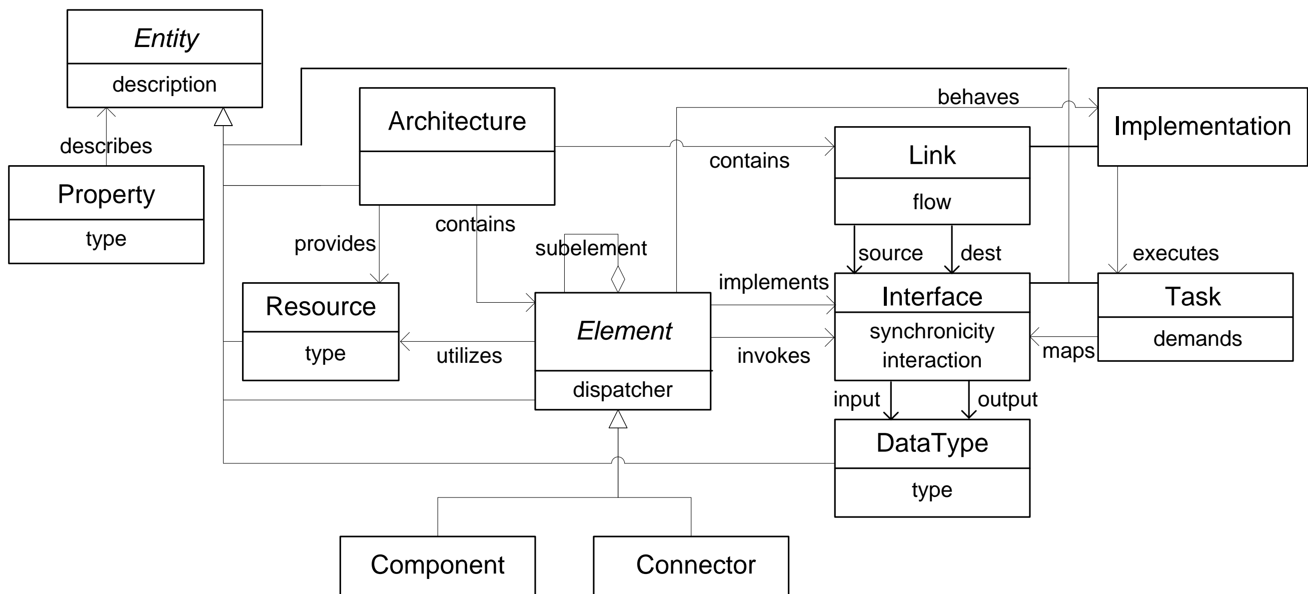

Fig. 3. Summary of the relationships between built-in XDEVS metatypes.

Component. The `component` metatype defines an independently deployable unit of software. To be independently deployable, a component defines a set of provided interfaces, a mapping from each provided interface to either a C++ class that implements the interface or the type-equivalent interface of a sub-component, and all external dependencies, such as required interfaces and resources, that the component expects its execution environment (represented by an architecture) to provide. Component types are implemented in XDEVS as subtypes of the `Component` class. Every component type must implement methods corresponding to each provided interface that delegate handling of interactions (initiated by an external entity) on that interface to either an another internal component object or an implementation object.

Using the basic types used in existing simulators, each component may be modeled as either a simple or compound node. This modeling approach has a major shortcoming however: it couples the *logic that describes how a task is performed* with the *entities that actually perform the task*, which is an example of failure to ensure separation of concerns (recall Objective 1. This distinction is critical in software architectures: software components describe reusable units of functionality, but that functionality is actually executed by shared resources (processors, threads, etc.). Consequently, there is no straightforward mapping from software components to discrete event nodes. Usually, rather than mapping software components to simple and compound nodes, software components are mapped to requests for shared resources (such as processor time), and the shared resources, not the software components, are modeled as nodes in the simulation. This solution remains unsatisfactory however, because it is an awkward way to describe a software architecture.

XDEVS properly separates logic from execution by supporting distinct component and resource metatypes. This separation both improves the reusability of component specifications in different contexts, permitting libraries of components and services to be easily defined and instantiated, and allows resources for which contention exists to be easily modeled and analyzed. Resource contention is a critical aspect of discrete event simulations because it is one of the fundamental determinants of system performance.

Interface. The `interface` metatype defines an interaction point between components and connectors. An interface may specify a data exchange in terms of typed inputs and outputs (e.g., parameters and return values) and/or a transfer of control (e.g., a method call in which a thread is passed from caller to callee). The relationship between an interface and a component may be either `implements` or `invokes`; interaction across an interface is always initiated by the component or connector that invokes the interface. The XDEVS `Interface` class serves a base type for classes that implement instances of the interface metatype.

The traditional discrete event simulation approach couples the interface of a node with its implementation. With existing simulators, a compound node can model an interface as a set of ports, and map those ports to the ports of sub-nodes that model the implementation of that interface. However, this prevents a node from implementing multiple interfaces because each node can only be contained within a single compound node. Thus, it is not possible to model a software component that implements multiple interfaces in this way. This is an example of both failure to separate concerns (Objective 1) and provide appropriate abstractions (Objective 2).

Implementation. The `implementation` metatype defines a set of concrete behaviors corresponding to the services it implements. Behaviors are described as sequences of `tasks` with arbitrarily complex, programmer-defined control flow. Each task requires a set of resources, such as CPU time or

Architecture: a collection of components, connectors, resources, and other types and the relationships among them.
Component: represents a locus of computation. Components interact with other components and connectors through interfaces.
Connector: implements communication and coordination facilities. Like components, connectors interact with external entities via interfaces.
Interface: an interaction point between a component or connector and external entities. Interfaces define entry points to component services in terms of data exchange and transfer of control.
Implementation: a set of concrete behavioral specifications that implement one or more component and connector interfaces.
Link: a logical connection between components or connectors over which information and/or control is exchanged.
Resource: an entity provided by the application execution environment which is utilized by components and connectors to perform tasks. Resources may be processing resources (e.g., threads), data resources (e.g., files), or communication resources (e.g., sockets).
DataType: a definition of data exchanged between components and connectors or maintained as part of a component's state.
Task: an externally accessible unit of functionality within a component or connector.

Fig. 2. Summary of the semantics of built-in XDEVS metatypes.

file access, to execute. When a task is initiated, requests are sent to all required resources. Each resource schedules the task's request according to its own scheduling policy. Once all resource requests have been satisfied, the task may perform computations, update the implementation's state, invoke required interfaces, or initiate other tasks within the implementation. Note that, by default, a task does not have to hold all required resources simultaneously; one resource request can be satisfied while other resource requests are blocking. Simultaneous resource access can, however, be implemented with appropriate modifications to the XDEVS framework.

To create an implementation in XDEVS, an engineer creates a class that inherits from the `Implementation` class. Every implementation class must implement several methods corresponding to each task. First, each task requires a method that implements the computation logic of the task – state updates, method invocations, control flow, and so on. Second, each task requires a method that creates a `Task` object that encodes any parameters passed to the task along with a reference to the implementation object that generated the task. This task object is sent to all required resources. This design is an application of the Command pattern, and allows resource requests to be queued, logged, and processed, without tangling the logic that performs a task with the resources that execute the logic. Finally, a method must be provided that computes the task's demand on each required resource. For example, an `encrypt` task might implement a method that computes the amount of processing time required to encrypt a data set as a function of the size of the data. This processing demand would be requested from and serviced by a processing-providing resource, such as a thread or CPU. Once a required resource is available, the task object is used to call the functions that compute the task's demand on the resource and perform the task's computational logic.

Resource. The `resource` metatype defines a resource provided by the computing environment which is used by implementations to perform tasks. Resources may be processing resources (e.g., CPUs or threads), data resources (e.g., files or buffers), or communication resources (e.g., network interfaces). Each resource may permit an arbitrary level of parallelism, i.e., a resource may be capable of servicing multiple requests simultaneously. For example, a `threadPool` resource might allow all the threads in the pool to be simultaneously assigned to different tasks. Requests may be scheduled according to any policy implemented by the resource (XDEVS provides a default FIFO scheduler).

4.2. Core Framework Components

The XDEVS simulation framework is organized as a set of components that interact to execute simulation models. These components implement basic functions required for simulation execution, including scheduling, routing, dispatching and exception handling. Figure **??** shows the XDEVS core framework components and their interactions.

Efficiency is critical to discrete event simulations because they are often used to analyze complex, large-scale systems with thousands of interacting objects. During the execution of a simulation, the simulation engine must determine, from a potentially extremely large set, the next events to execute, the recipients of output messages, and the next state of each node. Without constraints on the behavior of nodes, the simulation engine must assume worst-case conditions, e.g., that any node can produce output or change state arbitrarily. On the other hand, applying domain knowledge can expose opportunities for considerable improvement in algorithm efficiency, resulting in better scalability. Unfortunately, modifying the core scheduling, routing, and event handling algorithms of current discrete event simulation platforms is difficult because they were not designed with such customizability in mind: proprietary commercial simulation tools are closed source, while open source tools have poorly documented, monolithic implementations of the simulation kernel that make substitution of the algorithms with optimized, domain-specific versions difficult. Our previous experience implementing analysis capabilities on top of an open-source simulation platform

motivated the inclusion of extensibility mechanisms for the core framework functions.

The core algorithms of XDEVS that provide scheduling, routing, and other functions are encapsulated within independently substitutable modules. Each module implements an abstract interface that defines the operations the module provides.

Scheduler. The Scheduler is responsible for producing a global ordering of events, including state transitions and message exchanges. (The Scheduler does not schedule service requests or resource accesses locally at each node — this functionality is provided by the Dispatcher and Event Handler, as described below.) Events that are planned for future time steps are passed to the Scheduler along with an occurrence time and a reference to the Event Handler that will eventually interpret the event. At every simulation step, the Scheduler must be able to rapidly determine the next event(s) that will occur and invoke the appropriate Event Handlers for evaluation.

The Scheduler must be able to efficiently update its internal representation of scheduled events by adding new events as they are created and removing events that are no longer planned for execution. The optimal data structures and algorithms for searching and updating the set of scheduled events depends heavily on the characteristics of the system being simulated. For example, in systems where entities are continuously created and destroyed, it is beneficial for the Scheduler to support efficient removal of scheduled events that refer to destroyed entities. Similarly, in some systems, it may be possible for already-scheduled events to change their occurrence time without warning, and the Scheduler must take this possibility into account. In other systems, where such changes are not allowed, using a Scheduler that plans for and accommodates this possibility will be inefficient.

The Scheduler, as the global arbiter of event ordering, is also responsible for "breaking ties", i.e., choosing an ordering among events that occur at the same discrete time step. For example, when multiple entities are scheduled to transition state simultaneously, the Scheduler may enforce a bottom-up evaluation of state transitions (i.e., evaluating primitive entities at the leaves of the model first), a top-down ordering, or some other discipline. Again, the most efficient and appropriate policy varies from one domain to another.

Router. The Router is responsible for distributing messages that are output by simulation entities. For each output, the Router determines the set of recipients and passes the message to the appropriate Dispatcher (described next). In XDEVS, it is possible for a single simulation model to contain multiple Routers that distribute different types of events, implement different routing protocols, or service different simulation entities. The determination of which Router handles each event is always made dynamically by the sending node.

In most simulations, the Router's primary activity is efficiently finding paths between model entities, given a directed graph of logical connections. However, in some cases, quite different types of algorithms and data structures are required. For example, a Router that performs publish-subscribe event distribution must also implement routines for retrieving the list of subscribers for each published message. In Section **??**, we show how we modified the default Router implementation to support the pub-sub paradigm.

Dispatcher. The Dispatcher demultiplexes and queues incoming events that are delivered to a node by a Router. For each incoming event, the dispatcher determines when and how it will be scheduled for execution by an Event Handler. Like the Router, Dispatchers may have a n-to-m relationship with model entities; a Dispatcher may perform event demultiplexing and queuing for multiple entities, and a single node may utilize multiple Dispatchers.

In conventional simulations, the Dispatcher ensures that all resources required to execute the event are available before the event is passed to an Event Handler. For example, if a service request arrives at a component, the Dispatcher finds an available (simulated) thread within the component's (simulated) execution environment to perform the service. As such, the Dispatcher arbitrates access to shared resources by performing local scheduling of events at each node (as opposed to the global scheduling function performed by the Scheduler).

The Dispatcher may use a number of different schemes for ordering the processing of incoming messages and allocating resources under contention. For example, the Dispatcher may use a priority scheme for queuing messages or it may treat all requests fairly. It may implement message queues that grow dynamically or drop messages when overflow occurs. A number of possible strategies exist for assigning threads from a pool to handle queued requests. In Section **??**, we describe how the Dispatcher can be customized to perform leader-follower event demultiplexing.

References

[1] J. Banks, J. Carson, B. Nelson, and D. Nicol, *Discrete-event system simulation*. Prentice Hall Upper Saddle River, NJ, 2001.

[2] B. W. Boehm, R. K. McClean, and D. B. Urfrig, "Some experience with automated aids to the design of large-scale reliable software," in *Proceedings of the international conference on Reliable software*. New York, NY, USA: ACM, 1975, pp. 105–113.

[3] J. Dabney and T. Harman, *Mastering Simulink*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[4] G. Edwards and N. Medvidovic, "A methodology and framework for creating domain-specific development infrastructures," Sept. 2008, pp. 168–177.

[5] G. Jung and J. Hatcliff, "A type-centric framework for specifying heterogeneous, large-scale, component-oriented, architectures," in *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*. New York, NY, USA: ACM, 2007, pp. 33–42.

[6] J. Nutaro, "ADEVS (A Discrete EVent System simulator)," *Arizona Center for Integrative Modeling & Simulation (ACIMS), University of Arizona, Tucson, AZ, http://www. ece. arizona. edu/nutaro/index. php*.

[7] D. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[8] B. Sharda and S. J. Bury, "A discrete event simulation model for reliability modeling of a chemical plant," in *WSC '08: Proceedings of the 40th Conference on Winter Simulation*. Winter Simulation Conference, 2008, pp. 1736–1740.

[9] A. Varga *et al.*, "The OMNeT++ discrete event simulation system," in *Proceedings of the European Simulation Multiconference (ESM2001)*, 2001, pp. 319–324.