Simulation Modeling Architecture (SiMA), A DEVS Based Modeling and Simulation Framework

Ahmet Kara, Fatih Deniz, Doruk Bozağac, M. Nedim Alpdemir
TUBITAK UEKAE İLTAREN
06800 Ümitköy
Ankara, TURKEY
{ahmetk, fatihd, dorukb, nedima}@iltaren.tubitak.gov.tr

Keywords: Modeling and simulation, DEVS, strongly-typed environments, re-usability, model composition

Abstract
Defining and executing simulation models of complex systems is often a non-trivial task. One of the key requirements of simulation software frameworks for such complex systems is that the model construction should be supported by hierarchical and modular composition of reusable models in a uniform and scalable way. Further, the accompanying simulation execution protocol should provide a robust and reliable mechanism for executing the models according to a well-defined modeling formalism. In this paper we introduce our simulation software framework, namely Simulation Modeling Architecture (SiMA) and our contributions to the underlying formal model (i.e. DEVS). We also talk about our simulation construction environment and provide performance comparisons of SiMA with other popular DEVS-based simulation frameworks with respect to message processing efficiency and the ability to cope with coupling complexity caused by increased model population, cardinality and the level of hierarchy.

1. INTRODUCTION
Component-based simulation frameworks have long been investigated in the simulation research community. Modelica [1] language, DSBlock[2] and Simulink[3] systems are examples of component-based simulation systems with emphasis on Continuous Simulation. DEVS [4][5] approach COST [6], SIMKIT [7] systems on the other hand are examples that are more discrete event simulation centric. A common characteristic of these systems and approaches is that they try to establish mechanisms, rules, specifications that would enable the construction of complex simulation models from smaller sub-models in a systematic way. DEVS approach among these is of particular interest for a number of reasons: 1 - It provides a set theoretic formal basis for both structural and behavioral specification and hierarchical composition of simulation models with arbitrary complexity levels, 2 – It specifies a protocol for model execution, 3 – It promises a unifying theoretical frame for hybrid systems [8] with both continuous and discrete event properties (see [9] for an in-depth discussion on representing continuous systems with DEVS via quantization of state base). These properties have contributed to our decision to use DEVS as the formal basis for our framework. To elaborate more on this, SiMA was developed to meet strict requirements for a high degree of model composability and re-usability, high performance and robustness, and direct support for industrial quality simulation application development. It is a known fact that architectural flexibility and run-time efficiency are often conflicting goals for software frameworks. To overcome the challenge, we took a systematic approach to reconcile the tension between these conflicting goals:

- We benefited from the sound formal specification offered by DEVS with some constraints to establish strict semantics for model definition, composition and execution,
- We designed our framework as a layered architecture where CPU intensive simulators can be written in an efficient programming language (i.e. C++) whereas the framework layer is written in .NET platform to support easier binary component composition and distribution.

As a result, SiMA comes to possess the following distinct characteristics:
- It is based on DEVS approach which provides a solid formal basis for complex model construction;
- Its Simulation Execution Engine implements the parallel DEVS protocol which provides a well-defined and robust mechanism for model execution;
- It extends the DEVS formalism with a specific port type and transition function;
- It comes with a variety of tools such as a Graphical Model Editor, an automated model construction pipeline which aims at providing support for Model Driven Engineering, an efficient and configurable trace generation framework, a Code Generation Tool (KODO) that supports message processing infrastructure through automated marshalling/unmarshalling across the simulation layers.
The rest of the paper is structured as follows; Section 2 provides a short background on DEVS formalism to lay the ground for the discussion of our extensions to DEVS, Section 3 provides a discussion of our extensions to DEVS, Section 4 introduces SiMA software architecture, Section 5 provides the results of performance tests conducted to compare SiMA to a selected number of other DEVS based simulation framework in the context of a case study, and finally Section 6 provides some concluding remarks.

2. BACKGROUND

The Discrete Event System Specification (DEVS) is a formalism introduced by Zeigler (1976) to describe discrete event systems. In DEVS formalism, a basic model (atomic model), is defined formally as \[ M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta > \] where \( X \) is the set of input events; \( S \) is the set of states; \( Y \) is the set of output events; \( D \) is the set of component influencers of \( d \), \( \forall d \in D \); \( I_d \) is the set of component influencers of \( d \), \( \forall d \in D \cup \{N\} \); \( Z_{d,i} \) is the i-to-j output to input function, \( \forall j \in I_1 \); \( SELECT \) is the tie-breaking function to arbitrate the occurrence of simultaneous events. Complete description of DEVS semantics can be found in [4][5].

3. SIMA-DEVS: OUR APPROACH TO DEVS

Our modeling and simulation framework SiMA (Simulation Modeling Architecture) is based on the DEVS approach as a solid formal basis for complex model construction. SiMA [10] Simulation Execution Engine implements the parallel DEVS protocol which provides a well-defined and robust mechanism for model execution. SiMA builds upon a specialized and extended form of DEVS formalism which:

1. Formalizes the notion of “port types” leading to a strongly-typed (and therefore type-safe) model composition environment. In this respect we specialize the basic DEVS formalism by introducing semantic constraints on the port definitions;
2. Introduces a new transition function to account for model interactions involving state inquiries with possible algebraic transformations (but no state change), without simulation time advance. In this respect we extend the basic DEVS formalism. This is similar to the notion of zero-lookahead in HLA [11] from a time-management point of view.

Our SiMA-DEVS formalism is given below:

\[ SiMA-DEVS = <+ X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta, \delta_{gl} > \]

2. BACKGROUND
3. SIMA-DEVS: OUR APPROACH TO DEVS

Our modeling and simulation framework SiMA (Simulation Modeling Architecture) is based on the DEVS approach as a solid formal basis for complex model construction. SiMA [10] Simulation Execution Engine implements the parallel DEVS protocol which provides a well-defined and robust mechanism for model execution. SiMA builds upon a specialized and extended form of DEVS formalism which:

1. Formalizes the notion of “port types” leading to a strongly-typed (and therefore type-safe) model composition environment. In this respect we specialize the basic DEVS formalism by introducing semantic constraints on the port definitions;
2. Introduces a new transition function to account for model interactions involving state inquiries with possible algebraic transformations (but no state change), without simulation time advance. In this respect we extend the basic
Note also that, in addition, we introduce a new transition function, \( \delta \), that enables models to access the state of other models through a specific type of port, without advancing the simulation time. As such, it is possible to establish a path of connected models along which models can share parts of their state, use state variables to compute derived values instantly within the same simulation time step. As stated earlier, this is similar to the notion of zero-lookahead found in HLA [11]. One may argue that the zero-lookahead behavior could be modeled by adjusting the time advance function of an atomic model such that the model causes the simulation to stop for a while, do any state inquiry via existing couplings, then re-adjusting the time advance to go back to normal simulation cycle. Although this is possible, we argue that by introducing a transition function and a specific port type which is tied (through run time constraints imposed by the framework) to that particular transition function we gain several advantages:

1. The models can communicate and share state with each other without the intervention of the simulation engine thus providing a very efficient run time infrastructure.
2. Allowing such communications only to occur through a specific port type (compile time and run-time checks are carried out) the framework is able to apply application independent loop-breaking logic at the ports to prevent algebraic loops, thereby ensuring model legitimacy.

4. SIMA SOFTWARE ARCHITECTURE

SiMA is a software framework for developing simulation models and executing simulations (Figure 1). It has two main layers: SiMA Core and C++ Interface.

SiMA Core consists of five sub-components that are used for modeling and simulation. Modeling Framework is a set of classes and data types to use in model development. Atomic models and all their subclasses are defined in this framework. Connection Ports is the transportation component that contains classes for defining ports and their connection limits. Simulation Engine Core uses these two components, Modeling Framework and Connection Ports, to execute a simulation. In other words, it implements the DEVS simulation protocol and in addition exposes administrative interfaces to manage and track the simulations at run time. The Distributed Simulation Adapter is the interface for connecting SiMA simulations with external distributed simulation infrastructures. Messaging Constructs is the component that presents all base data type classes and rules for inter-communication of atomic models and SiMA Core components.

The C++ Interface layer is implemented to allow C++ to be used as a model implementation language for SiMA atomic models. All core SiMA components are developed in .NET but we support models implemented in both .NET and C++ to co-exist in the same run-time environment during a simulation. However, since there is a strict boundary between their coding environments, various adapters and components that manage the interoperability between .NET and C++ atomic models and SiMA components are implemented in the C++ Interface layer.

The C++ Interface layer consists of three sub-components. Unmanaged Modeling Adapter has the same interface and class hierarchy as the .NET Modeling Framework except it is developed in pure C++ language. Managed Modeling Adapter is developed in C++/CLI, which is a special edition of C++ language in .NET, that allows access to both C++ and .NET methods and data types. Managed Modeling Adapter handles the interoperability management and delegates all simulation commands to C++ models, and provides all information required by them from the simulation environment. Data Converters are special adapters that perform marshalling all values between .NET and C++ data types in both ways. A model developer can use the KODO tool (described in Section 4.1) to auto-generate these data converters for his/her data types.

4.1. Model-Driven simulation construction pipeline for SiMA

SiMA is supported with a number of tools that collectively allows automatic simulation model construction from previously defined SiMA models. The simulation construction pipeline consists of four steps with the support of five tools (Figure 2).

**KODO**: A model developer using SiMA needs to define data types to be used for model initialization and inter-port communication. Due to .NET/C++ interoperability requirements SiMA needs to apply special handling on port data types for serialization/deserialization of data values flowing between atomic models. Similarly, model initialization phase requires access to input
configuration data types. KODO plays its relieving role for the developer by auto generating not only data beans but also additional processing logic for marshaling and type conversion, for access to model input configuration files for state initialization, and for trace generation. KODO allows definition of data types with simple XML definitions and generates all classes and methods required for the model developer.

**Figure 2: Simulation Construction Pipeline**

**Scenario Analyzer:** There are two starting points in our simulation construction pipeline. One of those has a more constrained initial specification where a simulation study is defined by a *scenario* that describes all required models, model coupling information and model initialization data. In this case our Scenario Analyzer interprets a scenario definition and identifies all atomic models (from the component library) and defines their composition hierarchy in an intermediate form to be used by the model linker. It is generally an application dependent analyzer that converts the application dependent scenario file to be used by application independent model linker.

**Model Editor:** The second route to the pipeline is the SiMA Model Editor. The Editor provides a context independent model composition environment where both atomic and coupled models can be organized as libraries, thereby facilitating reusability of simulation models. SiMA allows defining atomic models either in C++ or in C# programming languages. Once these models are compiled into dlls, they can then be graphically coupled using SiMA Model Editor in hierarchical block diagrams to construct more complex systems. In Figure 3, the model structure and couplings for a sample simulation can be seen. Dark rectangular shapes with an ‘A’ sign on the top-left corners indicate atomic models and lighter rectangular shapes with no signs on the top-left corner indicate coupled models. Couplings are shown with directed arrows between model ports. The direction of the arrow connotes the direction of the data flow. Port names are denoted as labels beneath the ports. The Model Editor produces an intermediate form to be used by the Model Linker.

**Model Linker:** Model linker reads its special scenario definition file and loads the data type mapping rules for the current configuration and then produces two output files. The first file, mapping definition, includes all port connections, names and locations of atomic models and their hierarchy mapping to produce a simulator for the scenario. Second file, SiMA configuration, includes all initialization data for each atomic model in the scenario.

**Model Builder:** Model builder reads the mapping definition output of the model linker, then creates and combines all atomic model object instances in the memory of the host computer. As a result a single coupled model object instance is returned to the caller, which can be used by the SiMA simulator to run a simulation.

**Figure 3: Example Model Structure**

5. **CASE STUDY**

In this section, performance measurements and analysis of SiMA environment will be provided. SiMA will be compared to some existing DEVS based simulation frameworks, which are DEVSJava[12][13], PowerDEVS[9] and ADEVS[14].

For this reason, a sample Wireless Sensor Network (WSN) simulation is used. Required models are implemented using each of these frameworks and their performances are measured. The example WSN system consists of four components:

1. **Sensors:** Sensors sense the movement activities in the environment and can communicate with other sensors within their range. Each sensor consists of four subcomponents: Antenna, Sensing Unit, Processor and Battery. Each sensor is represented by a coupled model and its subcomponents are atomic models.

2. **Main Sensor:** Main sensor can communicate with the sensors within its range. It sends queries to the sensors and waits for the response messages. Main sensor contains two subcomponents: Antenna and Processor. Unlike other sensors, main sensor does not sense the environment and it does not contain a battery.
3. **Truck**: Truck has a predefined circular path and it follows this path during the simulation. Sensors that get close enough to the truck can understand the location of the truck and prepare a message to be delivered to the main sensor.

4. **Logger**: Saves all the location and data packages, created by sensors and truck, in a structured way.

A top level visual representation of DEVS models defined for this simulation is given in Figure 3.

In this case study, tests are performed using a trivial scenario where:
- Main sensor creates and broadcasts a query message.
- When a sensor receives this message, it also broadcasts it to be able to distribute the query to the maximum number of nodes.
- Sensors that receive the query message also start collecting movement activities from the environment via their sensing units.
- When a sensing unit receives movement activities from the truck, it sends a message to its processor.
- Processors create data messages according to the received query and collected data from the environment.
- When messages are ready, processor sends these new messages to its antenna to be sent to the main sensor.
- Main sensor collects these messages and understands the current location of the truck.

5.1. **Testing Environment and Test Criteria**
Tests are performed in Windows XP environment on a machine with the following features: Intel(R) Pentium(R) 4 CPU 2.40GHz Processor, 1 GB of RAM. The tests are organized according to two criteria:
- **The impact of message size on performance**: The target here is to observe the behavior of the framework in terms of performance when the message size increases. Number of messages circulating between sensors is proportional to the square of number of sensors. So, changing sensor count will considerably affect the number of messages and hence the amount of data exchanged during one simulation step. Therefore this criterion is tested by increasing the number of sensors. Note that in this test we do not increase the number of atomic models in the simulation, but rather we use a Manager Atomic Model that creates and manages sensor instances as required.
- **The impact of model complexity on performance**: The target in this case is to observe the impact of increasing the model number causing an increase both in coupling complexity, and in hierarchical sophistication level. As stated earlier more than one sensor can be modeled by a manager sensor model, however in this particular test, number of sensors in the simulation will be fixed, but the number of manager sensor models will vary to change the model complexity. In other words, in each run a different number of atomic models will be present in the overall simulation.

5.2. **Test Results**
Simulations are executed for 10 simulation seconds with a step size of 0.03 seconds. The completion time of each simulation is measured in terms of wall clock time. In each simulation ((Sensor Count*10)*100)/3 packets arrives to the main sensor. The impact of message size on performances of the tools can be seen in Table 1 and Figure 4. As it can be observed from the results, SiMA and PowerDEVS perform much better compared to DEVSJava and ADEVS environments. It can also be observed that, the gap between SiMA and PowerDEVS closes when sensor count increases and for 100 sensors SiMA tends to perform better than PowerDEVS.

<table>
<thead>
<tr>
<th>Sensor Count</th>
<th>SiMA Completion Time (s)</th>
<th>PowerDEVS Completion Time (s)</th>
<th>DEVSJava Completion Time (s)</th>
<th>ADEVS Completion Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.56</td>
<td>0.25</td>
<td>13.80</td>
<td>3.40</td>
</tr>
<tr>
<td>10</td>
<td>2.15</td>
<td>1.01</td>
<td>144.6</td>
<td>16.8</td>
</tr>
<tr>
<td>15</td>
<td>4.73</td>
<td>2.61</td>
<td>698.0</td>
<td>40.8</td>
</tr>
<tr>
<td>20</td>
<td>8.56</td>
<td>5.30</td>
<td>2210</td>
<td>83.2</td>
</tr>
<tr>
<td>25</td>
<td>13.81</td>
<td>8.61</td>
<td>6000</td>
<td>148.9</td>
</tr>
<tr>
<td>30</td>
<td>19.94</td>
<td>15.54</td>
<td>10204</td>
<td>236.5</td>
</tr>
<tr>
<td>50</td>
<td>56.26</td>
<td>50.59</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>100</td>
<td>268.4</td>
<td>348.1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 1: Performance measurements on message size

![Figure 4: Performance measurements on message size](image-url)
To observe the impact of model complexity on the performance, we used a simulation with 30 sensors. However, contrary to the previous test these 30 sensors are now modeled by an increasing number of manager models causing the number of atomic models in the simulation to increase gradually which causes the coupling complexity to increase, too. The measurement results can be observed in Table 2 and Figure 5. For each simulation, there were totally ((4*Manager Model Count) + 4) atomic models and we observed that adding complexity to the models slows down the simulations with comparable proportions for SiMA, PowerDEVS and ADEVS. However, note that as the model number increases, performance degradation tends to be nonlinear. PowerDEVS and SiMA perform relatively better compared to ADEVS, with PowerDEVS performing slightly better than SiMA. For DEVSJava the results seems to improve until the model number is 6 and tends to increase again from then on. We believe this is because the package dispatching and handling mechanism in DEVSJava suffers from a bottleneck when too many packages are produced by a single model. This behaviour is persistent up to a certain point where sensor data generation is a balanced between separate manager models.

<table>
<thead>
<tr>
<th>Manager Model Count</th>
<th>SiMA Completion Time (s)</th>
<th>PowerDEVS Completion Time (s)</th>
<th>DEVSJava Completion Time (s)</th>
<th>ADEVS Completion Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.94</td>
<td>15.55</td>
<td>10204</td>
<td>236.5</td>
</tr>
<tr>
<td>2</td>
<td>20.65</td>
<td>15.78</td>
<td>5169</td>
<td>244.2</td>
</tr>
<tr>
<td>3</td>
<td>21.53</td>
<td>15.86</td>
<td>4714</td>
<td>253.1</td>
</tr>
<tr>
<td>5</td>
<td>23.91</td>
<td>16.81</td>
<td>3741</td>
<td>264.1</td>
</tr>
<tr>
<td>6</td>
<td>24.99</td>
<td>17.63</td>
<td>3454</td>
<td>268.5</td>
</tr>
<tr>
<td>10</td>
<td>30.96</td>
<td>19.65</td>
<td>4627</td>
<td>285.3</td>
</tr>
<tr>
<td>30</td>
<td>54.01</td>
<td>34.22</td>
<td>9088</td>
<td>385.2</td>
</tr>
</tbody>
</table>

Table 2: Performance measurements on model complexity

6. CONCLUSIONS AND FURTHER WORK

In this paper we have described our DEVS-based Modeling and Simulation Framework, SiMA, by elaborating:

- First on our extensions on formal semantics of DEVS. In this respect we make strong typing and type-system dependency of the ports explicit in the formal model. Thereby introducing a type discipline to the definition of the externally visible model interfaces.

- Second on the architectural properties that makes SiMA a good performer and a flexible, easy to use software framework,

- Third on SiMA tool support to facilitate Model Driven Engineering for developers.

Further, we have presented the results of performance tests conducted to compare several popular DEVS-based simulation frameworks.

SiMA has been used in a number of simulation applications successfully with strict requirements targeting high performance and ease of model development for simulation of complex systems. We plan to improve our simulation construction pipeline in particular by focusing on SiMA Model Editor to make it a full-fledged layout editing and code generation environment. There is also ongoing work within our team to add dynamism support to SiMA in conformance to both dynDEVS and dynNDEVS [15] as the underlying formal specification, with some non-disruptive extensions to the original formal semantics.

References
Author Biographies

Ahmet KARA is a Senior Researcher at TUBITAK UEKAE ILTAREN. He has been involved in design and implementation of modeling and simulation architectures. He received his BSc (2003) and MSc (2006) degrees from Bilkent University, Ankara, Turkey. He is currently a PhD student in Department of Computer Engineering of Middle East Technical University.

Fatih DENİZ is a Researcher at TUBITAK UEKAE ILTAREN. He received his BSc degree from Bilkent University, Ankara, Turkey in 2007. He is currently a MSc student in Department of Computer Engineering of Middle East Technical University. His current research interests include model-driven engineering and variable structure models.

Doruk BOZAĞAÇ is a Senior Researcher at TUBITAK UEKAE ILTAREN. He has been involved in design and implementation of modeling and simulation architectures. He received his BSc (2003) and MSc (2006) degrees from Bilkent University, Ankara, Turkey. He is currently a PhD student in Department of Computer Engineering of Middle East Technical University.

M. Nedim ALPDEMİR received his MSc (1996) in Advanced Computer Science and PhD (2000) in Component-Based Simulation Environments from the Department Computer Science, University of Manchester, UK. He worked as a Research Associate, and later as a Research Fellow in the Information Management Group (IMG) at the Department Computer Science of University of Manchester, UK, until 2005. Currently he is the head of the Software Infrastructures Group and supervises the Simulation Software Frameworks team at TUBITAK UEKAE ILTAREN, Ankara, Turkey.