AutoDEVS: A Methodology for Automating Modeling and Simulation Software Development and Testing of Interoperable Systems
Manuel C. Salas and Bernard P. Zeigler

The online version of this article can be found at:
http://dms.sagepub.com/cgi/content/abstract/6/1/33
AutoDEVS: A Methodology for Automating Modeling and Simulation Software Development and Testing of Interoperable Systems

Manuel C. Salas, Bernard P. Zeigler

Abstract
Despite the increasing use of software system development methodologies that follow a highly-structured and systematic process, there remain critical limitations that need more advanced treatment. One such limitation is the failure to explicitly incorporate modeling and simulation (M&S) methodology, which becomes essential as the scale of systems under development exceeds the range supported by analytical tools. Other limitations include the lack of automation to reduce routine work and incoherence among the abstractions developed at various stages in development. Originally introduced as formalism for discrete event M&S, the Discrete Event System Specification (DEVS) methodology has become an engine for advances within the wider area of information technology. In this paper, we present a new DEVS-based tool called the ‘AutoDEVS’ that automates systems development and exploits ‘model continuity’ to maintain coherence through the development process and to support test artifact continuity and traceability through phases of system development. The AutoDEVS creates models and systems that can be mapped to DEVS/service-oriented architecture (SOA), a DEVS implementation compliant with a SOA infrastructure, such as that adopted by the Department of Defense’s (DoD’s) Global Information Grid initiative. After describing the AutoDEVS environment, we show how it provides a rapid means of integrated development and testing of defense command and control systems for interoperability.

Keywords
DEVS, SES, modeling, simulation, methodology, automation, software development, software testing, interoperable systems, SOA

I. Introduction
The development of complex software-intensive systems requires systematic methodologies to ensure that all requirement goals and objectives are met. Many organizations make use of the System Development Life Cycle (SDLC), Joint Application Development (JAD), and Rapid Application Development (RAD) methodologies to assist in developing systems. However, these methodologies tend to mishandle evolving requirements during the development of the system, calling for the support of feedback and iteration that is integral to the methodology and tools. Of particular interest in this paper are two main limitations.

1. None of the current methodologies integrate Modeling and Simulation (M&S) techniques to develop systems. M&S helps to better understand and optimize performance and/or reliability of systems and verify the correctness of designs. Simulation allows for the development of ‘virtual environments’ to observe and better understand the system under development. These virtual environments also permit speeding up of the development process by validating the system as early on in the development cycle as possible, before the actual hardware is ready, thus reducing risks and costs encountered during the testing phase.

University of Arizona, Tucson, AZ, USA

Corresponding author:
Manuel C. Salas, Arizona Center for Integrative Modeling and Simulation, University of Arizona, Tucson, AZ, USA
manuel.salasc@gmail.com
2. These methodologies do not overcome the incoherence problem caused among the different phases of the development process. For instance, the lack of clarity in requirements could impact the elaboration of architectures and test plans for the system that could not be detected until later in the development process, causing major design changes and costing the project both time and money.

1.1 Motivation

Several goals motivated the development of the ‘AutoDEVS’, a tool that brings new methodologies based on Discrete Event System Specification (DEVS) modular and hierarchical formalism:

- improve systems development to reduce human effort, time constraints, and production costs between different design stages;
- overcome the ‘incoherence problem’ between different stages of the development process;
- help developers spend most of their time on the parts of the system that really matter and reduce as much overhead as possible on repetitive activities;
- introduce automation in the development of systems to increase productivity and produce high-quality, structured solutions to complex problems;
- organize a family of alternative models from which a candidate model can be selected, generated, and evaluated; and
- meet the needs of the entire team, from requirements capture to deploying high-performance, real-time distributed executing models.

Overall, the main contribution of this article is the demonstration of a practical methodology to automate the development and testing of complex, distributed, real-time systems. This methodology is based on DEVS and System Entity Structure (SES) formalisms and overcomes the ‘incoherence problem’ between different stages of the development process by connecting the outputs of one stage to the inputs of the next stage. It allows the users to go from requirements to automated models and the simulation of a real-time distributed software system, and to make use of these models to auto-generate test models and make the system ready to run and interact with the real-world. In particular, we discuss the application of the methodology, as supported by the AutoDEVS tool set, to complex distributed command and control (C2) systems for which interoperability is a key attribute.

The remaining discussion is organized as follows: Section 2 discusses the state of the art in software development. Section 3 provides a brief review of the most advanced tools that are based on DEVS formalisms and methodologies for the development of systems. Section 4 introduces the AutoDEVS as the next generation tool to automate the development and testing of systems, going from natural language to a real-time executing system, exploiting the ‘model continuity’ concept through the stages of a development process. Finally, Section 5 provides conclusions for the AutoDEVS and the importance of M&S in the design and evaluation of systems for data and computation interoperability.

2. Software Development State of the Art

Traditionally, many organizations make use of the Systems Development Life Cycle (SDLC) methodology to assist in developing systems, as it ensures that all functional and non-functional requirement goals and objectives are met. The SDLC provides a set of models or methodologies such as waterfall, v-shaped, incremental, spiral, build and fix, and synchronize and stabilize. The waterfall model defines a sequence of stages in which the output of each stage becomes the input for the next. This model allows phases to be processed and completed one at a time, but it is very rigid and changes to requirements can potentially have a negative impact on the system. The v-shaped model is similar to the waterfall model with the difference that testing procedures are developed before starting the implementation phase. The v-shaped model has a greater probability for success than the waterfall model, due to the development of test plans early on during the life cycle process. On the other hand, no early prototypes of the system are produced. The incremental model consists of dividing the waterfall model into smaller, more easily managed iterations. The incremental model allows one to have a base working version of the system, where flaws are detected quickly and early on in the process. On the other hand, system architecture problems may arise due to changes in requirements in later iterations. The spiral model is most often used in large, complex and expensive systems. This model is similar to the incremental model, with more emphases on risk analysis. The spiral model includes a high amount of risk analysis that allows evaluation of different alternatives to mitigate a specific problem and then choose the best fit for the system in development. On the other hand, the project success is highly dependent on the risk analysis phase. In the build and fix model a system is built with minimal requirements and specifications and no design or testing is executed. The build and fix model is usually effective in very small projects where the complexity is very limited. However, maintenance could become an issue since no documentation is produced and when this model is applied to highly-complex systems it can result in a low-quality, delayed and costly system. In the synchronize and stabilize model, teams work concurrently on individual
application modules, executing frequent code synchronization between the teams, and allowing stabilization of the system regularly throughout the development process. Since the synchronize and stabilize model allows changes at any point throughout the process, it is inherently more flexible, and allows responsiveness to changing business requirements.

There are alternatives to the SDLC methodologies, such as Joint Applications Design (JAD) and RAD. The JAD technique allows end users to participate in the requirements development process to better understand their needs and set up the desired system. When the size of the group and the system is too large, the JAD technique could become expensive and cumbersome; however, it reduces the ambiguity produced during the system requirements elaboration phase. The RAD model involves the iterative development and construction of prototypes that allow emulation and provide the ability to quickly build working systems to test their usefulness. This model provides the ability to rapidly change system design as demanded by users. Following the rapid prototyping model can lead to a succession of prototypes that never culminate in a satisfactory production application. More speed and lower costs may lead to lower overall quality. The potential for inconsistent designs within and across systems and the difficulty with model reuse for future systems are some of the most common weaknesses of the RAD model.

Rational Rose RealTime is a complete lifecycle Unified Modeling Language (UML) development environment intended for the modeling and component construction of enterprise-level software applications that are highly event driven, concurrent and distributed. Rose RealTime unifies the project team by providing an extensive set of tool integrations to meet the needs of the entire team, from requirements capture through to high-performance code generation and debugging for real-time operating system targets. However, Rose RealTime is still suffering from several problems. Firstly, the tool focuses on design and it is not fully suited for requirements modeling. Second, Rose RealTime does not support concurrency in Statechart Diagrams, which is a disadvantage during the requirements and analysis phases. Finally, the tool does not support activity diagrams at all, and important processes such as the verification of properties (safety, utility, liveness), the simulation of the system, and the generation of test cases.

Matlab is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. Using the Matlab tool, developers can solve technical computing problems faster than with traditional programming languages, such as C, C++, and FORTRAN. Matlab is currently being used by a wide range of applications, including signal and image processing, communications, control design, test and measurement, financial modeling and analysis, and computational biology. In addition, this tool provides a number of features for documenting and sharing developers’ work. Matlab can be integrated with other languages and applications. Simulink is integrated with Matlab and it is an environment for multidomain simulation and Model-Based Design (MBD) for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that allow one to design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing. The Matlab environment for MBD allows engineers to mathematically model the behavior of the physical system, design the software and model its behavior, and then simulate the entire system model to accurately predict and optimize performance. The system model becomes a specification from which one can automatically generate real-time software for testing, prototyping, and embedded implementation, thus avoiding manual effort and reducing the potential for errors. The major drawbacks of this environment are its size and relative complexity; it takes some time to become familiar with its language and with several of the main routines needed for basic simulations. Equations must be handled in certain form and sequence, requiring the user to understand the assumptions underlying the tool, in addition to the system’s phenomena being analyzed. Furthermore, a good comprehension of numerical analysis, linear algebra, and linear systems is required.


DEVS is a unified framework for developing real-time software systems in which logical analysis, performance evaluation, and implementation can be performed, all based on the DEVS formalism. The DEVS framework is based on systems engineering principles and is used for M&S in many application domains. This framework supports a number of important features, such as component-based hierarchical simulation model development, scalability, reusability, and distributed simulation. The framework was later extended in 2003, namely to Communicating DEVS for logical analysis and Real-Time DEVS (RTDEVS) for implementation, to support parallel model specification and execution, hierarchical model development and modularity, and object orientation. The DEVS framework has been implemented in different programming languages, such as DEVSJAVA (Java), ADEVS (C++) and DEVS# (C#) and over various middleware, such as DEVSCORBA, DEVSHigh level architecture (HLA) and DEVSOA.

RTDEVS formalism extends the classic DEVS formalism in atomic DEVS models. The RTDEVS formalism for
coupled models remains the same as the original. In the classic DEVS formalism, simulation time advances only when a simulator calls the time advance function \( \tau_a \) of the associated model. The time advance function \( \tau_a \) in the RTDEVS formalism behaves the same as that in the classic DEVS formalism, except that here the time is an integer, while in classic DEVS, time is a real number. The time calculated by the time advance function is also synchronized with the wall clock time. This is because a simulation clock in RTDEVS is no longer a virtual clock, but a real-time clock. An activity is an operation that takes a certain amount of time to complete the assigned task. This was adopted by Hong and Kim\(^{15} \) to represent some time-consuming operations, such as waiting for a message, processing a job, and so forth.

3.1 DEVS and DEVSJAVA

DEVSJAVA implements the DEVS formalism on java. It provides a set of libraries, on top of the java native libraries, that allow the building of modular and hierarchical model compositions based on the closure-under-coupling paradigm to create a system, see Figure 1.

A system is described as a set of input/output (I/O) events and internal states along with behavior functions regarding the event consumption/production and internal state transitions. Generally, models are considered as either atomic models or coupled models. The atomic model can be illustrated as a black box having a set of inputs \( (X) \) and a set of outputs \( (Y) \). The atomic model includes a description of the interface, as well as the data flow between itself and other DEVS models. The atomic model also specifies a set of internal states \( (S) \) with some operation functions, such as the external transition function \( (\delta_{\text{ext}}) \), the internal transition function \( (\delta_{\text{int}}) \), the output function \( (\lambda) \), and the time advance function \( \tau_a \) to describe the dynamic behavior of the model. \( \delta_{\text{ext}} \) carries the input and changes the system states, \( \delta_{\text{int}} \) changes the internal variables from the previous state to the next when no events have occurred since the last transition, \( \lambda \) generates an output event to outside models in the current state, and \( \tau_a \) determines the time of the next internal event after generating an output event. Basic models may be connected in the DEVS formalism to form a coupled model. A coupled model is the major class that embodies the hierarchical model composition constructs of the DEVS formalism. A coupled model is made up of component models, and the coupling relations that establish the desired communication links. A coupled model specifies how to couple (connect) several component models together to form a new model. Therefore, two essential activities involved in coupled models are specifying its component models and defining the couplings that create the desired communication paths\(^{16} \).

3.2 SES and SESBuilder

The SES\(^{17} \) is a high level ontology framework targeted to modeling, simulation, systems design, and engineering. Its expressive power, both in strength and limitation, derive from the domain of discourse. A SES is a formal structure governed by a small number of axioms that provide clarity.

Figure 1. Example system developed in DEVSJAVA
Salas and Zeigler

37

Figure 2. SESBuilder natural language view

and rigor to their models. The structure supports hierarchical and modular compositions allowing large complex structures to be built in a stepwise fashion from smaller, simpler ones. Tools have been developed to transform SESs back and forth to extensible markup language (XML), allowing many operations to be specified in either the SES directly or in its XML guise. The axioms and functionally based semantics of the SES promote pragmatic design and are easily understandable by data modelers. Together with the availability of appropriate tool support, this makes development of XML schema transparent to the modeler. Finally, the SES structures are compact relative to the equivalent schema and automatically generate associated executable simulation models.

SESBuilder is a powerful stand alone application, which supports operations on SESs, see Figure 2. SESBuilder employs the XML to create Pruned Entity Structures (PESs) that represent the logically possible set of world state descriptions consistent with the SES. At the implementation level, a SES is represented by a schema or document type definition (DTD) whose instance documents represent possible pruning. The SESBuilder supports convenient specification of SESs, pruning to create PESs, and transformation to XML representations, all through natural language and graphical interfaces. Thus, it can be used to perform serious data engineering with a powerful multi-tab graphic user interface. It supports a local file system to load and save intermediate and final results for different purposes. The SESBuilder is built on the Java Standard Widget Toolkit (SWT) and could be deployed on either Windows or Unix/Linux-based operating systems.

Some of the highlights of the SESBuilder tool are:

- SES is an effective method of Knowledge Representation and data engineering in support of ontology development languages and environments based on SES axioms;
- SESBuilder provides a data engineering work space based in XML for syntax and validity checks;
- M&S-based data engineering is supported by the SES and the PES at ontology level implemented in XML schemata and XML instances;
- with its M&S-based data engineering approach, the SESBuilder supports the framework for information exchange for the net-centric environment.

3.3 Finite Deterministic DEVS (FDDEVS)

Finite Deterministic DEVS (FDDEVS) formalism was first introduced to restrict the class of DEVS models so as to allow deeper analysis of verifiable properties for the restricted class. The simplicity of the FDDEVS formulation was subsequently recognized to provide a useful abstraction for the development of DEVS models using template-based design.

As seen in Figure 3, the FDDEVS graphical user interface (GUI) allows the creation of Java and XML-expressed FDDEVS models; it allows the user to easily specify the
states, time advance, internal/external transitions, and incoming/outgoing messages for the models to be generated. It also permits one to employ the generated atomic models to create coupled models as described in the previous section.

As an alternative to the template-based design, the creation of FDDEVS models can be done using a constrained natural language input that has the following statement forms:

\[\begin{align*}
&\text{to start hold in PHASE for time SIGMA} ! \\
&\text{hold in PHASE for time SIGMA} ! \\
&\text{after PHASE then output MSG} ! \\
&\text{from PHASE go to PHASE’} ! \\
&\text{when in PHASE and receive MSG go to PHASE’} ! \\
\end{align*}\]

For convenience, some of these statements can be compounded as phrases in the following statements:

\[\begin{align*}
&\text{hold in PHASE for time SIGMA then output MSG and go to PHASE’} ! \\
&\text{hold in PHASE for time SIGMA then go to PHASE’} ! \\
\end{align*}\]

The semantics of these statements are provided in Table 1 along with examples of how to use them.

The semantics of compound statements are composed of the semantics of the individual phrases.

FDDEVS follows an XML schema that facilitates the development of state machines’ internal and external transition behaviors. For instance, the development of a wireless data generator FDDEVS model represented by the state diagram presented in Figure 4 could be translated into the following FDDEVS natural language:

\[\begin{align*}
\text{WirelessDataGenerator: to start passivate in passive!} \\
\text{WirelessDataGenerator: when in phase passive and receive Start go to active!} \\
\text{WirelessDataGenerator: hold in active for time 1 then output WirelessData and go to active!} \\
\text{WirelessDataGenerator: when in active and receive Stop then go to passive!} \\
\end{align*}\]

### 3.4 DEVS/SOA

DEVS/SOA\(^{18}\) is a prototype simulation framework that has been implemented using SOA technology. The central point resides in executing the simulator as a web service. The development of this framework helps to solve large-scale problems and guarantees interoperability among different networked systems and specifically DEVS-validated.
models. DEVS/SOA makes the DEVS simulation process transparent in the model–design cycle, allowing the modeler not to be concerned with the simulator compatibility or any platform issues as in earlier developments, such as DEVS/C++, DEVSJAVA, DEVS/resources management interface (RMI), and DEVS/common object request broker architecture (CORBA). With this simulation service platform the designer is able to execute the model over the Internet through web services, using SOA as the communication protocol. This framework is able to execute DEVSJAVA models, and is extensible to other simulation platforms.

### Table 1.

<table>
<thead>
<tr>
<th>Natural Language Phrase</th>
<th>XFD-DEVS Specification</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>to start hold in PHASE for time SIGMA! where SIGMA is 0, a positive real, or the symbolInfinity (other variants, such as Inf, are allowed)</td>
<td>TimeAdvanceTable(PHASE) = SIGMA and also set PHASE as initial state</td>
<td>to start hold in waitForJob for time Infinity!</td>
</tr>
<tr>
<td>hold in PHASE for time SIGMA where SIGMA 0, a positive real, or the symbolInfinity (other variants, such as Inf are allowed)</td>
<td>TimeAdvanceTable(PHASE) = SIGMA</td>
<td>This can also be expressed as: to start passivate in waitForJob!</td>
</tr>
<tr>
<td>after PHASE then output MSG! from PHASE go to PHASE!</td>
<td>OutputTable(PHASE) = MSG</td>
<td>after sendOut then output Job!</td>
</tr>
<tr>
<td>when in PHASE and receive MSG go to PHASE!</td>
<td>InternalTransitionTable(PHASE) = PHASE'</td>
<td>from sendOut go to waitForJob!</td>
</tr>
<tr>
<td>when in PHASE and receive MSG go to PHASE' eventually!</td>
<td>ExternalTransitionTable(PHASE, MSG) = PHASE' with a mark to note that this input is schedule-preserving in this state</td>
<td>when in waitForJob and receive Job go to processing!</td>
</tr>
</tbody>
</table>

A DEVS/SOA client takes the DEVS models package and, through the dedicated servers hosting simulation services, it performs the following operations:

1. upload the models to specific Internet protocol (IP) locations;
2. run-time compile at respective sites;
3. simulate the coupled-model;
4. receive the simulation output at the client’s site.

The DEVS/SOA client, as shown in Figure 5, operates in the following sequential manner.

1. The user selects the DEVS package folder at his machine.
2. The top-level coupled model is selected as shown in Figure 5.
3. Various available servers are selected. Any number of available servers can be selected.
4. The user then uploads the model by clicking the Upload button. The models are partitioned in a round-robin mechanism and distributed among various chosen servers.
5. The user then compiles the models by clicking the Compile button at the server’s end.
6. Finally, the Simulate button is pressed to execute the simulation using the Simulation service hosted by these services.
7. Once the simulation is over, the console output window displays the aggregated simulation logs from various servers at the client’s end.

In accordance with the model continuity requirement, both logical time (time-managed) and real-time discrete
event simulation are supported by DEVS/SOA. The user can determine the simulation mode of a model by selecting which simulation protocol to use, whether logical or real-time, without changing the model itself.

4. AutoDEVS Description

As indicated earlier, the AutoDEVS has been created to increase productivity in systems development by automating the life cycle process of a system in all the different phases. The name, AutoDEVS, comes from its support for the automation of DEVSJAVA model generation. It provides methodologies to develop systems by generating models and testing models from a spreadsheet containing requirement specifications that turn into an executing real-time system, see Figure 6.

4.1 AutoDEVS Lifecycle

The AutoDEVS process is iterative, allowing one to return to modify the reference master DEVS-model and the requirement specifications. The model continuity minimizes the artifacts that have to be modified as the process proceeds. The design methodology provides a process to transform the requirement specifications to a DEVS representation supporting evaluation and recommendations for a feasible design.

As seen in Figure 7, the AutoDEVS life cycle process combines system theory, the M&S framework, and model-continuity concepts. As illustrated, the tool bifurcates the process into two main streams – system development and test suite development – that converge in the system testing phase. The system development includes the definition of requirements, the capture of specifications to map formalized DEVS model components and to create a reference master model, and use of model continuity to execute the model in the DEVS real-time execution protocol, i.e. SimView. The test suite development includes the development of test models, and the execution of test models against the system under test (SUT) to provide a feasible design from simulation and analysis results\textsuperscript{18}.

4.2 AutoDEVS Tool

The AutoDEVS uses the natural language (English) specification of a SES as the preferred means of specifying discrete event systems and defining the structural aspects of the models being developed: see the Figure 8 ‘SESMicroRepresentation’ column. This natural language is bounded by rules that encompass all of the possible interactions related to any message type. These rules also limit the way the English language is used in terms of removing ambiguous statements.

The basic idea is as follows. The entity is considered as a collection of various message streams. It has been observed in complex systems that an entity node can act as both a receiver and sender simultaneously. It is logical to consider that a node may be processing more than one message at a given instant. Consequently, developing a framework where the entity node model can operate with multiple message streams is the objective of this type of requirement specification\textsuperscript{18}.

The rules that provide a binding to this type of requirement specification are provided in Table 2. The designer can specify each node’s behavior as a sender and a receiver with respect to any specific message type.
The AutoDEVS then uses FDDEVS natural language to describe the behavior aspect of the system being developed. This FDDEVS natural language defines the different states and internal and external transitions between the models being developed as described in Section 3.3.
Table 2. Rules for restricted natural language processing (NLP)-based requirement specifications

<table>
<thead>
<tr>
<th>Rules</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Copula</td>
</tr>
<tr>
<td></td>
<td>‘is’ and ‘are’ are treated the same.</td>
</tr>
<tr>
<td>2</td>
<td>Compounds</td>
</tr>
<tr>
<td></td>
<td>x and y; x, y, and z; NOTE: the commas are mandatory for 3 or more constituents x, y, z, and w; x or y; x, y, or z; x, y, z, or w;</td>
</tr>
<tr>
<td>3</td>
<td>Determiners</td>
</tr>
<tr>
<td></td>
<td>‘a’, ‘the’, … are removed from the input before processing</td>
</tr>
<tr>
<td>4</td>
<td>End of Sentence</td>
</tr>
<tr>
<td></td>
<td>use ‘!’ instead of ‘.’</td>
</tr>
<tr>
<td>5</td>
<td>Sentence Order</td>
</tr>
<tr>
<td></td>
<td>• The entity mentioned first in the first sentence becomes the root of the SES.</td>
</tr>
<tr>
<td></td>
<td>• A variable must be attached to an entity before giving it a range specification (see below).</td>
</tr>
<tr>
<td></td>
<td>• Otherwise sentences can be in any order.</td>
</tr>
<tr>
<td>6</td>
<td>Forms</td>
</tr>
<tr>
<td></td>
<td>In the following, CAPITALs indicate variables, lower case indicate mandatory key words in the order shown.</td>
</tr>
<tr>
<td>7</td>
<td>Specialization</td>
</tr>
<tr>
<td></td>
<td>THING can be VARIANT1, VARIANT2, or VARIANT3 in CLASSFAMILY!</td>
</tr>
<tr>
<td>8</td>
<td>Aspect</td>
</tr>
<tr>
<td></td>
<td>From VIEW perspective, THING is made of COMPONENT1, COMPONENT2, and COMPONENT3!</td>
</tr>
<tr>
<td>9</td>
<td>MultiAspect</td>
</tr>
<tr>
<td></td>
<td>From a multiple perspective, THINGS are made of more than one THING!</td>
</tr>
<tr>
<td>10</td>
<td>Attached Variables</td>
</tr>
<tr>
<td></td>
<td>THING has VAR1, VAR2, and VAR3!</td>
</tr>
<tr>
<td>11</td>
<td>Range specifications of variables</td>
</tr>
<tr>
<td></td>
<td>The range of THING’s VAR1 is RANGE!</td>
</tr>
</tbody>
</table>
See the ‘FDDEVSSRepresentation’ column of Figure 8. It contains the behavioral aspects of the system under development. The AutoDEVS makes use of this column to automate the development of the behavioral aspects for the DEVS models being created.

After constructing the structural and behavior aspects of the system, the AutoDEVS then executes automatic pruning on the models created, based on the different specialization identified, see Figure 9. The PES can be transformed into a composition tree, see Figure 10, and eventually synthesized into a simulation model. The AutoDEVS also provides the capability to modify the PES created by the tool and then simulate it to validate the models pruned, see Figure 11.

After the AutoDEVS has created all the models from the derived requirement specifications, it allows the user to automatically create DEVS test models for the system being developed. These test models are created with the objective of verifying the correctness of the DEVS models. The test methodology is based on minimal testable I/O pairs restricted to messages, and assuming they are the only automatable observables available for testing. The DEVS test models are in the form of an experimental frame and allow the developer to perform experiments against the SUT. The test engineer analyzes the requirements (from the spreadsheet) and creates the test scenarios that describe the behaviors of the SUT. The requirements are written in minimal testable I/O representation, and the test models are created by applying the model mirroring concept that reverses the minimal testable I/O pairs. Both the minimal testable file and test models are written in XML format and represented by a SES, allowing for the transformation between the two XML files. The I/O pairs are now represented by three primitive atomic models: holdSend, waitReceive, and waitNotReceive. Since the I/Os are in sequential order, only one atomic model is active each time, and the rest of the atomic models are passive. In order to try out these test models against the real system, they are converted to software programming source code; refer to Mittal18 for more details. This testing methodology is still under development and will be available soon in the AutoDEVS tool.
Finally, the AutoDEVS tool provides an interface to the FDDEVS tool, SESBuilder, and DEVSJAVA SimView. Refer to Section 4 for a description of FDDEVS and SESBuilder tools. The DEVSJAVA SimView is a program included in the DEVSJAVA framework to view and check that all the models and couplings are built as expected. In addition, SimView allows the developers to run, observe and evaluate the real-time execution of the system under development, see Figure 12.

As seen in Figure 12, SimView allows the user to stop, run and restart the execution of the model being simulated. This helps the developers detect flaws easily when simulating and testing the system, as it shows the transactions and state transitions for each of the models that are being executed. SimView provides the flexibility to choose and run the desired model and simulate different models without having to reopen the SimView application. In addition, DEVSJAVA SimView allows the developer to simulate hierarchical models. These models are coupled models with components that may be atomic or coupled models that constitute a part of the entire system. This permits one to simplify the scope of testing and find problems at different stages of the development, i.e. execute unit simulation to analyze models more concretely.

One very useful feature of the AutoDEVS tool is that it displays the model’s debugging messages into an execution log textbox, which is integrated in the tool. This permits the developer to quickly debug the system and resolve any bugs encountered during simulation or to just monitor the application in execution. SimView also provides the capability to adjust the simulation speed of the models by increasing or decreasing the execution time by a scale factor. For example, a time scale factor equaling 1 means that the simulation will run at the same speed as a wall clock; a time scale factor equaling 0.5 means that the
simulation will run twice as fast as the wall clock, and so on. This allows the designer to analyze the data to determine if the SUT fulfills the logical behavior as desired.

4.3 AutoDEVS and Model Continuity

The AutoDEVS is a software development methodology that supports model continuity for distributed real-time software development. This methodology is based on the DEVS M&S framework. Corresponding to the general ‘Design–Test–Execute’ development procedure, this tool provides a ‘Modeling–Simulation–Execution’ process that includes several stages to develop real-time software. During these stages, a model’s continuity is maintained because the same control models that are created will be tested by simulation methods and then deployed to the target system for execution.

Next is a description of the different stages that the AutoDEVS methodology utilizes to develop systems showing model continuity. This description takes advantage of a simple Agent Development System example. The first stage of the AutoDEVS methodology is defining user requirements for the Agent Development System.

As seen in Figure 13, the requirements for the new system are collected and organized in a spreadsheet table, i.e. the ‘RequirementText’ column. The second stage is describing the structural aspects for each of the requirements of the Agent Development System, i.e. fill the ‘SESMicroRepresentation’ column of the spreadsheet. The third stage is describing the behavioral aspects for each of the requirements, i.e. fill the ‘FDDEVSRepresentation’ column of the spreadsheet. The fourth stage is defining the multi-aspect coupling for the coupled models, which automates the coupling generation for multi-aspect models. This is defined in
Figure 13. Agent development example: define requirements

Figure 14. Agent Development example: capture spreadsheet data
the ‘MultiCouplingTest’ column of the spreadsheet, i.e. agent I/O coupling. The fifth stage is running the AutoDEVS tool to capture the spreadsheet data, i.e. agentDevelop.xls, see Figure 14.

Notice that the user needs to set the folder and spreadsheet file to be captured in the AutoDEVS tool, i.e. the ‘Set Folder’ and ‘Select File’ buttons. In addition, notice that subsequent to capturing the data from the spreadsheet, the AutoDEVS encodes the captured data into an XML schema definition (XSD)/DTD, i.e. Sheet1agentDevelopRowsSchema.xsd, ses.dtd. The sixth stage is to generate FDDEVS models based on the schema-type definition and the captured XML data from previous stage, i.e. the behavioral aspects of the Agent Development System (FDDEVSRepresentation column).

As seen in Figure 15, the DEVS java models are automatically generated, including an XML representation of those models, i.e. observer.java, observer.xml. Based on the MicroSESRepresentation column defined in the spreadsheet, the seventh stage is to add the structural aspects on the DEVS models created in the previous stage. During this stage, a SES representation of the models is created and parsed into an XML file, i.e. agentDevelopagentCoupledModsSeS.xml. Notice that this file could serve to see the SES representation as a tree view, see Figure 16. In addition, an automatic PES that represents the logically possible set state descriptions consistent with the SES is created, i.e. agenDevCoupModInst.xml.

The eighth stage is to run the PES that was automatically created in the previous stage, i.e. Transform PESToDEVS.

Figure 15. Agent Development example: generate FDDEVS
During this stage, the specialized models are created and the DEVS models are updated with the corresponding structural aspects, i.e. the PES is transformed into DEVS-JAVA models. This PES can also be modified by the developer to create individual pruning and to analyze the models of interest. The AutoDEVS tool allows one to choose the PES desired and then run it using the tool, as shown in Figure 17.

The ninth stage is to create a set of test models to validate the system under development, see Figure 18. As seen in Figure 18, the AutoDEVS lets the developer choose the test models to create from a list given. As described previously, these test models are based on minimal testable I/O pairs restricted to messages and are created with the objective of verifying the correctness of the DEVS models. This feature is being developed and will be available in the near future.

The tenth stage is to verify the models created in the FDDEVS models and SES Builder DEVSJAVA SimView applications, see Figure 19. Then, the models are modified according to the desired needs and start simulation, see Figure 20.

### 4.4 Mapping AutoDEVS-generated Models to DEVS/SOA

The DEVS/SOA tool, as described in Section 3.5, facilitates the developer to create transparent distributed simulation. It exploits model continuity by taking the generated
AutoDEVS models and allowing the upload, compilation and simulation of the developed models in a distributed way among different available servers. The AutoDEVS and DEVS/SOA tools minimize the number of modifications required to the developed source code to make the distributed simulation possible, see Salas24 for details. By distributing the simulation of the system developed in different machines, the memory requirements on any single system can be substantially smaller than the memory used in a single-workstation simulation. The developer needs to consider optimizing the transactions between the different workstations to allow the overall execution time of the simulation to be at least as fast as the original single-workstation simulation. Networking or distributing the simulation of the generated AutoDEVS models using the DEVS/SOA tool encourages improvement of the speed of highly-parallelizable tasks by distributing pieces of the system across many computers that together form a distributed-computing simulation system.

Furthermore, the AutoDEVS together with the DEVS/SOA provide an infrastructure to implement structural and hierarchical real-time distributed systems. In particular, DEVS/SOA supports the fourth step where all the models are deployed to their target platforms and tested in the real physical environment. In this case, the target platforms and physical environment are the SOA services and underlying network infrastructure in which the DEVS is implemented. The user can determine the simulation mode of a model by selecting the simulation protocol to use, whether logical or real-time, without changing the model itself. When executed under the real-time simulation model, the model effectively becomes an executing web service that operates in real-time and can be distributed to geographically dispersed nodes as desired. An example application to support negotiation is given by Hwang and Ziegler22.

4.5 AutoDEVS Applications to M&S for Defense Testing and Evaluation

M&S is finding increasing application in defense, such as the incorporation of M&S functionality into C2 systems25 and the use of M&S to support the development and testing of System of Systems (SoS). The SoS concept relates to the attempt to integrate disparate systems to achieve new requirements with the defining concern being interoperability, or lack thereof, among the constituent systems25,26. Achieving such interoperability is among the chief SoS engineering objectives in the development of C2 capabilities for joint and coalition warfare27,28. The DEVS/SOA operation over web middleware enables it to operate within the net-centric environment of the Global Information Grid (GIG)/SOA, which is being developed to achieve a new level of defense interoperability. The development of the AutoDEVS enhances the quality of solutions provided by the DEVS technology to support the Test and Evaluation...
Figure 19. Agent Development example: verifying models in SES, FDDEVS, and SimView

Figure 20. Agent Development example: running models in SimView
(T&E) of interoperability in defense SoS, particularly in net-centric environments such as GIG/SOA\textsuperscript{29,36}. Since the AutoDEVS can support model continuity through a simulation-based development and testing life cycle, the mapping of high-level requirement specifications into lower-level DEVS formalizations enables such specifications to be thoroughly tested in virtual simulation environments before being easily and consistently transitioned to operate in a real environment for further testing and fielding. Indeed, the DEVS Unified Process\textsuperscript{18,31} provides methodology and SOA infrastructure for integrated development and testing. Extended Department of Defense Architectural Framework (DoDAF) views support executable architectures using M&S for application to mission-based testing for GIG/SOA\textsuperscript{31,34,35}. The DEVS simulation architecture is layered to support technology neutrality and execution in different technological scenarios\textsuperscript{31,34,35}. Separating a model from the act of simulation itself, which can be executed on single or multiple distributed platforms, supports automated test generation and deployment in distributed simulation\textsuperscript{36}.

5. Conclusions

We have presented a new DEVS-based tool, called the ‘AutoDEVS’, which automates systems development and exploits ‘model continuity’ to maintain coherence through the development process and to support test artifact continuity and traceability through phases of system development. The AutoDEVS creates models and systems that can be mapped to DEVS/SOA, a DEVS implementation compliant with a SOA infrastructure, such as that adopted by the DoD’s GIG initiative. Therefore, it provides a rapid means of integrated development and testing of defense C2 systems for interoperability.

Achieving interoperability is one of the chief SoS engineering objectives in the development of C2 capabilities for joint and coalition warfare. The importance of M&S in SoS design and evaluation cannot be underestimated. M&S can be used strategically to provide early feasibility studies and aid the design process. As components comprising SoS are designed and analyzed, their integration and communication is the most critical part that must be addressed by the employed SoS M&S framework. The integration infrastructure must support interoperability at syntactic, semantic and pragmatic levels to enable such integration\textsuperscript{17,28}.

Currently, there are several other approaches to distributed simulation and to integration of M&S with advanced C2 systems. These approaches build on the Internet or other net-centric middleware to provide component connectivity and simulation services\textsuperscript{25,37}. The DEVS provides a formal systems-based abstraction that can support higher-level interoperability, whether alone or on top of HLA. The DEVS/SOA implementation provides a SOA implementation independent of HLA and is a viable approach to M&S integration with C2 SoS. The DEVS components, including decision-making agents, sensor simulators, and environmental representations, can bring the power of M&S to the development of C2 SoS, as well as serving as support for C2 in real operation. The underlying SOA standard that facilitates this interoperability can be expected to be widely implemented following its adoption by the DoD’s GIG initiative.

References


33. Mittal S. Extending DoDAF to allow DEVs-based modeling and simulation. *J Def Mod Sim* 3: 95-123.


**Author Biographies**

**Manuel C. Salas** has been a research assistant at the Arizona Center for Integrative M&S and a Software Engineer for Product Development at Modular Mining Systems in Tucson, Arizona since 2006.

**Bernard P. Zeigler** is Professor of Electrical and Computer Engineering at the University of Arizona, Tucson and Director of the Arizona Center for Integrative M&S and Chief Scientist at RTSync Corp. He is internationally known for his 1976 foundational text Theory of M&S, recently revised for a second edition (Academic Press, 2000). He has published numerous books and research publications on DEVs formalism. In 1995, he was named Fellow of the IEEE in recognition of his contributions to the theory of discrete event simulation. In 2000 he received the McLeod Founder’s Award by the Society for Computer Simulation, its highest recognition, for his contributions to discrete event simulation. In June 2002, he was elected President of The Society for M&S, International, and he was appointed Fellow of the Society in 2006. Dr. Zeigler is developing DEVs-methodology approaches for testing mission thread end-to-end interoperability of Defense Department acquisitions and transitions to the GIG with its SOA (GIG/SoA).