Simulation

Simulation: Transactions of the Society for Modeling and Simulation International
0(0) 1–24
© 2011 The Society for Modeling and Simulation International
DOI: 10.1177/0037549711436267
sim.sagepub.com
SAGE SCS

Graphical modeling and simulation of discrete-event systems with CD ++ Builder

Matías Bonaventura[1], Gabriel A Wainer[2], and Rodrigo Castro[1]

Abstract
We introduce CD ++ Builder, an open-source environment that aims at providing easy-to-use graphical modeling tools to simplify the construction of models and the execution of simulations of complex Discrete Event System Specification (DEVS) models. The architecture and implementation of CD ++ Builder focuses on providing simple definition and reuse of components, offering easy extensibility to support new features. CD ++ Builder includes graphical editors for DEVS-coupled models, DEVS-Graphs and C ++ atomic models; it provides code templates that are synchronized with their graphical versions, and it greatly simplifies the software installation and update procedures. We show how this environment can be used to build and simulate DEVS models, and we compare the process with previous versions and other simulation tools, showing that CD ++ Builder can improve model development by creating DEVS models in a completely assisted manner, including advanced graphical interfaces.

Keywords
Discrete Event System Specification formalism, open-source modeling and simulation tools, graphical simulation environments, Discrete Event System Specification-Graphs, Cell-Discrete Event System Specification, CD ++

1. Introduction

The DEVS (Discrete Event System Specification) formalism[1] is a sound formal framework (based on generic dynamic systems theory concepts) that provides a discrete-event approach that allows models to be defined in a hierarchical and modular manner. DEVS defines the models and their abstract simulation mechanisms independently from each other and from the underlying hardware and middleware. In recent years, DEVS has become very popular for modeling and simulation (M&S) of complex systems, and numerous DEVS simulators have been implemented using diverse technology.[2,3] DEVS has also been used successfully in diverse areas, ranging from natural systems to human-made dynamic systems.[4–7]

In most DEVS M&S environments, model behavior and structure are defined using high-level programming languages (such as C ++ or Java), making modeling more difficult for non-expert developers, and introducing difficulties for model validation. DEVS (and other) simulation tools can be difficult to extend with new features, as sometimes they are developed from the ground up without using standard user interfaces and technologies. Likewise, each DEVS simulator usually defines its own specific programming structures and Application Program Interface (API), and users need to learn these implementation details. Although these DEVS tools are based on the same formal concepts, the use of different programming languages makes it more difficult to reuse existing models in other DEVS M&S tools. They are also difficult to extend with new functionalities.

Some DEVS environments deal with these issues by providing graphical modeling capabilities for structural models, tools for tracking and animating simulations and code structure aids for programming model behavior.[3,8,9] We here present an open-source environment that tackles these problems in a different way. The environment, called CD ++ Builder,[10] provides aids to support different languages for specifying DEVS models declaratively and

[1]Computer Science Department, Universidad de Buenos Aires, Ciudad Universitaria, Argentina
[2]Department of Systems and Computer Engineering, Carleton University Centre of Visualization and Simulation (V-Sim), Canada

Corresponding author:
Matías Bonaventura, Computer Science Department, Universidad de Buenos Aires, Ciudad Universitaria, Pabellón I, (C1428EGA) Ciudad Autónoma de Buenos Aires, Argentina.
Email: abonaven@dc.uba.ar

graphically for the CD++ DEVS simulator.[2] In this way, the barrier entrance for non-developer users is reduced. DEVS atomic model behavior can be defined graphically based on the DEVS-Graph[11] notation, a declarative syntax that can potentially be used by other tools as a means for sharing models.

To solve the aforementioned issues, CD++ Builder integrates several tools that are available using a single Eclipse Integrated Development Environment (IDE),[12] thus reducing the learning curve for new users. The plug-in architecture of Eclipse makes it possible to add new features easily, and it also provides installation and update mechanisms, which can be used to reduce environment setup efforts and simplify distribution of new versions.

In order to improve reuse and extensibility of the environment, we used Eclipse graphical editor generator frameworks, which provide intuitive and easy-to-use interfaces, while guaranteeing the reuse of components.

CD++ Builder avoids using different applications and diverse formats, simplifying the M&S workflow. Compilation of C++ source code is automated, and code templates are used to avoid repetitive and error-prone tasks, providing sample structures that promote good modeling practices. Graphically defined DEVS models can coexist with other models developed in C++, which gives more flexibility to the modeler: structural and simple behavioral models can be defined graphically, while C++ can be used to implement behavior that is more complex. The environment also makes easy the definition of automated regression tests, which are important when new functionalities are included in the environment, providing a mechanism to verify the behavior of the software after new code is introduced. We will discuss the advantages of this approach in the following sections.

The rest of the paper is organized as follows: Section 2 introduces the DEVS formalism, the DEVS-Graphs notation and different simulation tools. Section 3 shows the CD++ Builder architecture and the technologies that helped solve previous tool limitations. Section 4 provides implementation details and it presents an overview of the CD++ Builder environment, its graphical editors and main features. Section 5 demonstrates the M&S process by showing the creation of a sample model using CD++ Builder. Section 6 presents a discussion about the results and conclusions.

2. Background

In this section, we introduce the main features about different DEVS tools and the DEVS formalism; we also discuss the graphical specification of DEVS atomic models and DEVS-Graphs. We finally give a brief introduction to the CD++ toolkit.

2.1 The DEVS formalism and DEVS tools

DEVS[1] is a M&S formalism based on systems theory concepts for modeling both discrete and continuous worlds. DEVS formal specifications provide the means for mathematical manipulation of the models, and permit independence of the language chosen to implement them.[1,2] To attack system complexity, DEVS models use a hierarchical composition of behavioral models (atomic) and structural models (coupled) with well-defined modular interfaces.

The DEVS formalism has been thoroughly discussed in the literature; the reader can find a detailed description in Appendix I.

AS DEVS is independent from any simulation mechanism, several simulation tools have been developed by different groups, each tackling different needs and providing advantages on specific applications. In order to define an atomic DEVS model, we need a mechanism to specify the model's behavior (specified by the functions δ_{int}, δ_{ext}, λ and ta described in the Appendix), for which tools provide different aids to implement them. Likewise, most tools provide mechanisms to define coupled models. A non-exhaustive list (the interested reader can find a comprehensive list at http://cell-devs.sce.carleton.ca/devsgroup/) of those DEVS simulators includes the following.

- ADEVS,[13,15] a C++ library for developing discrete-event simulations based on the parallel DEVS and Dynamic Structure Discrete Event System Specification (DSDEVS) formalisms. It includes support for standard, sequential simulation and conservative, parallel simulation on shared memory machines with Portable Operating System Interface for Unix (POSIX) threads.
- DEVSJAVA,[3] a DEVS simulator that provides Java classes for the users to implement their own models. Four Java packages separate M&S from user interface. It allows parallel and distributed simulation, hierarchical model definition and visualization. New DEVS-coupled and atomic models are built by extending one of the base Java classes provided by the framework. SimView, a graphical component of DEVSJAVA, allows the user to specify the model layout (this must be done in the same model's source code, making model behavior more difficult to understand). During the simulation execution, synchronization messages sent between models are displayed.
- DEVSim++,[15] an object-oriented environment for M&S of discrete-event systems. The software includes numerous support tools: VeriTool (for DEVS model verification), DEVSimHLA (a library to support high-level architecture (HLA)), PlugSim (a distributed simulation framework plug-in) and others.

- SimStudio,[9] a web-based framework implemented using Java web technologies, using a layered architecture that supplies modeling, visualization and analysis players. The modeling plug in, which is implemented in Flash, allows one to specify a graphical model, and to generate an Extensible Markup Language (XML) file that is used by other tools.
- PowerDEVS,[8] which allows one to specify coupled DEVS models graphically, and atomic models in C++. A special editor aids the modeler with code structure, and a model library enables model reuse in a drag and drop fashion. Tracking model state during simulation is done by special atomic models that interact with outside devices. Although this approach is useful, model definition and simulation tracking are mixed into the same editor.
- JAMES II,[16] a JAva-based Multipurpose Environment for Simulation. This tool includes different formalisms, including DEVS. It is designed to be adapted as a back-end into any existing software. It provides a graphical user interface (GUI), an experimentation layer, and it uses an open architecture, distributed as open source.
- DEVS/SOA,[17] a Java implementation of DEVS over a Service Oriented Architecture (SOA) framework. The framework provides runtime composability of coupled systems using SOA. We describe the architecture and designs of the server and the client.
- CD++[2,18] implements DEVS and Cell-DEVS theories, and it has been widely used in several areas of interest, such as urban traffic, physical systems, computer architecture and embedded systems.[2] CD++ is implemented in C++ as a class hierarchy, where models are related to simulation entities. Atomic model behavior is programmed in C++, and coupled models are defined in a model definition file using a built-in high-level language.

2.2 Graphical specifications

The DEVS simulators discussed in the previous section (and many other similar ones that have been implemented) use the abstract DEVS simulation algorithms and provide different APIs to define new models. In general, model behavior must be implemented in some programming language (such as Java or C++), making it more difficult for non-expert developers. Most of these DEVS simulators define their own programming structures and APIs, and users need to learn these implementation details. Although these DEVS tools are based on the same formal concepts, the use of different programming languages makes it more difficult to reuse existing models in other DEVS M&S tools.

Many of these tools provide graphical aids to define the coupled model structure and to analyze the simulation results, but no descriptive high-level language is supported to specify the atomic model behavior. In some cases, tools provide specialized editors and generate code structures to help programming new atomic models, but they lack complete integration with coupled model editors; therefore, code updates are restricted or not kept consistent with the graphical representations. The absence of descriptive languages makes it very difficult to use models on different simulators interchangeably, and it forces users to learn specific implementation APIs. On the other hand, most of the tools have been developed from the ground up using diverse technologies (such as Java, .Net, C++, etc.), making it hard to leverage existing features.

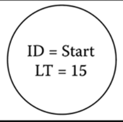
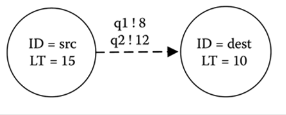
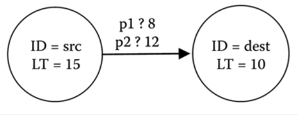
To deal with these issues, CD++ supports the definition of atomic models using DEVS-Graphs, an extension to the original DEVS atomic model.[11] This graphical notation allows one to define the behavior of atomic models, and it is depicted in Figure 1.

The graphical specification shown in Figure 1 can improve interaction with stakeholders during system specification, and it has the advantage of allowing the modeler to think about the problem in a more abstract way.[2] This graphical notation can be formally specified as

$$\text{DEVS} - \text{Graphs} = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, D >$$

DEVS-Graphs models represent atomic model state changes using a graph-based specification, where bubbles represent each state $s \in S$ (including an identifier and the

Definition	States	Internal transitions	External transitions

Graphical		 ID = src
		LT = 15
		q1 ! 8
		q2 ! 12
		ID = dest
		LT = 10
			 ID = src
			LT = 15
			p1 ? 8
			p2 ? 12
			ID = dest
			LT = 10

Textual	state: stateId stateId: lifetime initial: stateId	int: src dest {q ! v}* {(action;)*}	ext: src dest exp ([p?v]*) {(action;)*}

Figure 1. DEVS-Graphs graphical notation.[2]

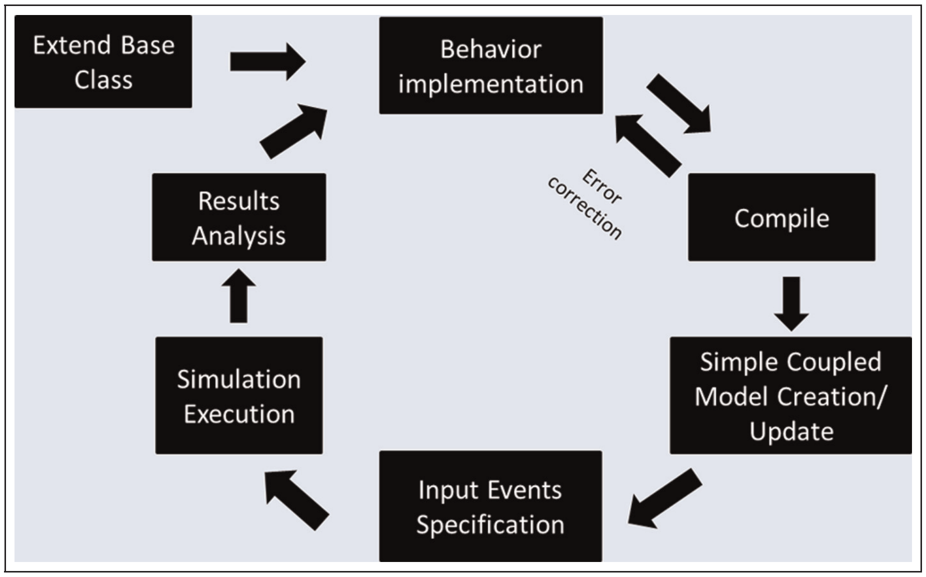


Figure 2. Workflow for the definition and simulation of C++ models in CD++.

state lifetime), dashed arrows represent the internal transition function δ_{int}, and full arrows represent the external transition function δ_{ext}, as shown in Figure 1. When the lifetime of a state is consumed, the model will change its state by executing an internal transition function, which can be associated with a pair of port/value $(p, v) \in Y$ that represents the output function λ. When an external event $(p, v) \in X$ occurs, the external transition whose condition is satisfied is executed. Our DEVS-Graphs notation also allows the definition of temporary state variables as $S = B \times P(V)$, where B is a set of 'Bubbles' (Figure 1), and V is the set of variables and values. Any number of state variables can be defined and internal and external transitions have an optional list of actions (depicted as {(action;)*} in Figure 1) to update the values of these variables. In addition, state variables can be used to specify the condition for external transitions (depicted as [p?v] in Figure 1), which are used to decide if the transition is be executed or not. Likewise, different output values can be associated to internal transitions (which is defined as a list of output ports and their values, depicted as {q ! v}* in Figure 1). Values used in conditions, actions and output values can be any valid expression formed by variables, constants, input ports and functions that can be composed to specify more complex expressions. Appendix II shows the complete grammar of the DEVS-Graph language and a simple example of model definition.

The DEVS-Graphs notation needs to be converted into an executable specification that can be used for computation. Figure 1 also presents an equivalent textual notation that can be used with this purpose,[19] in which the model name is noted as [*modelname*], and the input and output ports are specified using the *in* and *out* statements, respectively. The *state* construction shown in Figure 1 declares all the state identifiers; the *lifetime* of each state is then assigned to the corresponding identifier in a separate statement, and one of the states must be declared as the *initial* state of the model. The keyword *int* is used to define internal transitions, indicating the *source* and *destination* states, a list of *actions* and a list of output events denoted as *q!v* (i.e. sending value *v* through port *q*). Similarly, the keyword *ext* is used to define external transitions, indicating the *source* and *destination* states, a list of *actions* and an expression that must be satisfied for the transition to be executed. Actions and conditions can be specified as simple mathematical expressions, or they can be implemented in user-defined C++ functions, providing a flexible mechanism for defining complex model behavior. The keyword *var* allows defining temporary variables.

When defining new atomic models in CD++ and other DEVS frameworks, the users typically follow the workflow described in Figure 2. New atomic models must implement a base class provided by the framework and the model behavior must be programmed in an object-oriented programming language (such as C++, C# or Java). Afterwards, the complete simulator must be recompiled to include the new model. In order to test the implementation, a coupled model containing the new component and a set of test input events must be specified. Depending on the simulation results, the implementation might need corrections and the process should be repeated. This includes repetitive and error-prone tasks, such as the base class implementation and the compilation process.

3. CD++ Builder architecture

To solve the varied limitations introduced in Section 2, we designed and implemented CD++ Builder,[2,10] an

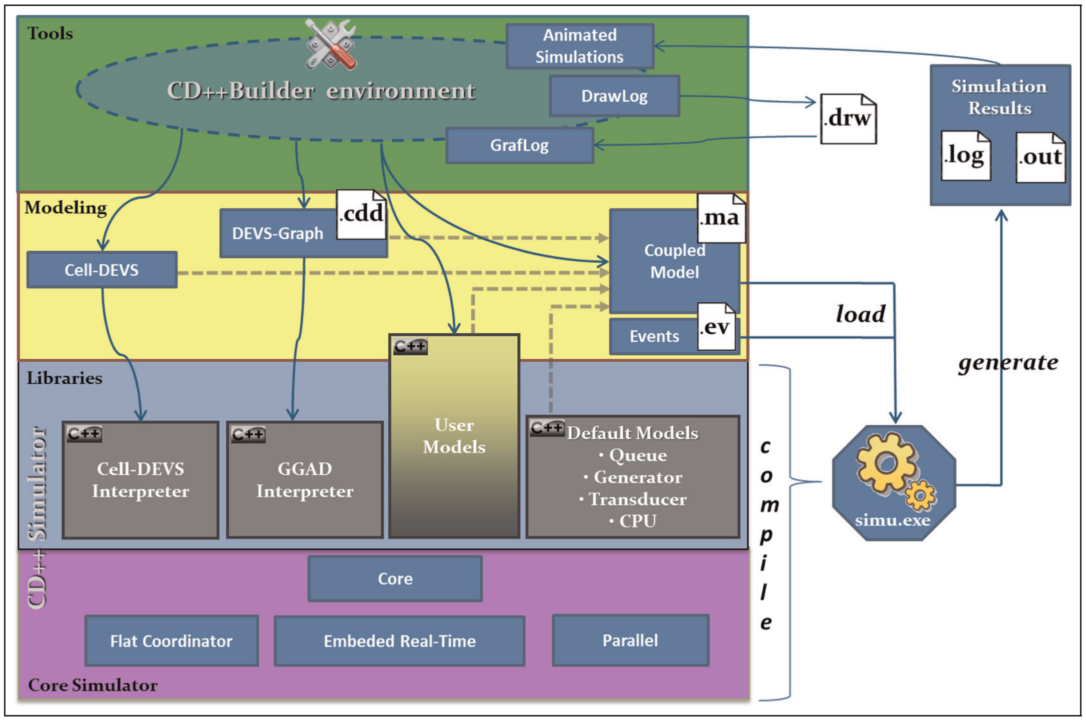


Figure 3. CD + +, high-level modeling languages, execution process and supporting tools.

environment based on the previously existing CD++ toolkit.[2,18] The software is open source and it is based on Eclipse's plug-in infrastructure, making it possible to be adapted to other M&S software tools. In this section, we discuss the overall architecture of this software application.

3.1 Architecture components

In order to achieve the objectives discussed in Section 1 and to deal with the problems discussed in Section 2, we introduced a layered architecture with a clear separation between simulation execution, model definition, supporting tools and underlying libraries, as shown in Figure 3. This allows one to modify the simulation runtime without affecting the models, tools or visualization engines that already exist. The same tools and interfaces can be used to facilitate model definition, whether those models will run in single processor, parallel, distributed or embedded environments.

Figure 3 depicts the role of each CD++ component and their main relationships. At the lowest layer (on top of the Operating System), the Core Simulator layer implements the different versions of the abstract simulators algorithms (e.g. parallel, flat, real time). The next level (Libraries) provides different language interpreters, and basic out-of-the-box atomic models that can be directly used to define coupled models. The interpreters accept input files coming from the Modeling level, which define Coupled Models, Cell-DEVS models (both interpreted by the Cell-DEVS Interpreter) or DEVS-Graph atomic models (interpreted by the GGAD interpreter). Other area-specific interpreters are also available, such as ATLAS for describing urban traffic[20] or M/CD++[21] for describing continuous models using Bond Graphs and Modelica.[2] High-level languages from the Modeling layer are independent from the Core and Libraries layers, so they can be used independently of the simulator version. When a custom atomic model behavior needs to be defined, it can be done with User Models (which extend the Atomic C++ base class of the framework, after which recompiling CD++ is required), or using the DEVS-Graphs notation presented in Figure 1. To execute a simulation, the users choose varied options (coupled model definition, input events, etc.). The simulation results are stored on two output files (*.out* files contain port-value pairs for the output events of the model and *.log* files contain all the message passing and synchronization information between different models).

At the top Tools layer, different applications have been developed to facilitate output file visualization, such as Drawlog for Cell-DEVS models and CD++ Modeler to animate Cell-DEVS, coupled model message passing and atomic model output values. CD++ Builder also provides varied graphical editors to specify the model behavior, and to generate the high-level specifications to be used by the lower layers.

This architecture is extensible, and it allows the inclusion of other new tools into CD++ Builder. Other simulators might take advantage of already developed components, and some of the graphical editors' functionality could be reused for simulators supporting high-level language modeling. On the other hand, CD++ high-level languages could be easily updated to interpret new model notations, reusing existing graphical capabilities.

The hierarchical architecture of CD++ Builder was implemented using several well-known Eclipse frameworks that provide the overall user interface and core plug-in services. Eclipse provides several frameworks to implement the graphical editors, including the Standard Widget Toolkit (SWT), the Abstract Window Toolkit (AWT) and Swing. Other more specific editors include Draw2D and the Graphical Editing Framework (GEF).[22] The first three libraries are based on Java and they provide general GUI controls, which are useful for building form windows. Nevertheless, they are not practical for manipulating figures and shapes, and they do not provide any special infrastructure for Eclipse-based editors. Figures are the building blocks for Draw2D, which builds on top of the SWT library. The GEF allows one to generate graphical editors based on an existing application model. Due to these reasons, we chose Eclipse's Graphical Modeling Framework (GMF),[23] as this library acts as a bridge between GEF and the Eclipse Modeling Framework (EMF),[24] and it specifically tackles the creation of graphical Eclipse-based editors. The GMF also relies on the Model-View-Controller (MVC) architectural pattern to separate the model from its graphical representation, which has been successfully used in other editors. A similar software stack has also been used by Ehrig et al.[25] for graphical editors of visual languages.

3.2 CD++ Builder software stack

The Tools layer presented in Figure 3 makes use of the different tools and software packages shown in Figure 4, which were used to implement the graphical editors that support the high-level languages in the Modeling layer. CD++ Builder integrates the Eclipse IDE and the CD++ tools by means of graphical editors implemented using the EMF, GEF and GMF. The EMF provides several services for specifying, maintaining and persisting entities, and it is used to specify Java classes that represent *model entities* (i.e. Links, Models, Ports, etc.; the *model* part of MVC, used by the IDE to define the behavior of the editors). These model entities can be specified in the XML Metadata Interchange (XMI)[26] format, or using a graphical editor. The EMF uses these to generate Java classes and interfaces that describe the behavior and rules for each entity. These classes are independent from any graphical implementation, and they do not store any graphical information (that represents each entity, colors, positions, etc.). Custom code and methods can be added to these classes in order to provide extra behavior to the *model* portion of the architecture. The EMF recognizes special code comments in customized methods, so that they are not overwritten

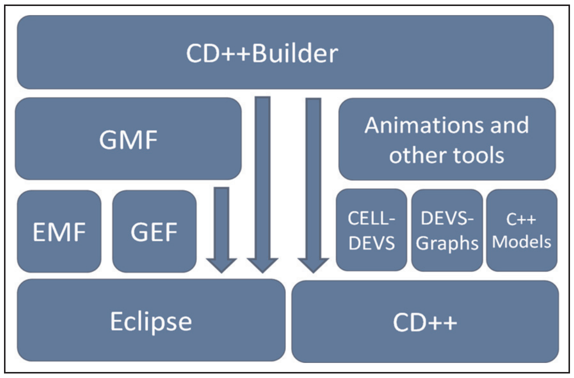


Figure 4. CD++ Builder technology architecture.

when the *model* is regenerated. The EMF also provides persistence and validation services for generated models. The detailed description of the model used to represent DEVS entities in CD++ Builder is discussed in Section 4.

The GEF and GMF leverage the classes generated by the EMF to supply the graphical features of CD++ Builder. These frameworks provide base classes, which we extended to implement the *view* and *controller* classes of the MVC pattern. The GEF extends Draw2D to make it easier to create a graphic representation of the model and provides base Eclipse editor implementations. The GMF provides a generative component and runtime infrastructure for developing graphical editors based on the EMF and GEF.[22] The GMF runtime can be seen as a white-box framework combining EMF-generated models with the GEF's controllers and views, and providing additional services (i.e. transactional support). GMF code generation can be seen as a black-box framework defining meta-model information in XML files, which are used afterwards to generate Eclipse editors' code.

We used GMF code generation to define the general look and feel, layout and behavior of the editors (many new features were added, customizing and extending the code generated by the GMF as explained in Section 4). The GMF generates a decoupled infrastructure where controller, view and editor implementation are separated from the model. This suits CD++ Builder's requirements, as the model can be reused by other CD++ or DEVS plug-ins without depending on the editor implementation. The model is independent from graphical and edition details. The graphical editor information, which consists of model information provided by the EMF and graphical data generated by GMF editors, is stored in XMI format, but persistence services can be updated to use any other suitable format. This could be useful in scenarios where DEVS models need to be shared between different simulators.

3.3 Installation and updates

Installation and updates are centralized using the Eclipse Update Manager. This allows hosting plug-in compiled

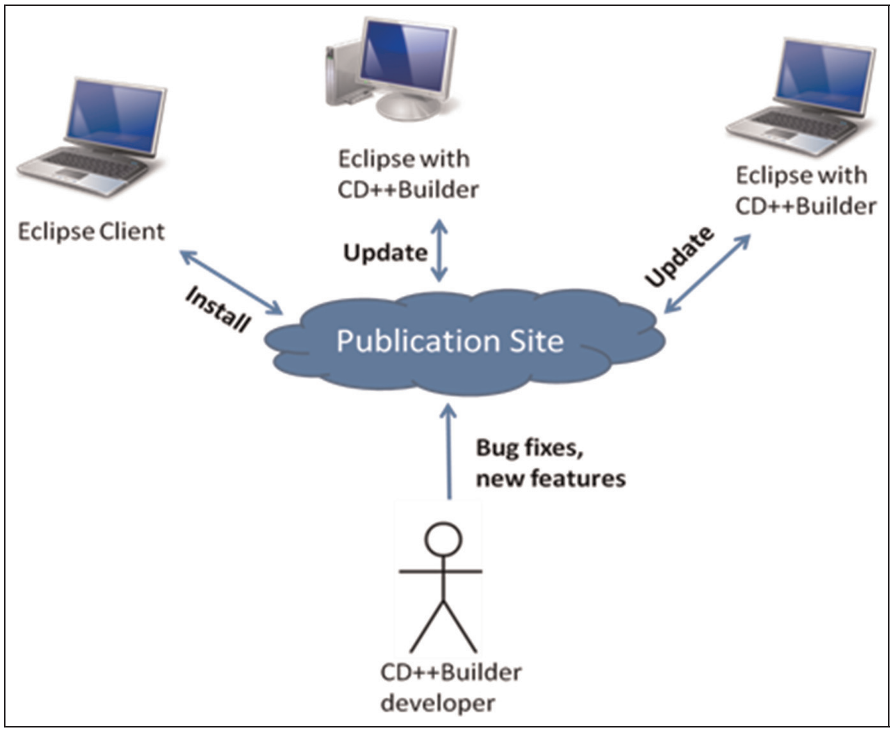


Figure 5. Centralized installation and update architecture.

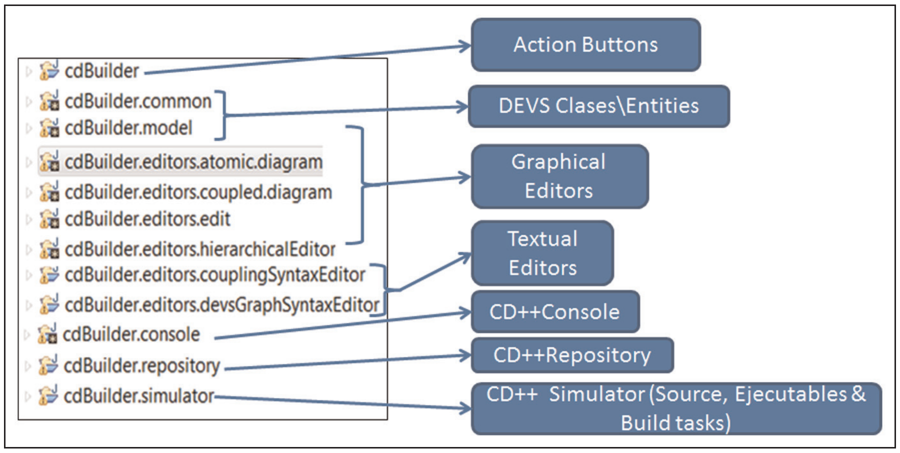

cdBuilder
cdBuilder.common
cdBuilder.model
cdBuilder.editors.atomic.diagram
cdBuilder.editors.coupled.diagram
cdBuilder.editors.edit
cdBuilder.editors.hierarchicalEditor
cdBuilder.editors.couplingSyntaxEditor
cdBuilder.editors.devsGraphSyntaxEditor
cdBuilder.console
cdBuilder.repository
cdBuilder.simulator
Action Buttons
DEVS Clases\Entities
Graphical Editors
Textual Editors
CD++Console
CD++Repository
CD++ Simulator (Source, Ejecutables & Build tasks)

Figure 6. CD++ Builder plug-in projects' structure.

code and its metadata in a single publication site, which all users access for installation and periodical checks for updates, as seen in Figure 5. This scheme allows easier wizard-guided installation into already running instances of Eclipse. More importantly, it resolves versioning problems; software fixes and new features do not have to be distributed to users individually, but uploaded into a central point. Integration with the Eclipse Update Manager allows the clients to trigger manual update checks or to configure for scheduling automatic periodic updates.

In summary, CD++ Builder was built using an extensible architecture, based on well-known infrastructure and frameworks provided by Eclipse. They allow updating and extending CD++ Builder functionalities with simple, robust and well-documented procedures, an enabler for the adoption of the tool by research and engineering communities demanding customized features. This represents a major advantage from other existing simulation tools. In addition, the integration with Eclipse Update Manager centralizes the software and updates distribution, reducing installation efforts and facilitating the delivery of new versions and fixes. Moreover, as the graphical model information is stored using standard persistence formats,[26] they can be easily updated and then model files could potentially be consumed by other tools, and even be used as a means for porting models to other simulators.

4. CD++ Builder implementation

As discussed earlier, a core requirement for CD++ Builder was to allow easy extensibility, as new features are continuously added by different teams in geographically distant places. The plug-in architecture in Section 3 enables this by allowing new decoupled features into CD++ Builder and integrating them seamlessly. In this section, we discuss the implementation of those features.

Figure 6 depicts the different Eclipse plug-in projects that implement the CD++ Builder environment. Each plug-in implements a specific component of the environment, simplifying individual updates, installation and project interdependencies.

- **cdBuilder** defines the Eclipse perspectives and action buttons.
- **cdBuilder.common** defines functionality common to all projects (such as helpers for manipulating Eclipse resources).
- **cdBuilder.model** contains the EMF description files and the Java classes generated in this project, which are used by the graphical editors (and possibly by other plug-ins to represent DEVS models).
- **cdBuilder.editors.atomic.diagram, *.diagram** and ***.hierarchicalEditor** contain GMF-generated classes and custom extensions to implement views and controllers for DEVS-Graphs and coupled models.
- **cdBuilder.editors.edit** contains the GEF-generated commands used by the graphical editors to interact with the EMF model entities.
- **cdBuilder.editors.couplingSyntaxEditor** and ***.devsGraphsSyntaxEditor** implement the classes for the atomic DEVS-Graphs and DEVS-coupled model textual editors.
- **cdBuilder.console** implements the classes and helper methods to interact with Eclipse's console, which is used to give user feedback (i.e. when users are compiling and simulating models).
- **cdBuilder.repository** implements the CD++ repository, an Internet-searchable database of models.
- **cdBuilder.simulator** contains the CD++ simulator executable (represented by the Simulator and Libraries layers in Figure 3) and the source files to recompile new atomic models. It also provides helper methods that implement compilation and simulation execution.

As we can see, the model behavior and its graphical representation are clearly separated. Figures, sizes, layout, colors and graphic-specific information are stored separately from the model. While this is conceptually correct, it presents some implementation challenges. Previously existing applications, such as CD++ Modeler[10,19] used custom formats for model graphics, from which textual definition could be extracted (as in Figure 1). Nevertheless, the opposite operation (i.e. generating a graphical representation from a textual model) was not possible. The vast legacy models available in the CD++ repository (http://cell-devs.sce.carleton.ca/) could not be opened using the previously existing graphical tools. In addition, once the models were exported to the CD++ format, the graphical representation could not be easily updated without losing consistency. To overcome these limitations, we developed parser and writer classes to interpret different formats and to translate from CD++ to the graphical representation, and vice versa.

4.1 Main implementation classes and hierarchies

As explained on Section 3, the classes implementing the graphical editors are generated by the EMF, and they are described using EMF modeling tools. CD++ Builder uses

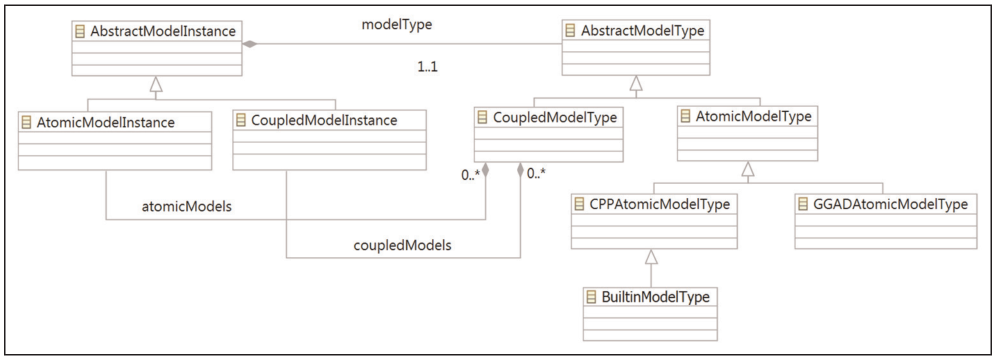


Figure 7. CD++ Builder model class hierarchy and relationships.

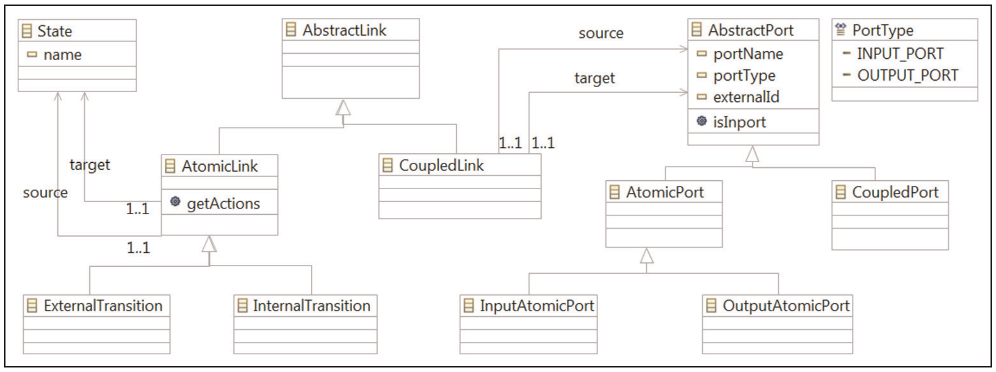


Figure 8. Class hierarchy and relationships for ports, links and transitions.

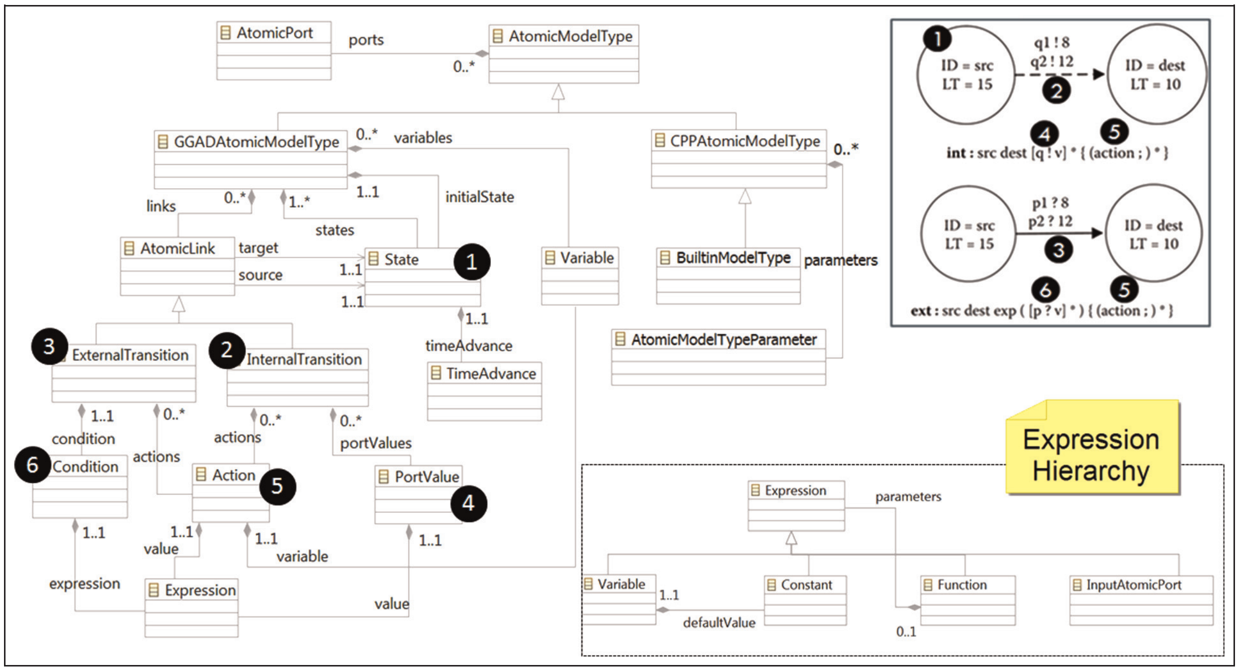


Figure 9. Atomic model class hierarchy mapping with DEVS-Graphs notation.

over 59 classes to describe each of the DEVS elements, implemented on the *cdBuilder.model* project. The most relevant classes, hierarchies, and relationships for this project are shown in Figure 7.

As we can see, atomic and coupled models are represented as both instances and types; in this way, coupled model types are composed of a set of model instances, allowing several instances of the same type (which was not possible in previous versions). The diagram also shows the different atomic model types available: DEVS-Graphs, represented by GGADAtomicModelType; and C++, represented by CPPAtomicModelType. BuiltinModelType represents special C++ atomic models already included in the CD++ simulator, which cannot be updated.

Figure 8 shows the classes for *links* and *ports*. Atomic links can represent either internal or external transitions for the DEVS-Graphs. Coupled links are connected to ports, allowing simple creation and graphical feedback validation (having links connected to components would make visual creation and validation harder).

Figure 9 shows the different entities used when defining atomic models with DEVS-Graphs or C++. DEVS-Graph editors use the Expression subclasses to verify the correctness of its actions and conditions. For C++ models we store the source file path, and we only represent its external interface (ports and parameters), identified by a parser that reads the source file to obtain this information. The DEVS-Graph class hierarchy represents the DEVS-Graphs notation (GGADAtomicModelType). The numbers in Figure 9 identify the mapping between the theoretical concepts shown in Figure 1 and the implementation; for example, the State class represents the Bubbles, and the InternalTransition class represents the dotted arrows. The output values, actions, and conditions for transitions (numbers 4, 5 and 6, respectively) are specified using expressions. A difference with previous editors is that the expressions used in DEVS-Graphs textual notation are modeled hierarchically, which allows easier definition and validation in graphical editors. Variables, constants, input ports and functions can form a valid expression, and complex expressions can be built by composing functions, as seen in Appendix II.

It is important to note that these figures represent executable models, serving as live documentation for developers. They are used by the EMF to generate the underlying classes, and no reference is made to the graphical implementation (which enables reuse of model classes by non-graphical plug-ins). The graphical editors' code is generated by the GMF, based on the EMF model and other files that describe base editors in the cdBuilder.model. Based on these, the GMF generates the projects cdBuilder.editors. atomic.-diagram, *.coupled.diagram and *.edit. A description of the code generated can be found in the GMF's documentation,[23] while some of the most important customizations performed to the code generated (in order to fulfill CD++ Builder requirements) are listed below.

- **Multiple page editors** to support the graphical rendering of a model in one page and the textual representation on another page (some distributions allow a third page with a tree-like hierarchical view).
- **Synchronization mechanisms** between pages to keep consistency between graphical and textual representations. Automatic and manual synchronization options are available.
- **Translators** for each of the persistence formats. Persistence of the graphical model is provided by the EMF and GMF, but custom translators and parsers were developed for CD++ coupled and DEVS-Graphs notations, for C++ atomic models and for CD++ Modeler formats, which are used when synchronizing models and animating results.
- **Hierarchical model navigation** by including commands that execute when figures are double clicked. For coupled models, an editor for the sub-model is opened in a new tab; for atomic models, the necessary files are created (using editors for DEVS-Graphs models or C++ files).
- **A tools panel with reusable models**, which dynamically loads built-in models and allows one to reuse them in a drag and drop fashion (creating a new instance of the associated model type).

4.2 Unit tests and automated updates

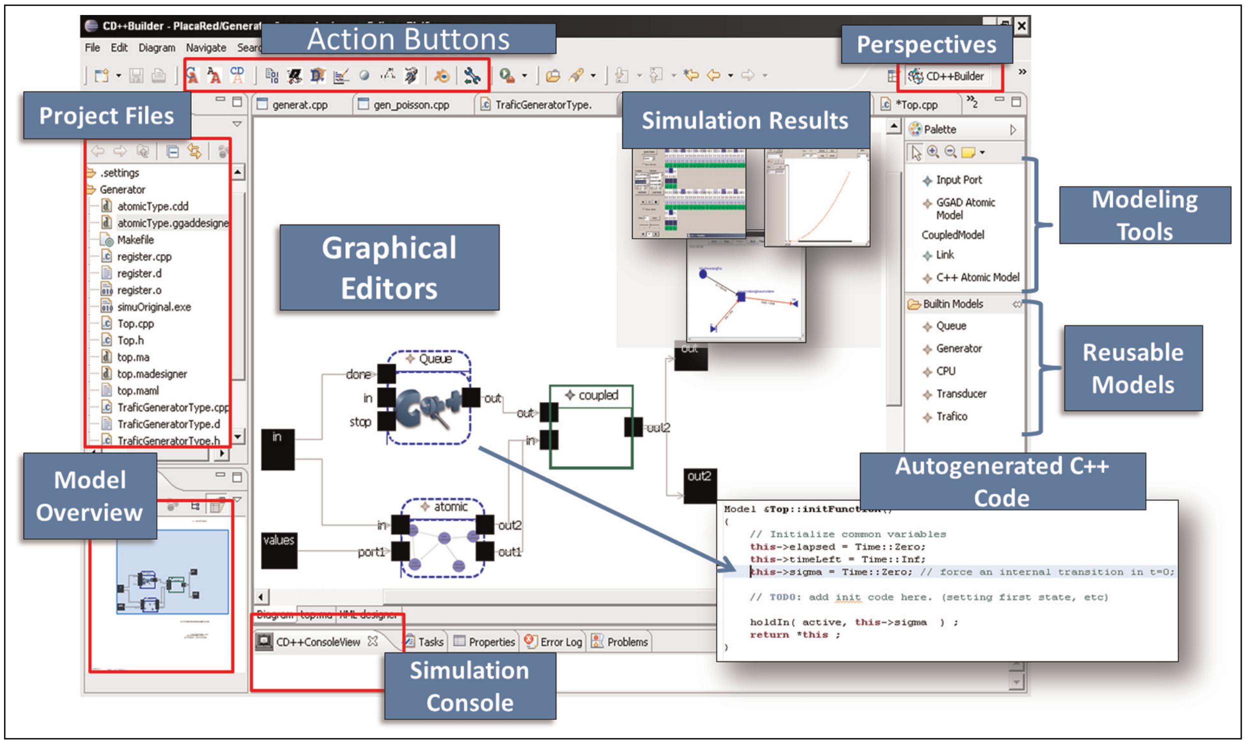
Each of the previously mentioned plug-ins is associated with a project that implements JUnit[27] testing for their main functionality. These tests were implemented by leveraging the Eclipse JUnit frameworks, which allow tests to run in a separate Eclipse testing environment instance. A testing project was created for each of the projects to verify the particular functionality of each component. In the case of generated projects, unit tests verify the correctness of custom-created classes. On the other hand, created integration tests validate the behavior of several features as a whole. Models defined in CD++ notation are interpreted and stored in XMI format, which in turn are used as inputs for the coupled and DEVS-Graphs editors to verify the correctness of the model. Graphical editors are used to translate models back to CD++ notation to verify that they are equivalent to their original definition. A similar process is used for coupled and atomic animations that require transformations from XMI format to CD++ Modeler format and vice versa.

The different plug-ins described above enable simple reuse and functionality updates. The separation of components into various plug-ins allows varied functionality to be reused by other tools, while enabling each plug-in to be updated independently (for example cdBuilder.simulator could be modified to support the Embedded CD++, while the graphical editors would not need to be updated). On the other hand, the EMF classes that describe DEVS models and their generation facilities could be easily modified using the EMF editors to support new scenarios (without changing the persistence mechanisms). The parsers that identify CD++ high-level languages and translate them into graphical formats could be modified to support new languages and other toolkits. One example of such extensibility is the inclusion of the CD++ repository,[28] an Internet-searchable database of CD++ models, which was developed by other authors in parallel and independently of CD++ Builder, and has been easily integrated as part of the software package.

We also defined two projects that leverage the installation and update features of new plug-ins. The cdBuilder. installation.features project defines an Eclipse installable feature and the cdBuilder.installation.updateSite project provides information to publish features and central in a unique site. For that purpose, a Sourceforge repository was created (http://cdppbuilder.sourceforge.net/updatesite/site.xml), providing the central access point for all users.

4.3 CD++ Builder user interface

Figure 10 shows the CD++ Builder user interface, in which the *Action buttons* in the top toolbar allow executing external tools (implemented in the cdBuilder plug-in). A special section is reserved animations and legacy tools (CD++ Modeler, GGADTool and Drawlog). The *Build* button uses the cdBuilder.simulator classes to generate a makefile and compile the source code of a given project, generating a new simulator that includes the new atomic models. The *Execute* button allows one to specify



```
Model &Top::initFunction()
{
    // Initialize common variables
    this->elapsed = Time::Zero;
    this->timeLeft = Time::Inf;
    this->sigma = Time::Zero; // force an internal transition in t=0;

    // TODO: add init code here. (setting first state, etc)

    holdIn( active, this->sigma  ) ;
    return *this ;
}
```

Figure 10. CD++ Builder environment.

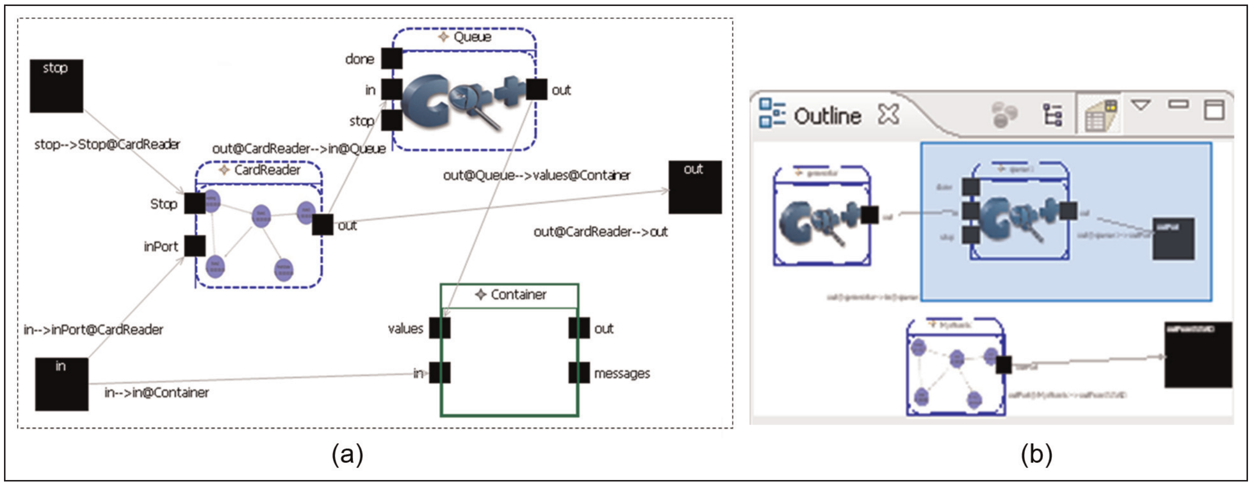
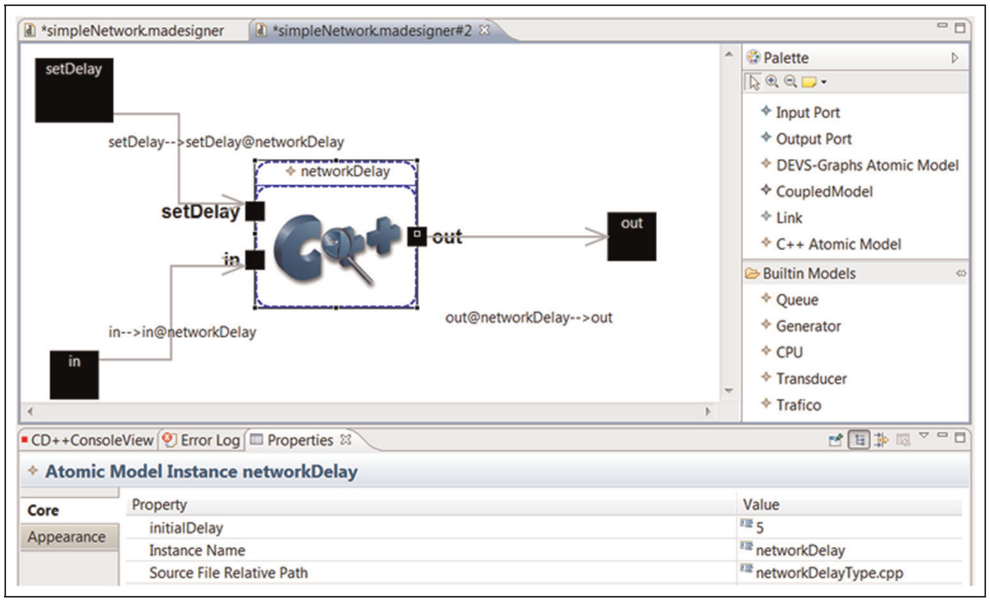


Figure 11. CD++ Builder coupled model editor and outline views.

parameters and start a simulation. Model editing is done within the Eclipse interface. Eclipse allows the users to rearrange windows to personalize the interface, and the *Perspectives* buttons and panels have been used to improve this organization for different scenarios.

Some of the figures to be discussed in the next paragraphs are included in Section 5, in which we will illustrate the use of the environment through a complete example. For instance, Figure 10 (and also Figure 15) shows that CD++ Builder is now integrated with Eclipse C/C++ Development Tools (CDT) to facilitate C++ coding. It also includes new animation features to allow visualization of simulation results. This graphical framework improves usability, including the usual editing actions (such as copy, paste, undo and redo). Other features include zoom in/out capabilities, flexible look and feel and different styles to avoid links obstructions and overlapping. Both coupled and atomic model editors



Property	Value
initialDelay	5
Instance Name	networkDelay
Source File Relative Path	networkDelayType.cpp

Figure 12. Coupled model tool pane with reusable models.

provide a special pane with tools for easily creating available entities (atomic/coupled models, links, ports, transitions, states, variables, etc.). The Eclipse Properties view (bottom of Figure 12) is used to show and edit entities, and the Outline (Figure 10 and Figure 11(b) shows an overview of the model.

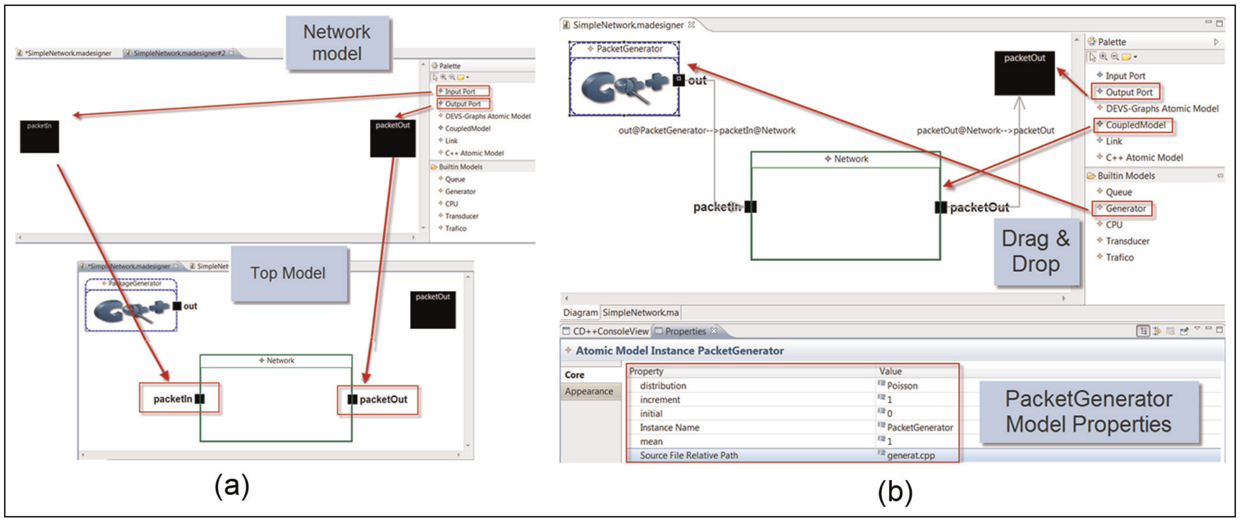
The coupled model editor shown in Figure 11 (and Figure 17) defines components using colored rectangles (with different ones for Coupled, DEVS-Graphs or Atomic C++ models). Ports are rendered as black boxes with their names and directed links. The DEVS-Graph atomic editor (Figure 14) uses DEVS-Graphs as in Figure 1, additionally rendering variables and allowing actions and conditions to be defined using the Eclipse Properties window.

To simplify the C++ atomic model definition, code generation capabilities were added to the coupled DEVS model editor. When a new C++ atomic model is selected, C++ files are generated based on a template that supplies the code structure to extend the *Atomic* class, including helpful comments and code samples (Figure 15). All methods that must be implemented (*initFunction*, *externalFunction*, *internalFunction* and *outputFunction*) are set up with brief comments useful for non-expert users. Comments are also used to show how to add input/output (I/O) ports and parameters.

When developing atomic models in C++, the model's graphical representation is kept consistent with its C++ underlying code, as described earlier. When C++ files are modified and saved, the parser recognizes the model's name, I/O ports and parameters. In this way, the model graphical representation and the code are always synchronized (with no restriction imposed on the code), enabling one to modify the graphical metaphor at any time.

DEVS-Graphs editors show both graphical and textual views (see Figure 14), and when any of the views is saved, both files are synchronized (using the translators to keep them consistent). Although the graphic files contain all the information needed to rewrite the CD++ model definition completely, the opposite is not true. Thus, when the textual file is saved, the diagram can be updated, but all its graphical information is lost. To overcome this issue, when the textual model definition is saved, the old diagram is synchronized using its graphical information (layout, figure sizes, colors, etc.) to supersede any missing information. Translators can also generate a new graphical representation from a textual model definition (even when it is written in an older version of CD++), enabling one to use models that were not originally built using graphical editors. In this case, default values are used for the missing graphical information. In addition, the new editors have been adapted to reuse the animation features of CD++Modeler. A control is provided to manage time advance, and links are dynamically highlighted to represent events between models (as in Figure 19). For coupled models, a block representation is used; for atomic models, the input and output trajectories are shown on different ports over time (Figure 20).

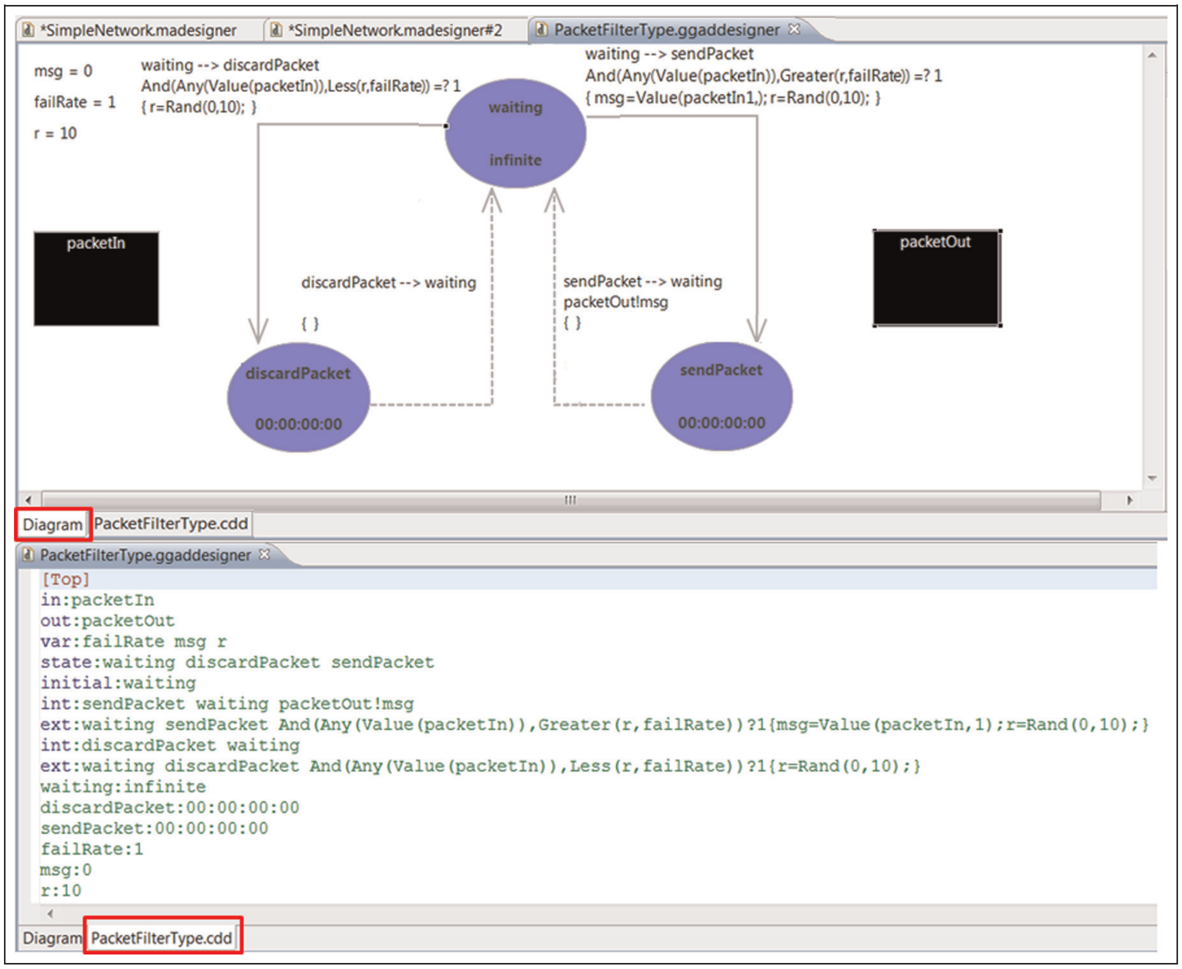
Having the ability to view all models graphically helps in better understanding other users' models, facilitating model reuse among the community. In this sense, we added a pane (seen on the right-hand side of Figure 12)



Atomic Model Instance PacketGenerator

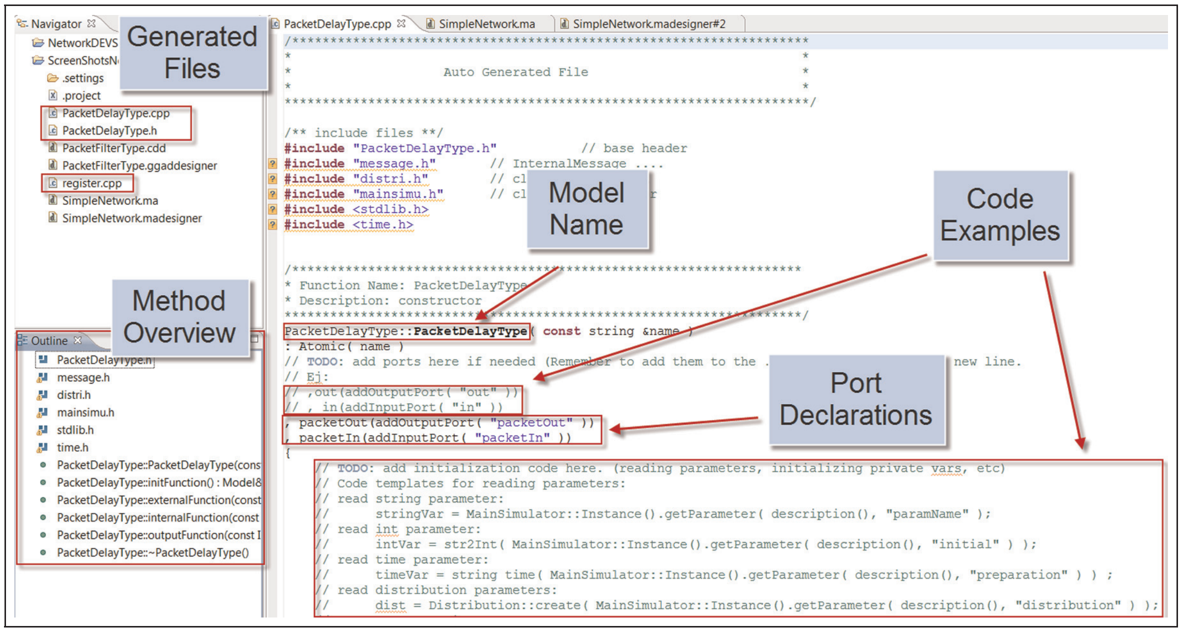
Property	Value
distribution	Poisson
increment	1
initial	0
Instance Name	PacketGenerator
mean	1
Source File Relative Path	generat.cpp

Figure 13. Port synchronization; Top and Network models creation; PacketGenerator properties.


*SimpleNetwork.madesigner
*SimpleNetwork.madesigner#2
PacketFilterType.ggaddesigner
msg = 0
failRate = 1
r = 10
waiting --> discardPacket
And(Any(Value(packetIn)),Less(r,failRate)) =? 1
{ r=Rand(0,10); }
waiting --> sendPacket
And(Any(Value(packetIn)),Greater(r,failRate)) =? 1
{ msg=Value(packetIn1,); r=Rand(0,10); }
waiting
infinite
packetIn
packetOut
discardPacket --> waiting
{ }
sendPacket --> waiting
packetOut!msg
{ }
discardPacket
00:00:00:00
sendPacket
00:00:00:00
Diagram
PacketFilterType.cdd
PacketFilterType.ggaddesigner

```
[Top]
in:packetIn
out:packetOut
var:failRate msg r
state:waiting discardPacket sendPacket
initial:waiting
int:sendPacket waiting packetOut!msg
ext:waiting sendPacket And(Any(Value(packetIn)),Greater(r,failRate))?1{msg=Value(packetIn,1);r=Rand(0,10);}
int:discardPacket waiting
ext:waiting discardPacket And(Any(Value(packetIn)),Less(r,failRate))?1{r=Rand(0,10);}
waiting:infinite
discardPacket:00:00:00:00
sendPacket:00:00:00:00
failRate:1
msg:0
r:10
```

Figure 14. PacketFilter DEVS-Graphs model definition (graphical and textual views).



```
/*********************************************************************
*                                                                   *
*                    Auto Generated File                            *
*                                                                   *
*********************************************************************/

/** include files **/
#include "PacketDelayType.h"           // base header
#include "message.h"        // InternalMessage ....
#include "distri.h"         // cl
#include "mainsimu.h"       // cl
#include <stdlib.h>
#include <time.h>


/*********************************************************************
* Function Name: PacketDelayType
* Description: constructor
*********************************************************************/
PacketDelayType::PacketDelayType( const string &name )
: Atomic( name )
// TODO: add ports here if needed (Remember to add them to the .            new line.
// Ej:
// ,out(addOutputPort( "out" ))
// , in(addInputPort( "in" ))
, packetOut(addOutputPort( "packetOut" ))
, packetIn(addInputPort( "packetIn" ))
{
    // TODO: add initialization code here. (reading parameters, initializing private vars, etc)
    // Code templates for reading parameters:
    // read string parameter:
    //      stringVar = MainSimulator::Instance().getParameter( description(), "paramName" );
    // read int parameter:
    //      intVar = str2Int( MainSimulator::Instance().getParameter( description(), "initial" ) );
    // read time parameter:
    //      timeVar = string time( MainSimulator::Instance().getParameter( description(), "preparation" ) ) ;
    // read distribution parameters:
    //      dist = Distribution::create( MainSimulator::Instance().getParameter( description(), "distribution" ) );
```

Figure 15. PacketDelay C++ model-generated file

with a section for reusable models. This lets non-expert users learn which models are available, and drag and drop them into the Editor pane to be composed within the model being edited.

5. Using CD++ Builder

In this section, we will show the development of a complete model representing a simple network that randomly loses packets and generates delays during packet transmission (which can be found in the CD++ repository). As shown in Figure 16, the top coupled model is composed of a packet generator and the actual network. We use the three different types of models available: DEVS-Graphs to represent network unreliability (only 90% of the packets received are retransmitted), the Generator built-in model to represent incoming packets (using probabilistic distributions built-in in CD++) and a C++ atomic model representing the packet delay behavior (the model resends packets after a random period between 0 and 1s).

CD++ Builder provides wizards to assist in the creation of the project. The Top model is automatically opened on the coupled model graphical editor (Figure 13(b)). We first add the PacketGenerator using the Built-in Generator model (drag and drop into the Editor's pane from the Built-in Models pane). Then, using the Model Properties window, we configure its parameters (bottom of Figure 13(b)). In this case, we choose packets to be generated using a Poisson distribution with average 1. We then create the Network coupled model, and its I/O ports. The Top model is completed by defining the PacketOut port at the Top model, and then connecting the internal ports (Figure 13(b)).

To specify the Network model of Figure 16, we open it in a different tab (top of Figure 14(a)) to define its contents. The two editors are synchronized, so adding new ports to the Network model will automatically make them available to the Top model editor. Then, PacketFilter and PacketDelay must be added. PacketFilter is a DEVS-Graph using three states: *waiting*, *discardPacket* and *sendPacket* (Figure 14). The model remains in *waiting* state, and when a new packet arrives at port *PacketIn*, it randomly

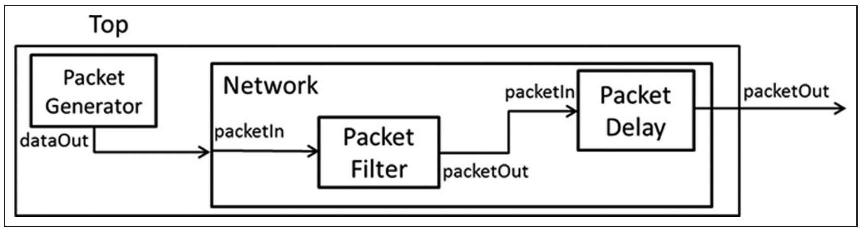
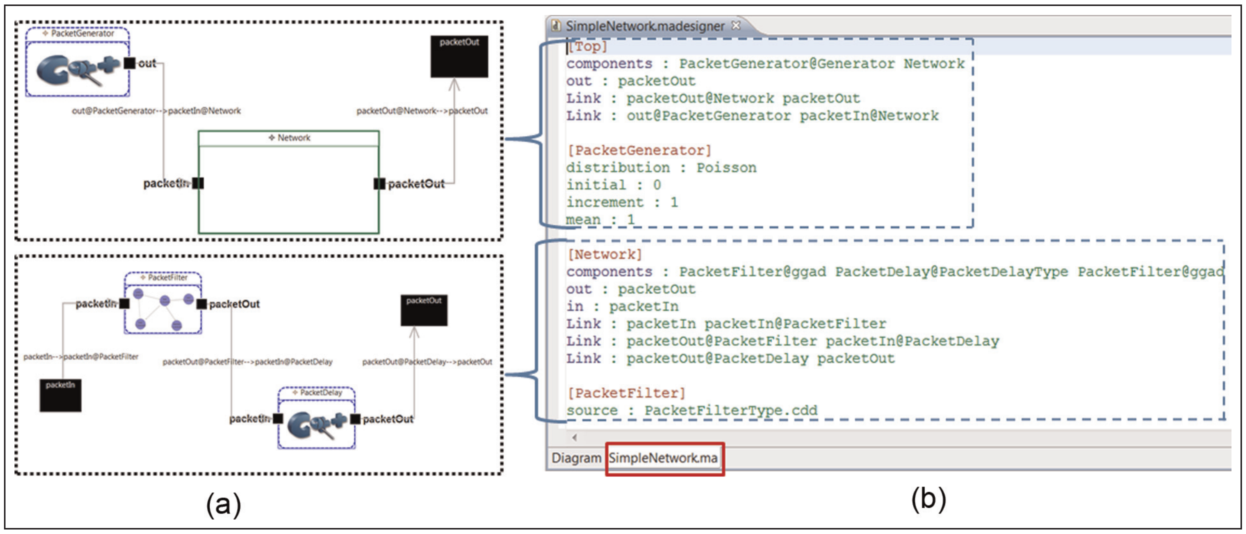


Figure 16. Network DEVS model.



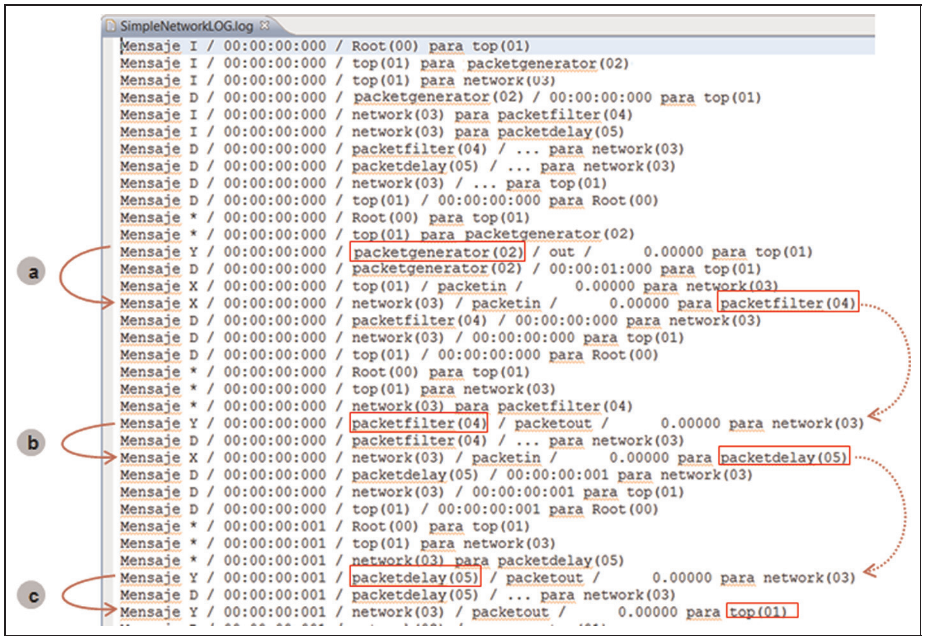
```
[Top]
components : PacketGenerator@Generator Network
out : packetOut
Link : packetOut@Network packetOut
Link : out@PacketGenerator packetIn@Network

[PacketGenerator]
distribution : Poisson
initial : 0
increment : 1
mean : 1

[Network]
components : PacketFilter@ggad PacketDelay@PacketDelayType PacketFilter@ggad
out : packetOut
in : packetIn
Link : packetIn packetIn@PacketFilter
Link : packetOut@PacketFilter packetIn@PacketDelay
Link : packetOut@PacketDelay packetOut

[PacketFilter]
source : PacketFilterType.cdd
```

Figure 17. Simple network model: Graphical representation and Textual definition.



```
Mensaje I / 00:00:00:000 / Root(00) para top(01)
Mensaje I / 00:00:00:000 / top(01) para packetgenerator(02)
Mensaje I / 00:00:00:000 / top(01) para network(03)
Mensaje D / 00:00:00:000 / packetgenerator(02) / 00:00:00:000 para top(01)
Mensaje I / 00:00:00:000 / network(03) para packetfilter(04)
Mensaje I / 00:00:00:000 / network(03) para packetdelay(05)
Mensaje D / 00:00:00:000 / packetfilter(04) / ... para network(03)
Mensaje D / 00:00:00:000 / packetdelay(05) / ... para network(03)
Mensaje D / 00:00:00:000 / network(03) / ... para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:000 para Root(00)
Mensaje * / 00:00:00:000 / Root(00) para top(01)
Mensaje * / 00:00:00:000 / top(01) para packetgenerator(02)
Mensaje Y / 00:00:00:000 / packetgenerator(02) / out /       0.00000 para top(01)
Mensaje D / 00:00:00:000 / packetgenerator(02) / 00:00:01:000 para top(01)
Mensaje X / 00:00:00:000 / top(01) / packetin /       0.00000 para network(03)
Mensaje X / 00:00:00:000 / network(03) / packetin /       0.00000 para packetfilter(04)
Mensaje D / 00:00:00:000 / packetfilter(04) / 00:00:00:000 para network(03)
Mensaje D / 00:00:00:000 / network(03) / 00:00:00:000 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:000 para Root(00)
Mensaje * / 00:00:00:000 / Root(00) para top(01)
Mensaje * / 00:00:00:000 / top(01) para network(03)
Mensaje * / 00:00:00:000 / network(03) para packetfilter(04)
Mensaje Y / 00:00:00:000 / packetfilter(04) / packetout /       0.00000 para network(03)
Mensaje D / 00:00:00:000 / packetfilter(04) / ... para network(03)
Mensaje X / 00:00:00:000 / network(03) / packetin /       0.00000 para packetdelay(05)
Mensaje D / 00:00:00:000 / packetdelay(05) / 00:00:00:001 para network(03)
Mensaje D / 00:00:00:000 / network(03) / 00:00:00:001 para top(01)
Mensaje D / 00:00:00:000 / top(01) / 00:00:00:001 para Root(00)
Mensaje * / 00:00:00:001 / Root(00) para top(01)
Mensaje * / 00:00:00:001 / top(01) para network(03)
Mensaje * / 00:00:00:001 / network(03) para packetdelay(05)
Mensaje Y / 00:00:00:001 / packetdelay(05) / packetout /       0.00000 para network(03)
Mensaje D / 00:00:00:001 / packetdelay(05) / ... para network(03)
Mensaje Y / 00:00:00:001 / network(03) / packetout /       0.00000 para top(01)
```

Figure 18. Simulation Results. Log file: synchronization and input/output messages.

transitions to the states *sendPacket* or *discardPacket*, which represent packet transmission and packet loss, respectively. The model also uses three state variables: *failRate*, *r* and *msg*. The probability to transition to one or other state is given by the state variables *failRate* and *r*. In this example, the *failRate* state variable is fixed at 1 (which represents a 10% chance of discarding packets), but it could be updated dynamically. Figure 14 shows the representation of the PacketFilter model in the CD++ Builder DEVS-Graphs editor, which allows one to view the graphical representation (at the top of the figure) and the CD++ grammar textual representation (at the bottom of the figure). Appendix II shows the complete specification of CD++ DEVS-Graphs grammar.

The values of the state variables *msg* and *r* are calculated in the actions associated with the external transitions, and they are updated every time a new packet arrives: *msg* stores the value of the messages arrived and *r* is calculated at random. The random transition to the states *sendPacket* or *discardPacket* is modeled using two different external transitions that compare the *failRate* and *r* variables to calculate the next state. The condition associated to the first external transition "*And(Any(Value(packetIn)),Greater(r,failRate))?1*" specifies that the transition should be executed if there is

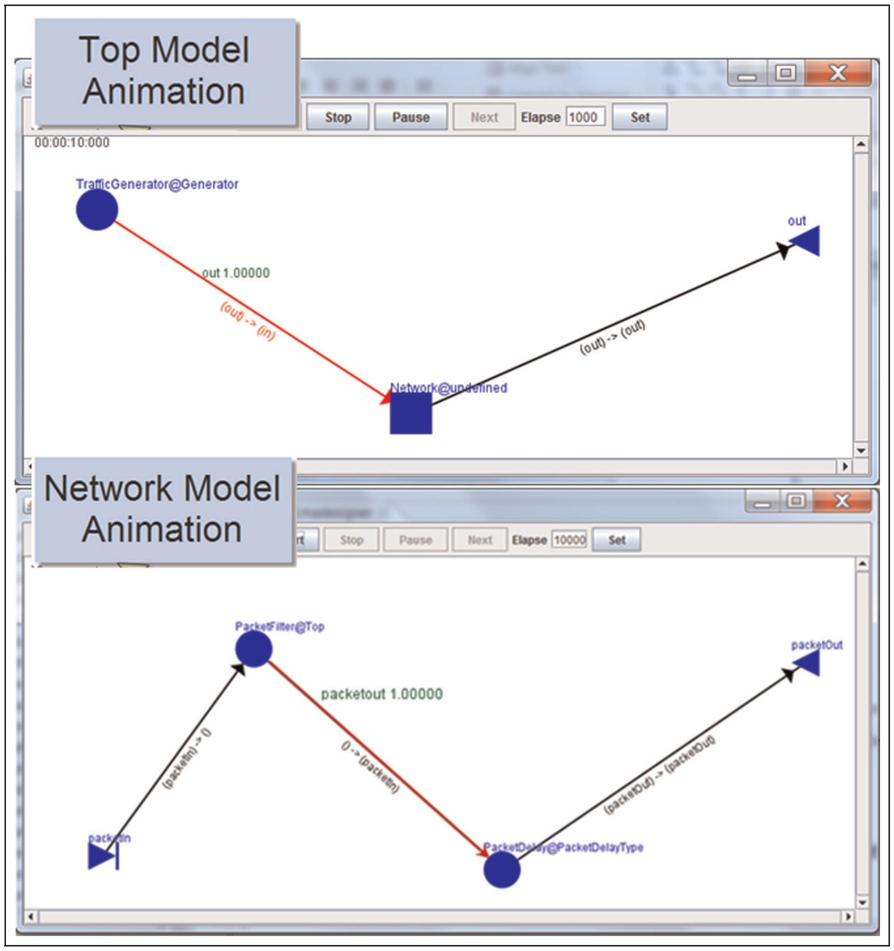


Figure 19. Simulation Results Animation Dynamic information shown on the model structure

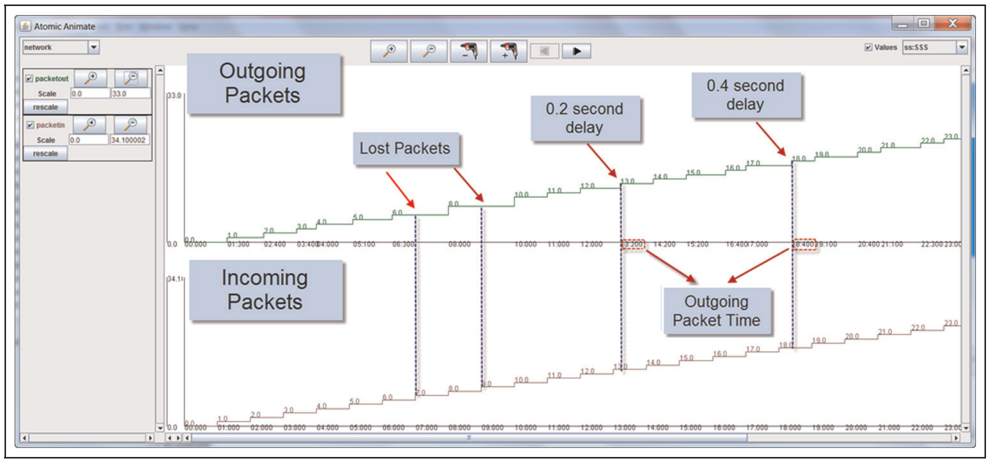


Figure 20. Simulation Results Animation Timeline of output values.

value in the *packetIn* port ("*Any(Value(packetIn)*") and the value of variable *r* is greater than the value of variable *failRate* ("*Greater(r,failRate)*"). When this condition is met, the actions "*msg=Value(packetIn,1)*" and "*r=Rand(0,10)*" are executed, which specify the new values for the *msg* and *r* variables. The condition associated to the second external transition "*And(Any(Value(packetIn)), Less(r,failRate))?1*" is similar to the previous one, but is executed when the value

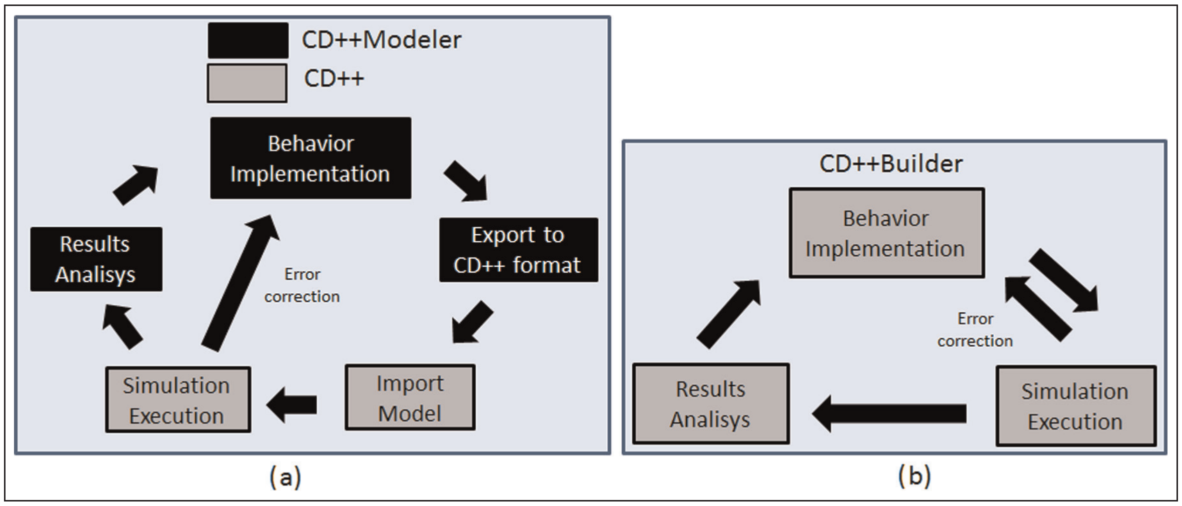


Figure 21. Integration of tools supporting the modeling and simulation process into a single environment.

of variable *r* is smaller than the value of variable *failRate*. In the case the condition of the second external transition is met, the same actions are executed, but the value of the *msg* variable does not need to be stored.

When the model changes to *sendPacket*, the value of the arrived packet is stored in the *msg* variable, so that it can be retransmitted later if needed (this is represented by the expression *msg=Value(packetIn,1)*). When the model is in states *sendPacket* or *discardPacket*, an instantaneous internal transition is executed (as the *ta* value for both these states is 0) and the model changes to the *waiting* state. When transitioning from *discardPacket* to *waiting*, no value is sent through the output port (a packet is lost). When transitioning from *sendPacket* to *waiting*, the value of the *msg* variable is sent through the *PacketOut* output port, representing a packet transmitted.

As seen in Figure 15, the PacketDelay atomic model is programmed in C++. This model uses one input port for receiving packets, and one output port for sending packets after a random period. It includes a queue to store messages received in the external transition function. The editor recognizes the ports and parameter declaration in C++ code, and it keeps consistency between the model definition and its graphical representation. Thus, Network shows the PacketDelay atomic model with its corresponding input and output ports, which can be used to create the necessary links.

Figure 17 shows the complete coupled model definition (Figure 17(a)) and the CD++ textual declaration (Figure 17(b)), which are kept consistent.

The simulation generates a log file (shown in Figure 18), which is used by the animation tool (shown in Figure 19). The log file includes synchronization and I/O messages sent between models. As seen in Figure 20 at the left the simulation starts with *initialization* (*I*) messages and their corresponding *done* (*D*) messages (which indicate the next scheduled event for each model), after which the imminent models are identified and activated. The *imminent* message * tells the models to execute their output function (which can generate an output messages *Y*) and the internal transition function after. In this case, we can see that PacketGenerator sent a *Y* message with value 0.00000 through the out port in the top coupled model. After executing its internal transition function, it announced its next scheduled event (a *D* message 1 second ahead in time). Using the coupled model information, the *Y* message is converted into an internal message *X* (using the Z_{ij} function), and it is sent to the PacketIn port of the Network coupled model. In turn, the coupled model is queried to find out that the *X* message must be propagated to the PacketIn port of the PacketFilter atomic model. This cycle is completed with three *D* messages notifying the next scheduled time for each model.

The next simulation cycle activates the imminent model by means of the propagation of a * message in the model hierarchy. In this case, the imminent PacketFilter atomic model output function generates an output through its PacketOut port to the parent coupled model Network, and the internal transition function is executed. The time advance function generates a new D message indicating that it has no further scheduled events ('…' indicates infinity). The value of the last Y message is converted into an internal message X sent to the PacketIn port of PacketDelay. This simulation cycle ends with three D messages, making the PacketDelay model the new imminent atomic model with a future event to be processed in 1 millisecond.

A new simulation cycle starts with the PacketDelay atomic model generating a Y message through PacketOut to its parent, Network, and then passivating (a D message with time infinity). This time, a new Y message is immediately sent upwards form Network to its parent Top coupled model. On the left-hand margin of Figure 18 we can follow the thread of I/O messages, which trace the lifetime of a network packet from its generation until it has passed through the network. Trace (a) shows Packet ID 0.00000 emitted as a Y message from PacketGenerator and received as an X message by PacketFilter. Trace (b) shows how the packet is forwarded from PacketFilter to PacketDelay through the PacketOut and PacketIn ports, respectively. Finally, Trace (c) shows how the packet leaves PacketDelay and the Network subsystem.

While the log file includes all the required information to reproduce a simulation experiment, it becomes hard to follow complex threads of I/O messages. CD++ Builder was thus provided with animation tools to graphically sequence I/O messages as time progresses, making it easier to study the simulation results. For instance, in our previous example, we can see the delay generated for the packet transmission and the lost packets. This can be seen in Figure 19, which shows the Coupled Animation tool visualizing the dynamics of the Top coupled model and the Network coupled model. The structure of the model is represented graphically (circles are atomic models, squares are coupled models, and triangles are I/O ports). The arrows bear information about port linkage, and they show the message and port values dynamically (i.e. the ‘active’ link is highlighted, and extra information is provided for that instant).

The output file can also be visualized using the Atomic Animation tool, which provides a multi-chart timeline for the message values flowing through input and output ports. For example, Figure 20 shows an animation for the Network model, where incoming and outgoing packets can be compared as time advances, and therefore lost packets and delay times become easy to visualize. The topmost chart uses a timeline to show the PacketOut port values, while the lower chart shows the PacketIn port values. When a packet is lost, the PacketOut port remains unchanged, as shown for the packets that arrived at times 7 and 9. The randomly generated packet delay can be seen by comparing the packet arrival times (lower chart) and packet retransmissions (top chart).

6. Discussion and conclusion

We presented the architecture and features of CD++ Builder, an open-source Eclipse plug-in intended to facilitate the process of DEVS M&S with CD++. In this section, we give a brief discussion about the results of the research results, and how the goals were achieved. We then give a conclusion and discuss future related work in this area.

6.1 Discussion

Comparing the M&S workflow using previously existing tools for CD++ and CD++ Builder, we can see that the new IDE eliminates error-prone manual steps, simplifying the process and reducing learning curve for new users. All the steps originally shown in Figure 2 had to be executed manually. A first evolution of that process allowed one to define the models graphically with CD++ Modeler, in which the behavioral definition of the models using DEVS-Graphs did not need to be compiled, as seen in the workflow depicted in Figure 21(a). Nevertheless, additional steps (e.g. exporting/importing the graphical models between different tools) were needed. CD++ Builder integrates the graphical modeling capabilities within Eclipse, simplifies the graphical definition of atomic and coupled models and keeps consistent versions of the declarative and the graphical representation of models. Figure 21(b) shows that, by using CD++ Builder, the workflow for M&S with DEVS-Graphs is now even simpler, as the IDE facilitates all the steps, eliminating, for instance, the export/import phase.

Besides reducing overhead in the graphical modeling process, CD++ Builder introduces tools to support developing atomic models in C++ (detailed in the case study

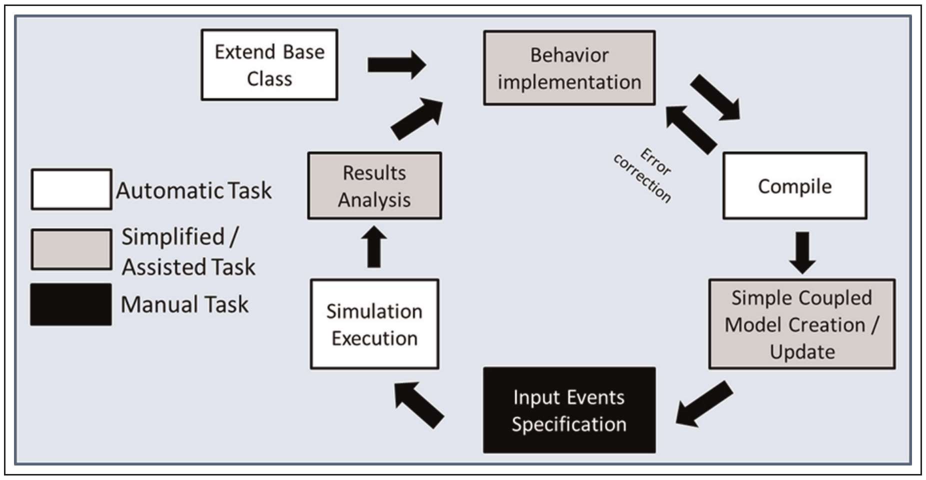


Figure 22. Process for the creation of C++ atomic models assisted by CD++ Builder.

in Section 5). Figure 22 presents a graphical summary of the unified workflow obtained in the present work, emphasizing the evolution of the M&S process from its previous status (Figure 2, Section 2) to the current version.

As we can see, three tasks have been simplified and another three have been automated. When defining new atomic models, the underlying corresponding C++ class is automatically generated by CD++ Builder, both for DEVS-Graphs and for models programmed in C++. In the latter case, the implementation of the atomic model is assisted by code templates, code highlighting and other programming aids (functions outline, intellisense, etc.). When an atomic model is defined using C++, a compilation task is required (the build action is also automated). Building and updating coupled models is greatly simplified, and it can be done using the coupled model graphical editor, which is completely integrated into the CD++ Builder environment (there is no need to export/import from/to CD++ Modeler anymore). Executing the simulation is automatically done by defining the simulation parameters (stop time, input events file, etc.) in a simple manner, and launching the simulation execution. Finally, the analysis of simulation results was previously done by interpreting the textual log file or by running different independent tools with no contextual relation to the model. This task has been simplified and the graphical animation tool can be used for visualizing atomic and coupled simulation results integrated within the CD++ Builder IDE, offering visualization alternatives contextualized with the model under development. (Teaching the use of DEVS simulation to new students using CD++ has been reduced from 9 hours to 3 hours. The time needed for a first complete model to be developed by those students has been reduced from 1 month to 2 weeks, thanks to the use of CD++ Builder.)

6.2 Conclusion and future work

The development of CD++ Builder has provided us with an IDE that:

- includes all M&S tasks (modeling, compiling, simulation execution and analysis);
- supplies editors that support the complete modeling cycle to be performed graphically;
- includes C++ code templates to aid in the implementation of atomic models, while keeping the graphical representation of these models consistent with their C++ code;
- supports extensibility and development of new features into the environment, including automated regression testing capabilities.

Having all the features integrated into Eclipse allows for easy extensibility by adding new plug-ins. While most tools are built from the ground up, CD++ Builder was built upon Eclipse's extensible mechanisms, which allows adding new plug-ins independently from the rest of the features. Well-known frameworks have been used to develop graphical editors, guaranteeing easy-to-use interfaces and simplifying extensibility scenarios, as discussed on Section 5.

DEVS-Graphs and C++ atomic models can now be used to define atomic model behavior. Other existing simulators do not support atomic models to be defined graphically, and they require this behavior to be specified in some programming language. CD++ Builder supports complete graphical modeling (as shown on Section 5), combining CD++ coupled models and DEVS-Graphs, allowing non-expert developers to model real-world systems utilizing the DEVS formalism. In addition, the CDT Eclipse plug-in is used for developing more advanced models using C++ (whenever this is needed), allowing a combination of graphical and non-graphical models, and making the coding of C++ models easier. We also showed how the modeling and definition of new C++ atomic models is simplified by auto-generated code templates, which are kept synchronized with their graphical representation; other tools restrict the portions of code that can be edited, or their model interfaces are not represented graphically. Likewise, descriptive model languages, not supported by most other simulators, can be used to port models among simulators. Eclipse's IDE extensibility features also allowed the integration of several previously existing tools into the same environment. For instance, the modeler's programming experience was enhanced by integrating the Eclipse CDT plug-in to facilitate C++ coding, thus reducing the learning curve for new users.

We improved issues about usability and modeling limitations using new editors, a tool for easier model reuse, a coupled model editor with automatic discovery, and new install and update mechanisms. An example of this is the Virtual Laboratory of Model-Based Development for Network Processors (NPs), and Embedded-CD++, both currently under construction by our group. The IDEs for both tools are based on CD++ Builder, and they are targeted to design advanced embedded control algorithms, including a specialized version for the Intel IXP family of NPs[29] and a different one for embedded controllers on microcontrollers and e-puck robotic kits.[30,31] CD++ Builder provides a transparent interface for dealing with the intricacies of the target hardware, such as compiling, downloading and monitoring models for their real-time execution on the target platform. It also provides an integrated environment for mixing DEVS models with low-level hardware-specific drivers, making the simulator interact with real network signals in a real time.

In the future, we will also include the synchronization of the right tool pane with the online CD++ repository, to extend the set of models to be reused and facilitate searching and uploading models. Likewise, we will explore the integration with different versions of the simulator using Web Services[32] or the RISE middleware,[33] which would allow the users to use a lightweight client on their computers and run advanced simulations on high-performance servers remotely.

Funding

This research has been partially funded by NSERC and GRAND NCE.

References

1. Zeigler B, Praehofer H and Kim T. *Theory of modeling and simulation*. 2nd ed. Academic Press, 2000; Orlando, Florida, USA.
2. Wainer G. *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2009: Boca Raton, Florida, USA.
3. Sarjoughian H and Zeigler B. DEVSJAVA: basis for a DEVS-based collaborative M&S environment. In: *proceedings of the international conference on web-based modeling & simulation*, San Diego, CA, 1998.
4. Wainer G. Applying Cell-DEVS methodology for modeling the environment. *Simulation* 2006; 82: 635–660.
5. Chen Y and Sarjoughian H. A Component-based simulator for MIPS32 processors. *Simulation* 2010; 86: 271–290.
6. Pérez E, Ntaimo L, Bailey B, et al. Modeling and simulation of nuclear medicine patient service management in DEVS. *Simulation* 2010; 86: 481–501.
7. Kim T, Seo C and Zeigler B. Web-based distributed network analyzer using a system entity structure over a service-oriented architecture. *Simulation* 2010; 86: 155–180.
8. Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87: 113–132.
9. Traoré M. SimStudio: a next generation modeling and simulation framework. In: *proceedings of SIMUTools*, Marseille, France, 2008.
10. Chidisiuc C and Wainer G. CD++ Builder: an eclipse-based IDE for DEVS modeling. In: *proceedings of SpringSim (DEVS symposium)*, Norfolk, VA, 2007.
11. Song HS and Kim TG. The DEVS framework for discrete event systems control. In: *5th annual conference on AI, simulation and planning in high autonomous systems*, Gainesville, FL, 1994.
12. Budinsky F, Steinberg D, Merks E, et al. *Eclipse modeling framework*. Addison-Wesley Professional, 2005: Boston, MA, USA.
13. Nutaro J. ADEVS: A Discrete Event System simulator, http://www.ornl.gov/~1qn/adevs/adevs-docs/manual.pdf (accessed 12 August 2011).
14. Wainer G, Glinsky E and Gutierrez-Alcaraz M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simulation* 2011; 87: 555–580.
15. Kim TG, Sung CH, Hong S-Y, et al. DEVSim++ toolset for defense modeling and simulation and interoperation. *J Defense Model Simulat* 2011; 8: 129–142.
16. Himmelspach J and Uhrmacher A. The JAMES II framework for modeling and simulation. In: *2009 international workshop on high performance computational systems biology*, Trento, Italy, 2009.
17. Saurabh M, Risco-Martín J and Zeigler B. DEVS/SOA: a cross-platform framework for net-centric modeling and simulation in DEVS unified process. *Simulation* 2009; 85: 419–450.
18. Wainer G. CD++: a toolkit to define discrete event models. *Software Pract Ex* 2002; 32: 1261–1306.
19. Christen G, Dobniewski A and Wainer G. Modeling state-based DEVS models in CD++. In: *proceedings of MGA, advanced simulation technologies conference 2004 (ASTC'04)*, Arlington, VA, 2004.
20. Wainer G. Developing a software tool for urban traffic modeling. *Software Pract Ex* 2006; 37: 1377–1404.
21. D'Abreu M and Wainer G. M/CD++: modeling continuous systems using Modelica and DEVS. In: *proceedings of MASCOTS 2005*, Atlanta, GA, 2005.
22. Eclipse Consortium. Eclipse Graphical Editing Framework (GEF) – Version 3.4, http://www.eclipse.org/gef (accessed 14 September 2010).
23. Eclipse Consortium. Eclipse Graphical Modeling Framework (GMF), http://www.eclipse.org/gmf (accessed 14 September 2010).
24. Shatalin A and Tikhomirov A. Graphical modeling framework architecture overview. In: *Eclipse modeling symposium*, 2006,
25. Ehrig K, Ermel C, Hansgen S, et al. Generation of visual editors as eclipse plugins. In: *20th IEEE/ACM international conference on automated software engineering*, Long Beach, CA, 2005.
26. OMG/XMI. October 1998, XML Model Interchange (XMI) OMG Document AD/98-10-05.
27. Massol V and Husted T. *JUnit in action*. Manning Publications, 2003: Samford, NY, USA.
28. Chreyh R and Wainer G. CD++ repository: an internet based searchable database of DEVS models and their experimental frames. In: *proceedings of SpringSim'09*, San Diego, CA, 2009.
29. Castro R, Kofman E and Wainer G. A DEVS-based end-to-end methodology for hybrid control of embedded networking systems. In: *3rd IFAC conference on analysis and design of hybrid systems*, Zaragoza, Spain, 2009.
30. Saadawi H and Wainer G. Principles of DEVS models verification. *Simulation*. [Epub ahead of print] 2011. DOI: 10.1177/0037549711424424. Accessed: October 23, 2011.
31. Moallemi M, Wainer G, Bergero F, et al. Component-oriented interoperation of RealTime DEVS engines. In: *proceedings of annual simulation symposium*, Boston, MA, 2011.
32. Wainer G, Al-Zoubi K and Madhoun R. Distributed simulation of DEVS and Cell-DEVS models in CD++ using web-services. *Simulat Model Pract Theory* 2008; 16: 1266–1292.
33. Wainer G and Al-Zoubi K. Distributed simulation using RESTful interoperability simulation environment (RISE) middleware. In: Tolk A (ed.) *Handbook on intelligence-based systems engineering*. Springer, 2011: Berlin, Germany.

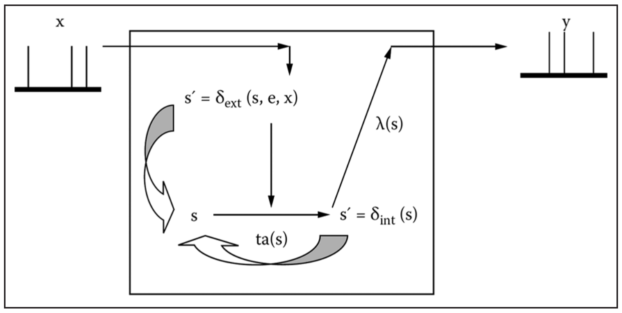


Figure 23. DEVS atomic model.[2]

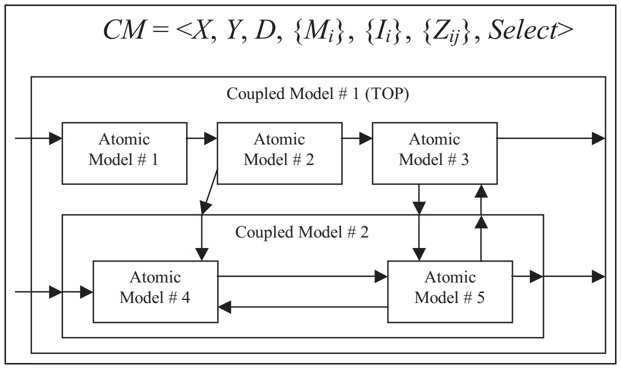


Figure 24. DEVS-coupled model structure.

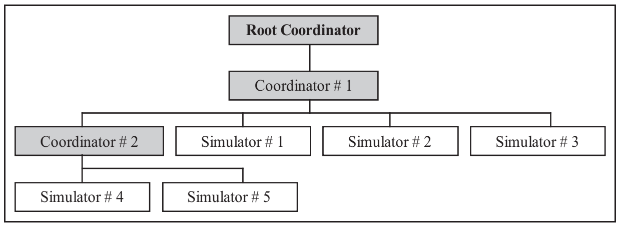


Figure 25. Simulation hierarchy for the coupled model in Figure 24.

Author biographies

Gabriel A Wainer (SMSCS, SMIEEE) received his MSc (1993) and PhD degrees (1998, with highest honors) from the University of Buenos Aires (UBA), Argentina, and Université d'Aix-Marseille III, France. After being Assistant Professor at the Computer Science Department of the UBA, in July 2000 he joined the Department of Systems and Computer Engineering at Carleton University, where he is now an Associate Professor. He has been a visiting scholar at ACIMS (The University of Arizona); LSIS/CNRS, University of Nice and INRIA (Sophia-Antipolis), France. He is the author of three books and over 250 research papers; he has edited nine other books and helped organize over 110 conferences, including being one of the founders of SIMUTools and SimAUD. He is the Vice-President of Publications, and was a member of the Board of Directors, of the SCS. He is Special Issues Editor of *Simulation*, member of the Editorial Board of *Wireless Networks* (Elsevier), *Journal of Defense Modeling and Simulation*, and the *International Journal of Simulation and Process Modelling* (Inderscience). He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He has been the recipient of various awards, including the IBM Eclipse Innovation Award, SCS Leadership Award and various Best Paper awards. He has been awarded Carleton University's Research Achievement Award (2005–2006), the First Bernard P. Zeigler DEVS Modeling and Simulation Award and the SCS Outstanding Professional Award (2011). Further information can be found at http://www.sce.carleton.ca/faculty/wainer.

Matias Bonaventura received his MSc (2010) from the UBA, Argentina. From 2009 to 2011 he was ad honorem lecturer at the Department of Computer Science, School of Sciences, UBA.

Rodrigo Castro, MSCS, MIEEE, received his MASc in Electrical Engineering (2004) and his PhD in Electrical Engineering (2010) from the Universidad Nacional de Rosario, Argentina. Since 2007 he has been an adjunct Lecturer at the Department of Computer Science, School of Sciences, Universidad de Buenos Aires, Argentina (UBA), and a Lecturer for the graduate programs at the School of Engineering at UBA. In 2011, he was appointed as Assistant Researcher at the National Scientific and Technical Research Council (CONICET), Argentina. Since 2000, he has participated in several projects for Siemens, Cisco and Hewlett Packard, in the area of networking, performance analysis and software development. In 2007 he was awarded a PhD Fellowship from Fundación Repsol YPF and a grant from the Organization of Ibero-American States for studies in Cooperation University-Enterprise for Development (Spain). In 2009 he received an Emerging Leaders in the Americas Program scholarship from the Government of Canada. He was a visiting research scholar at Carleton University (Ottawa, Canada) and the ETH Zurich (Zurich, Switzerland). His current research interests are related with modeling formalisms and methodologies, real-time simulation and energy-based specification world-scale systems.

Appendix I: the DEVS formalism

Formally, a DEVS atomic model (shown in Figure 23) is defined with the following structure:

$$M = < X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta >$$

A DEVS atomic model in a state $s \in S$ remains in that state for the time indicated by the time advance function $ta(s)$. Once the time $ta(s)$ expires, the model can generate an output event evaluating the $\lambda(s)$ output function and, then, it transitions to state s' indicated by the $\delta_{int}(s)$ internal transition function. If an external event $x \in X$ occurs, the model transitions to state s' indicated by the external transition function $\delta_{ext}(s, e, x)$, which also takes the state s

in which the model was when the external event occurred, and e, the time elapsed since the last transition.

A DEVS-coupled model (shown in Figure 24) is composed of several atomic or coupled submodels, creating a hierarchical structure and it is formally defined by:

$$CM = <X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, Select>$$

In this case, X and Y are the set of external event values and output values for the coupled model. Coupled models are composed of a set of submodels M_i ($i \in D$) that interact with each other through their I/O interfaces X_i and Y_i. I_i defines the set of models influenced by each M_i. The translation function Z_{ij} converts the output of each model into input for other models ($X_i \rightarrow Y_j$). The Select tie-breaking function is used to resolve simultaneity conflicts between the M_i.

The simulation process is carried out by *Processors* that drive the simulation by exchanging messages. Inter-process interaction is carried out through message passing. Each message includes information of the source (or destination), the event simulated time, and a content (consisting of a port and a value). The message is the base class that defines the different messages. There are four different messages: * (internal event), X (external event), Y (model's output) and done (a model has finished with its task).

Two types of Processors exist:

1. *simulators*: drive the simulation of atomic models; and
2. *coordinators*: drive the execution of coupled components and coordinate the activities of all their dependent children.

Figure 25 shows a sample model whose topmost component has three atomic submodels (Atomic Models #1, #2 and #3) and one coupled model (Coupled Model #2). That inner-coupled component is formed by two atomic components (Atomic Models #4 and #5). The processor hierarchy corresponding to this example is shown on the right-hand side of the figure.

A simulator object manages an associated atomic object, handling the execution of its δ_{int} (internal transition function), δ_{ext} (external transition function) and λ (output function). A coordinator object manages an associated coupled object. Only one *root coordinator* exists in a simulation. It manages global aspects of the simulation. It is involved with the topmost-coupled component, which has the highest level in the model hierarchy. Moreover, the root coordinator maintains the global time, and it starts and stops the simulation process. Lastly, it receives the output results that must be sent to the environment.

Appendix II: DEVS-Graphs grammar

In this section, we introduce the grammar for DEVS-Graphs in CD++, and show a simple example of the definition of these models.

II.1 Context-free Grammar for DEVS-Graph models

```
DEVSGraph -> ModelName GGADT_EOL
  GgadRules
ModelName -> GGADT_LBRACKET GGADT_ID
  GGADT_RBRACKET
GgadRules -> GgadRule GGADT_EOL GgadRules
  | GgadRule GGADT_EOL | GgadRule
GgadRule -> InDecl | OutDecl | StateDecl |
  VarDecl | StateDef | InitialState |
  IntDef | ExtDef | VarDef
InDecl -> GGADT_IN GGADT_COLON
  PortInIdList
OutDecl -> GGADT_OUT GGADT_COLON
  PortOutIdList
VarDecl -> GGADT_VAR GGADT_COLON
  VarIdList
VarDef -> GGADT_VARIABLEID GGADT_COLON
  GGADT_CONSTANT
StateDecl -> GGADT_STATE GGADT_COLON
  StateIdList
StateDef -> GGADT_STATEID GGADT_COLON
  GGADT_TIME_CONSTANT | GGADT_STATEID
  GGADT_COLON GGADT_INFINITE
InitialState -> GGADT_INITIAL GGADT_COLON
  GGADT_STATEID
IntDef -> GGADT_INT GGADT_COLON
  GGADT_STATEID GGADT_STATEID
  PortValueOutList Actions
PortValueOutList -> GGADT_PORTID
  GGADT_OUTPUT GGADT_CONSTANT |
  GGADT_PORTID GGADT_OUTPUT
  GGADT_CONSTANT PortValueOutList
ExtDef -> GGADT_EXT GGADT_COLON
  GGADT_STATEID GGADT_STATEID Expression
  GGADT_INPUT GGADT_CONSTANT Actions
Expression -> FunctionCall | GGADT_PORTID
  | GGADT_VARIABLEID | GGADT_CONSTANT
FunctionCall -> GGADT_FUNCTIONID
  GGADT_LPAR ActualParamList GGADT_RPAR
ActualParamList -> ActualParameter |
  ActualParameter GGADT_COMMA
  ActualParamList
ActualParameter -> GGADT_CONSTANT |
  GGADT_VARIABLEID | GGADT_PORTID
StateIdList -> StateIdList GGADT_ID |
  GGADT_ID
```

```
PortInIdList -> PortInIdList GGADT_ID |
  GGADT_ID
PortOutIdList -> PortOutIdList GGADT_ID |
  GGADT_ID
VarIdList -> VarIdList GGADT_ID | GGADT_ID
Actions -> GGADT_BEGIN ActionList
  GGADT_END | GGADT_VARIABLEID
  GGADT_ASSIGNMENT Expression | /*empty */
ActionList -> Action GGADT_SEMICOLON |
  ActionList Action GGADT_SEMICOLON
```

II.II Tokens

GGADT_CONSTANT	
GGADT_IN	reserved word ``in''
GGADT_OUT	reserved word ``out''
GGADT_STATE	reserved word ``state''
GGADT_INITIAL	reserved word ``initial''
GGADT_ID	an identifier
GGADT_STATEID	a state identifier
GGADT_PORTID	a port identifier
GGADT_FUNCTIONID	a function identifier
GGADT_VARIABLEID	a variable identifier
GGADT_INT	reserved word ``int''
GGADT_EXT	reserved word ``ext''
GGADT_VAR	reserved word ``var''
GGADT_CONSTANT	integer o real constant
GGADT_TIME_CONSTANT	time constant in cd++ format hh:mm:ss:nn
GGADT_INFINITE	reserved word ``infinite''
GGADT_COLON	``:''
GGADT_EOL	end of line character
GGADT_OUTPUT	output operator ``!''
GGADT_INPUT	input operator ``?''
GGADT_LPAR	``(``
GGADT_RPAR	``)''
GGADT_LBRACKET	``[ ``
GGADT_RBRACKET	``]''
GGADT_COMMA	``,''
GGADT_BEGIN	``{ ``
GGADT_END	''}''
GGADT_SEMICOLON	``;''
GGADT_ASSIGNMENT	``=``

II.III DEVS-Graphs built-in functions

DEVS-Graphs actions and conditions can be defined using functions (*FunctionCall* in the context-free grammar above). CD++ provides some functions already implemented and new ones can be programmed and added to the framework. Functions are declared by their name and followed by their parameters in parenthesis; for instance, the function *add* can be used as follows "*add(1,2)*". All these constructions can be combined to define the behavior of atomic models. The following built-in functions are available with the tool:

- *value*: checks the value contents of a given port;
- *add/minus/multiply/divide*: arithmetic operations on two sub expressions;
- *pow*: power function;
- *any:* returns the result of an expression;
- *equal/notequal/less/greater/greaterequal*: compare two expressions;
- *and/or/not*: Boolean operators;
- *rand*: pseudorandom number generator.

In addition to these built-in functions, new custom user-defined functions can be added to the framework. In order to add a new function the following steps are required.

1. **Extend the abstract *GgadFunc* class.** This class defines the *executeImpl* method where the actual function needs to be implemented. In addition, a name for the function needs to be specified by implementing the *name* method. Figure 26 shows a sample implementation of the *round* function.
2. **Register the function in *ggadfuncregister.cpp*.** This file contains all the functions that the framework recognizes, so the new function needs to be added. Figure 27 shows the code that includes the new round function.
3. **Recompile CD++.** Once the new simulator executable is generated by the compilation, it is ready to be used with the new function.

II.IV A simple DEVS-Graphs model example

Figure 28 shows a simple DEVS-Graphs model defined using CD++ grammar, and the corresponding graphical representation.

```
class GgadFuncRound : public GgadFunc {
public:
    GgadFuncRound() {
        setFormalParams(1); // specify that this function takes 1 parameter              }

    virtual ~GgadFuncRound() {}

    virtual std::string name() { return "round"; } // name for this function

    virtual GgadValue executeImpl () {
            GgadValue vv = getParameter(0); // read 1st parameter (the cast will automatically
round it)
            return vv ;           }

    virtual GgadFuncRound* create() { return new GgadFuncRound(); }
};
```

Figure 26. Example of a DEVS-Graphs new user-defined function that rounds the parameter.

```
/** Register builtin functions to be used.     */
void GgadSymbolTable::RegisterFunctions() {
    addFunction( "value", new GgadFuncValue() );
    addFunction( "add", new GgadFuncAdd() );
    … // other built-in function are registered here
    addFunction( "greater", new GgadFuncGreater() );
    addFunction( "greaterequal", new GgadFuncGreaterEqual() );
    addFunction( "round", new GgadFuncRound() ); // register the new round function
}
```

Figure 27. Example code to register the new round function in CD++.

DEVS-Graphs Definition

```
[coin]
in: coin_in
out: coin_out
var : coin_amount
state: idle prepare
initial: idle
int: prepare idle
coin_out!coin_amount
ext: idle prepare any(coin_in)?1
            {coin_amount =
coin_in;}
idle: infinite
prepare: 0:0:1:0
coin_amount: 0
```

Graphical representation

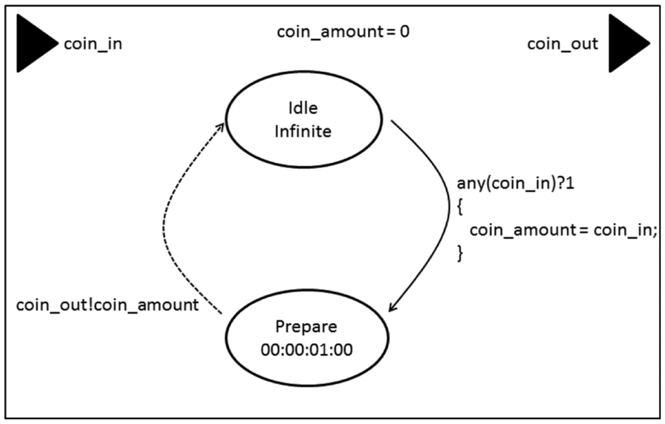


Figure 28. Example of the CD++ textual and graphical representation of a simple DEVS-Graphs model.