Advanced IDE for Modeling and Simulation of Discrete Event Systems

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

[1] Computer Science Department
Universidad de Buenos Aires.
Ciudad Universitaria. Pabellón I
(1428) Buenos Aires, Argentina.
Email: abonaven@dc.uba.ar, rodrigocastro@ieee.org

[2] Dept. of Systems and Computer Engineering
Carleton University Centre of Visualization and Simulation
(V-Sim)
1125 Colonel By Dr. Ottawa, ON, Canada.
Email: gwainer@sce.carleton.ca

Abstract: Creating models and analyzing simulation results can be a difficult and time-consuming task, especially for non-experienced users. Although several DEVS simulators have been developed, the software that aids in the modeling and simulation cycle still requires advanced development skills, and they are implemented using non-standard interfaces, which makes them difficult to extend. The architecture and design of CD++Builder we present here can simplify the construction and simulation of DEVS models, facilitate model reuse and promote good modeling practices by allowing enhanced graphical editing and integration of tools into a single environment. The Eclipse-based environment includes new graphical editors for DEVS coupled models, DEVS-Graphs and C++ atomic models (including code templates that are synchronized with the graphical versions). Integration with Eclipse allows extensibility while simplifying software development, installation and updates.

1. INTRODUCTION

In recent years, the DEVS formalism [1] has become very popular, and several simulators have been implemented using diverse technology. In most DEVS simulators, models are defined using some programming language, which makes a difficult task modeling real world systems for non-expert developers. In order to deal with this problem, several tools now simplify the process of creating DEVS models, executing simulations and analyzing the results. Some provide graphical modeling capabilities, tools for tracking and viewing simulation results. Most of the tools have some modeling limitations and require the users to have programming experience.

While most of these tools allow the graphical definition of DEVS coupled models, they have been developed from the ground up without using standard user interfaces. That makes extending their functionality very difficult, as it is required to know the implementation details. Likewise, atomic models must be defined in some programming language. This makes it difficult for non-expert users to create new models. In some cases where code structure aids are provided, no graphical support is given for atomic model behavior specification. For instance, CD++ [2] provides different languages for specifying DEVS coupled and atomic models. CD++ models can be defined in C++, but graphical representation of the model is also available.

Here, we present new facilities of the CD++Builder tool [3], which have focused on these problems. The integration of several CD++ tools available into a single environment reduces the learning curve for new users and students, and simplifies the M&S processes by avoiding different model formats. Enabling the creation and edition of coupled and atomic models graphically allows users to specify complete DEVS models without programming. In the cases where complex behavior needs to be developed in C++, code templates avoid repetitive and error-prone tasks, and provide basic sample structures that promote good programming and modeling practices. CD++ simulator is continually being updated and new tools are provided. To keep the M&S environment integrated in the future and avoid new tools to be developed in different platforms, CD++Builder features need to be easily reusable and simple to extend.

To fulfill these requirements, CD++Builder refactors the graphical modeling capabilities of CD++Modeler and GGADTool [4] (which were developed in Java and Visual Basic respectively) into the Eclipse environment [5].

Eclipse's plug-in architecture makes it simple to add new features, while guaranteeing the easy reuse of components (by means of the standard frameworks used by CD++ Builder). Thus, all the main activities can be done using the same user interface: composing and defining DEVS models, animation of simulation results, programming of C++ code, compilation of new atomic models into the CD++ framework and launching the execution of simulations.

CD++Builder provides graphical modeling editors, based on atomic DEVS-Graph models [6]. Coupled models are described in a model definition file using a platform-independent language that can potentially be used by other tools as a mean for porting models between simulators. CD++Builder provides code templates with the basic structure to implement the CD++ abstract *Atomic* class, and these models are graphically represented and kept synchronized with graphical coupled editors. Automated regression tests are included, and they help to include new functionalities in CD++Builder, while provide a way to verify the correct behavior of the software after new code is introduced.

2. BACKGROUND

DEVS [1] is a general formalism for modeling and simulation of any discrete systems using hierarchical composition of behavioral models (atomic) and structural models (coupled) with well-defined interfaces. DEVS is independent from any simulation mechanism, which allowed several simulation tools to be developed, tackling different needs and providing advantages on specific scenarios. A non-comprehensive list includes:

- DEVSJAVA [7] provides four Java packages that separate modeling and simulation from user interface, allowing hierarchical model definition and visualization. DEVS coupled and atomic models are built by extending one of the base Java classes provided by the framework. Models need coded and recompiled for changes to take effect (which is difficult for non-Java experts). SimView [8], a graphical component of DEVSJAVA allows the user to specify the model layout in the model's source code. CoSMos [9] allows generating the Java code used to extend DEVSJAVA base classes, which simplifies the definition of atomic models. On the other hand, the user still needs to program, and once the code is generated, model structure cannot be modified (adding/removing ports, changing model name).

- JDEVS [10] is a visual tool that provides a module for executing simulations, a module for the user interface and two modules for 2D and 3D visualization. Though general-purpose models can be defined, the visualization modules are specifically built for natural systems and its graphical editors do not allow creating hierarchical models.

- SimStudio [11] is a web-based framework implemented using Java web technologies, using a layered architecture that supplies modeling, visualization and analysis players. The modeling plug-in, implemented in Flash, allows to get from a graphical specification of a model, a XML file that is used by other tools.

- VLE [12], implemented in C++, is a modeling and simulation environment oriented to integrate heterogeneous formalisms wrapping submodels as DEVS models to enable interoperability. It provides separated modules for the GUI, for visualizing results and a core that implements the simulation algorithms.

- PowerDEVS [13] allows graphical specification of coupled DEVS models, and atomic models are defined 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.

Our work is based on CD++ [2, 14], a modeling and simulation tool that implements the DEVS and Cell-DEVS theory. CD++ has been widely used is several areas of interest such as urban traffic, physical systems, computer architecture and embedded systems. CD++ is implemented in C++ as a class hierarchy where models are related to simulation entities. Atomic models behavior is programmed in C++, and coupled models are defined in a model definition file using a built-in high-level language. Though defining atomic models in C++ gives the modeler flexibility to specify behavior, it requires advanced programming skills. Thus, CD++ also supports defining DEVS-Graph atomic models without a programming language.

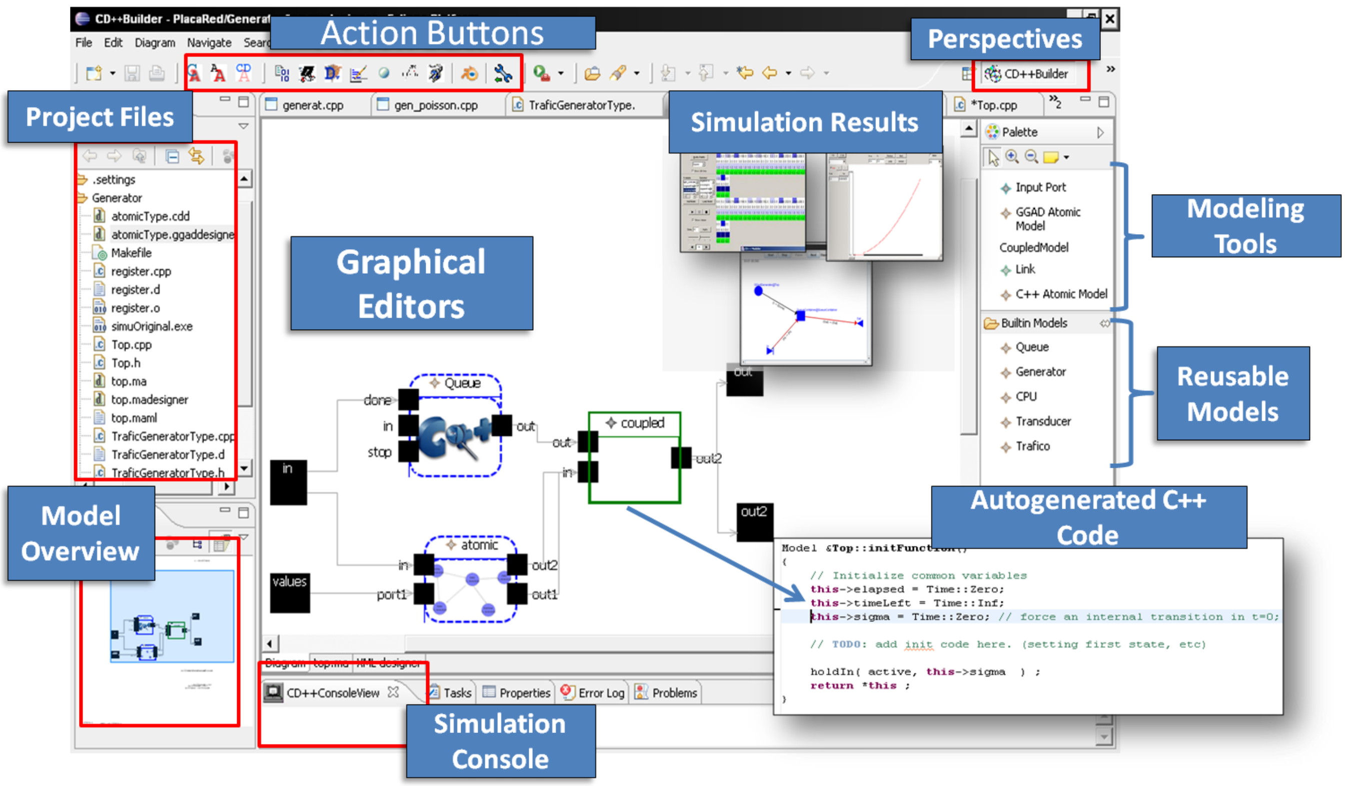
CD++Builder [3] is an Eclipse plug-in that integrates varied applications and utilities that aids in creating CD++ DEVS models, simulating and analyzing results. Among these applications, CD++Modeler provides a graphical editor for coupled and DEVS-Graph atomic editors, and visualization of simulation results. CD++Modeler lacks integration with the rest of CD++ applications and utilities (requiring to export models to different formats) and has some limitations (i.e., not being able to define atomic models in C++). New extensions to CD++Builder were built from the ground up, tackling most of the problems of other CD++ tools and introducing new features available in other simulators into CD++ to facilitate modeling and simulation tasks.

3. CD++BUILDER

Figure 1 shows the CD++Builder environment, which integrates all the capabilities available in CD++ tools with features that facilitate modeling and compilation. Incorporation of all features into the Eclipse environment allowed the development of powerful tools built upon existing features and Eclipse infrastructure.

Action buttons in the top toolbar allow executing external tools. A special section is reserved for CD++Modeler animations, including CELL-DEVS, Atomic and Coupled models. Buttons for launching other legacy tools (CD++ Modeler, GGADTool and Drawlog) are provided. The *Build* button generates a make file and compiles the source code of all atomic models in the project, generating a simulator. The *Execute* button pops up a wizard that allows specifying necessary parameters to run a simulation. New model files (coupled or atomic) can be added using Eclipse *New File* wizard, which will be shown in the project files navigation panel. Editing models is also done within Eclipse interface by means of graphical editors. Coupled model editor allows defining C++ atomic models and generates code structure to extend the Atomic CD++ class. Eclipse allows users to rearrange windows to personalize the interface, and the CD++Builder *Perspectives* layout buttons and panels have been used to improve the modeling of particular scenarios.

As seen on Figure 1, CD++Builder has been now integrated with Eclipse C/C++ Development Tools. It also has been provided with new animation features to allow visualization of simulation results graphically. This graphical framework improves usability including usual editing actions (as copy, paste, undo and redo).



```
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 1. *CD++Builder environment*

Other features that facilitate graphical modeling were incorporated, such as zoom in/out capabilities, flexible look and feel for texts and shapes and different styles for model links to avoid obstructions and overlapping. Both coupled and atomic model editors provide a special pane with tools for easily creating available entities (atomic/ coupled models, links, ports, transitions, states, variables, etc.).

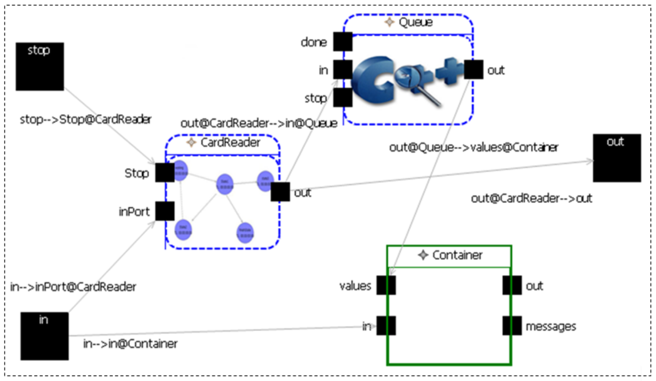


Figure 2. CD++Builder coupled model editor.

The Eclipse Properties view is used to show and edit entities details, the Outline view to show an overview of the model and the New File Wizards for creating new atomic and coupled DEVS model diagrams. CD++Builder uses a common description file and editing domain for models and submodels. Submodels are edited in their own tabs, and all the opened models are kept consistently linked.

In the coupled model editor (shown on Figure 1 and 2), models are graphically represented by colored rectangles with the model's name. Coupled and atomic models are differentiated by color and shape, while inner images are used to distinguish atomic models types (DEVS-Graph and C++). Ports are rendered as black boxes with their names and directed links to represent the associations with the models (which makes easy the creation and understanding of links).

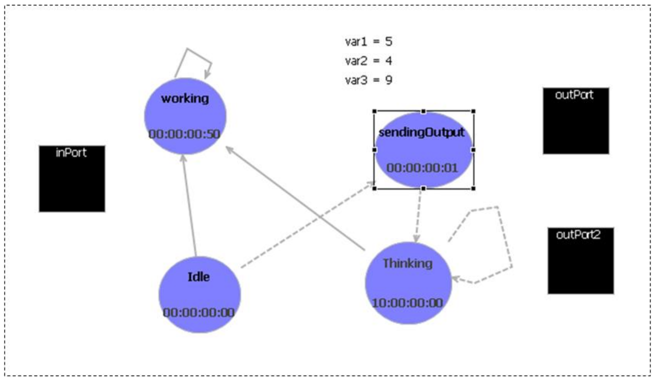
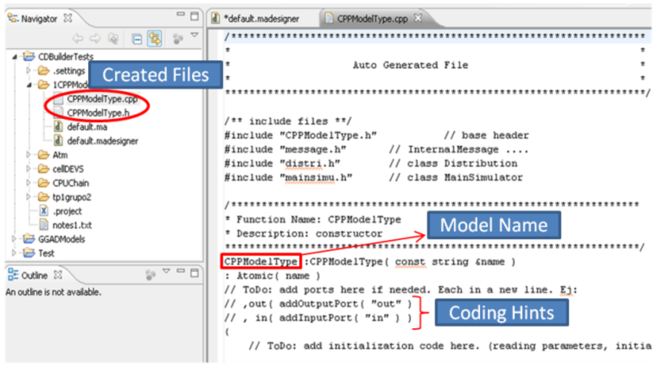


Figure 3. CD++Builder DEVS-graph model editor.

The DEVS-Graph atomic editor (Figure 3) uses DEVS-Graphs notation: atomic model's *states* are represented by circles with *id* and *time advance* values. Internal transitions are represented with dotted arrows and external transitions with full arrows linking origin and destination states.

Implementing C++ atomic models in CD++ requires creating C++ files that define a new class derived from the abstract *Atomic* class, and modifying the *register.cpp* file to register the model within the framework. These tasks are tedious and error prone. To simplify and speed up 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, and *register.cpp* file is automatically updated. The template (shown in Figure 4) supplies the code structure to extend the *Atomic* class, providing helpful comments and code samples. The model name is used to create the .cpp and .h files and to name the new class. All methods that must be implemented (initFunction, externalFunction, internalFunction and outputFunction) are set in place with a brief comments useful for people learning DEVS or CD++. Comments are also used to give an example of how to add input/output ports and parameters to the model. A similar template is also provided to implement the header (.h) file.



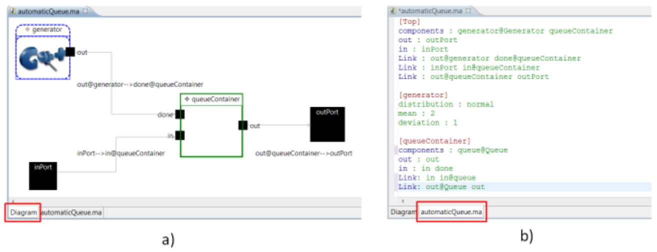
```
/*********************************************************************
*                                                                   *
*                     Auto Generated File                           *
*                                                                   *
*********************************************************************/


/** include files **/
#include "CPPModelType.h"           // base header
#include "message.h"        // InternalMessage ....
#include "distri.h"         // class Distribution
#include "mainsimu.h"       // class MainSimulator

/*******************************************************************
* Function Name: CPPModelType
* Description: constructor
********************************************************************/
CPPModelType::CPPModelType( const string &name )
: Atomic( name )
// ToDo: add ports here if needed. Each in a new line. Ej:
// ,out( addOutputPort( "out" )
// , in( addInputPort( "in" ) )
{
    // ToDo: add initialization code here. (reading parameters, initia
```

Figure 4 – Template-generated code for the Atomic class.

When developing atomic models directly in C++, the model's graphical representation is kept consistent with its C++ underlying code by means of a newly developed parser. When C++ files are modified and saved, the parser recognizes special code structures to identify model name, input/output ports and parameters (constant values to configure atomic models). In this way, the model graphical representation and the code are always synchronized (with no restriction imposed on the code) enabling to modify the graphical metaphor at any time.

New editors have been adapted to reuse the animation features previously available for CD++Modeler (as they have been successfully used for visualizing simulation evolution and results). A control is provided to manage time advance, and links are dynamically highlighted to represent events from one model to another. For coupled models, a block representation is used; for atomic models, the input/output trajectories are shown on different ports over time

CD++ coupled models and DEVS-Graph high-level languages have the power for fully describing DEVS models and enabling graphical representation while not restricting it with visualization-specific information. Thus, model behavior definition and graphical representation are clearly separated. Figures, sizes, layout, colors, and all graphic-specific information are stored in a separate file from model definition. While this separation is conceptually correct, it presents some challenges when implementing graphical editors. CD++Modeler and GGADTool used custom structured file formats for representing model graphics from which model definition could be extracted (through export operations). Nevertheless, the opposite operation (generating a graphical representation from model definition) was unavailable in both tools. CD++ has a vast model repository, so this limitation is a stopper for using these tools: models that were already implemented and tested could not be opened in previous graphical tools. Moreover, once models were exported to CD++ formats they could not be easily updated without losing consistency with the graphical representation. In CD++Builder, the graphical representation information is stored in XMI [15] (XML Metadata Interchange) format and persistence from and to this format is handled by EMF services. New Parsers and Writers were developed to implement translators, from CD++ grammar to graphical representation and vice versa. This way, new coupled and DEVS-Graphs editors can show both views (graphical model representation and textual CD++ model definition) which can be selected by means of small tabs at the bottom (Figure 5). When any of the views is saved, both files are synchronized using the translators to keep them consistent.



```
[Top]
components : generator@Generator queueContainer
out : outPort
in : inPort
Link : out@generator done@queueContainer
Link : inPort in@queueContainer
Link : out@queueContainer outPort

[generator]
distribution : normal
mean : 2
deviation : 1

[queueContainer]
components : queue@Queue
out : out
in : in done
Link: in in@queue
Link: out@Queue out
```

Figure 5 – Model views available: a) graphical b) textual.

Although graphic files contain all the information to rewrite the CD++ model definition completely, the opposite is not true. Thus, when the textual file is saved, the graphic diagram file can be consistently updated, but all graphical information is lost. To overcome this issue, when the textual model definition is saved, the idea is to synchronize the old diagram using its graphical information (layout, figure sizes, colors, etc) to supersede any missing information. To tackle the limitations of previous tools, translators can also generate a new graphical representation from a textual model definition, enabling to use models not built using graphical editors. In this case, a default values are used for the missing graphical information.

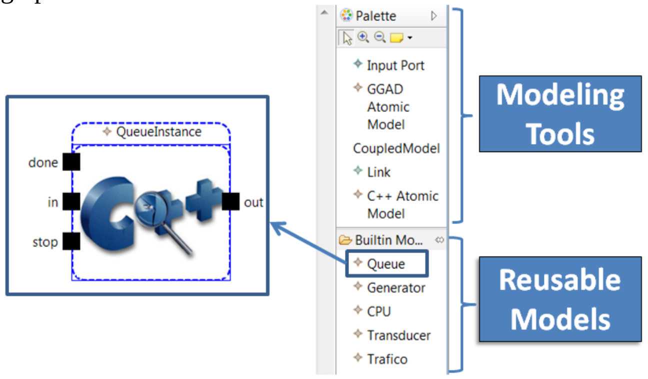


Figure 6 – Coupled model tool pane with reusable models.

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 (Figure 6) including a section for reusable models. This let users that are not familiar with the CD++ model library to know which models previously created are available. By dragging and dropping, these models can be composed within the model being edited.

Installing CD++Builder previously was done by downloading Eclipse and other tools. Installation and updates of CD++Builder Eclipse plug-in is now moved into a centralized schema, integrated with the Eclipse Update Manager package. This allows hosting plug-in compiled code and its metadata in a single publication site, which all users will access for installation and periodical check for updates. The following figure shows this new architecture:

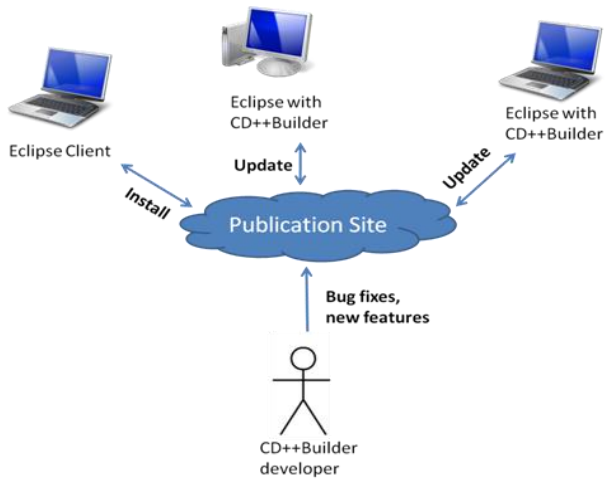


Figure 7. Centralized installation and update architecture.

This new scheme allows easier wizard-guided installation into already running instances of Eclipse. More importantly, it resolves versioning problems; software bug fixes and latest features do not have to be distributed to users individually, but uploaded into a centralized point. Integration with the Eclipse Update Manager allows clients to trigger manual update checks or to configure for scheduling automatic periodic updates.

For the development of key components in CD++Builder, a Test Driven Development [16] approach was used. Automated unit tests provided help improving software quality, and they facilitate extensibility. Tests provide a higher level of certainty that a given functionality is correctly implemented, and as they can be automatically run at any time, they can be used to verify whether a code refactoring or a new feature did not break other features. For this aim, JUnit [17] framework was used, as it provides out-of-the-box support of Eclipse plug-ins testing.

4. ARCHITECTURE AND TECHNOLOGY

The layered scheme in Figure 8 depicts the role of each CD++ component and their main relationships. At the lowest layer (on top of the operating system) the CD++ Simulator implements the simulation algorithms based on DEVS and Cell-DEVS formalisms. CD++ provides different versions of the abstract simulators (e.g. parallel, flat, real-time).

At the next level (Libraries) CD++ provides a basic out-of-the-box atomic model library, including a Generator, a Transducer, and a Queue, which can be directly used to define coupled models. The Core Simulator and the Libraries layers are usually distributed together as the CD++ Simulator. In addition, interpreters for high-level modeling languages are part of the Libraries. These interpreters accept input files, coming from the Modeling level, which can define coupled models compositions, Cell-DEVS models or DEVS-Graph atomic models, without requiring the use of a high-level programming language for their definition.

Other area-specific interpreters are also available in some versions, such as ATLAS for describing urban traffic or M/CD++ to describe continuous models using Bond Graphs and Modelica [2]. All these interpreters are implemented using the core classes, 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 (in this case recompiling CD++ is required).

To execute a simulation, the coupled model definition file and input events must be specified, among other options. A simulation can generates two output files that can be used to track the simulation run; the *.out* file contains the port-value pairs for the output events of the model, while the *.log* file contains all the message passing and synchronization information between different models.

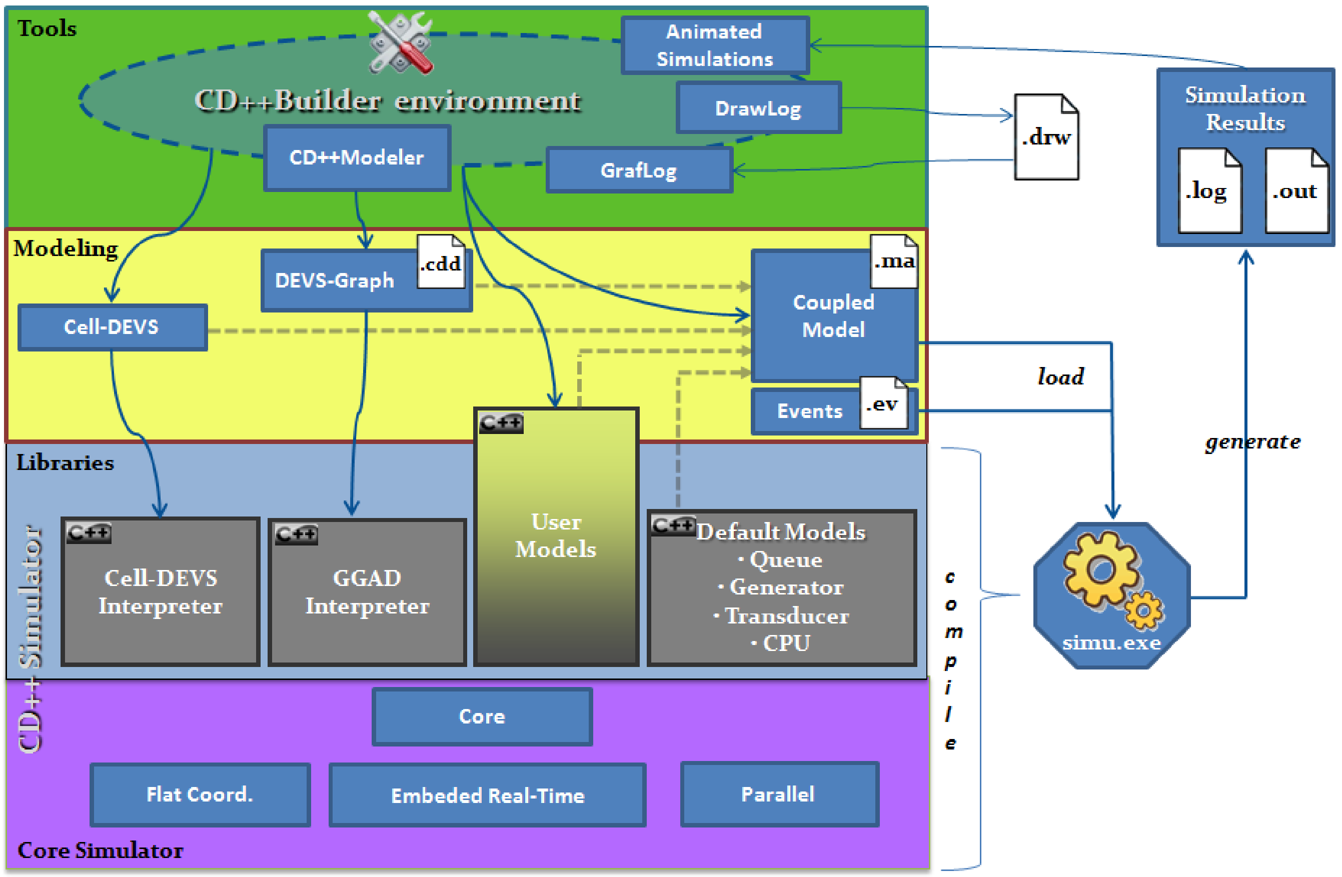


Figure 8. *CD++, high-level modeling languages, execution process and supporting tools.*

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

This layered architecture and the clear separation between simulation execution, model definition, supporting tools and underlying libraries, allows modifying the simulation runtime without affecting already developed models, tools or visualization engines. The same tools and interfaces can be used to facilitate model definition, whether those models will run in a single processor, in a parallel distributed environment or an embedded system.

CD++Builder is implemented on top of several well-known Eclipse frameworks, which provide the overall user interface and core plug-in services. A core requirement for CD++Builder was to allow easy extensibility, as new features are continuously being added and developed in geographically distant places. The Eclipse plug-in architecture enables developing new decoupled features into CD++Builder and integrate them seamlessly. One example of CD++Builder extensibility is the CD++Repository [18], an internet searchable database of CD++ models, which was developed in parallel with the present work in a totally independent way, and has been easily integrated as a part of the same software package.

To implement the graphical editors discussed earlier, several frameworks have been considered. Some of them include the basic graphic libraries Standard Widget Toolkit (SWT), the Abstract Window Toolkit (AWT) and Swing; others more specific include Draw2D and the Graphical Editing Framework (GEF). The first three libraries are based on Java and they provide general GUI controls 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 that builds on top of the SWT library. GEF allows generating a graphical editor based on an existing application model [19]. Due to these reasons, we choose Eclipse's Graphical Modeling Framework (GMF), as this library [20] acts as a bridge between GEF and Eclipse Modeling Framework (EMF) and it specifically tackles the creation of graphical Eclipse-based editors. 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 DEVS editors. A similar framework stack has also been used in [21] for graphical editors of visual languages.

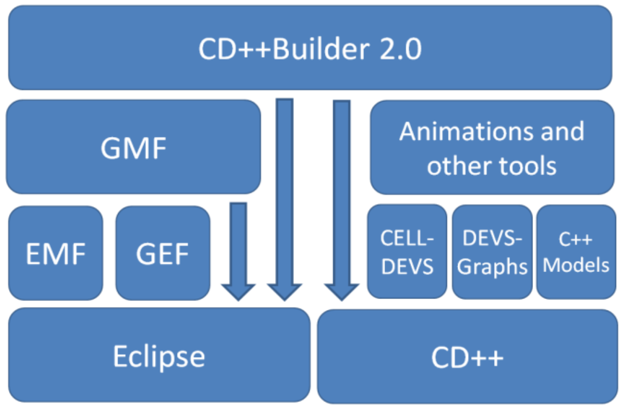


Figure 9 – CD++Builder technology architecture.

Figure 9 shows a description of the conceptual architecture of CD++Builder, considering the tools and software packages used. As we can see, EMF is used to define model entities (the *model* part of MVC), as it provides several services for specifying and maintaining entities. The model can be specified in XMI format, or in a graphical editor, which afterwards EMF uses to generate Java classes and interfaces. The Java classes generated implement the observable pattern, providing methods that notify whenever one of their properties has changed. This greatly helps in keeping the model completely decoupled from the rest of the implementation. Custom code and methods to provide extra behavior to the *model* portion of the architecture can be added to these classes. EMF recognizes special code comments in customized methods not to overwrite them when the *model* is regenerated. EMF also provides persistence and validation services for generated models. A detailed description of the model used to represent DEVS entities in CD++Builder can be found in [22].

GEF and GMF supply base classes which we extended to implement the *view* and *controller* parts of the MVC pattern. GEF extends Draw2D to make it easier to create a graphic representation of the model and provides several base Eclipse editor implementations. GEF controllers need to be provided with a *model* that exposes its properties and notifies whenever a change occurs.

The Eclipse Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF [23]. GMF runtime can be seen as a white-box framework as it combines and extends EMF generated models with GEF's controllers and views, and provides additional services such as transactional support. A *generative* part can be seen as a black-box framework as it allows defining meta-model information in XML files (graphical editors are provided for this purpose), which are used afterwards to generate Eclipse editors code.

In order to be able to generate new graphical editors code based on an EMF model, GMF needs to be provided with three meta-model files: graphical definition model (*.gmfgraph*), tooling definition model (*.gmftool*), and a mapping model (*.gmfmap*). *gmfgraph XML* file describe the shapes and figures that are going to be used in the editor, together with their properties and how they will be composed and layout. The *gmftool* file is used to describe diagram palette tools set, like Selection Tool, Zoom Tool, Creation Tool, etc, and how they will be grouped and shown. The *gmfmap* file references both previous files, and maps EMF model entities to a graphical representation (defined in *gmfgraph*) and associated tools (defined in *gmftool*).

For the purposes of CD++Builder editors, GMF code generation facilities were used in the beginning to define the general editors look and feel, layout and behavior. Nevertheless many necessary features have been developed, customizing and extending the generated code. GMF generates a decoupled infrastructure, where controllers, views and eclipse editors implementation are separated from the model, which is kept in a separated project. This suites CD++Builder's requirements as the model can be reused by other CD++ or DEVS plug-ins without the need to depend on the editors implementation. The model is completely agnostic from graphical and edition details.

5. CONCLUSIONS

We presented a new architecture and the new features available in CD++Builder. This Eclipse plug-in is intended to facilitate the process of modeling and simulation with the CD++ simulator. The tool now:

- Provides an Integrated Development Environment (IDE) for all modeling and simulation tasks (modeling, compiling, simulation execution and analysis).
- Supplies editors that support the complete modeling cycle to be performed in a graphical manner.
- 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.

We showed how CD++Builder, in contrast with previous CD++ tools, provides a unified user interface under Eclipse for all the tasks involved in creating and updating DEVS models, compiling new CD++ atomic models, executing simulations and analyzing results. We also showed how new DEVS graphical editors have been integrated into the Eclipse IDE that facilitates model creation, maintainability and comprehension. These editors are based on CD++'s high-level languages to represent model definition. Graphical information is stored independently and is kept synchronized with model definition automatically.

Additionally, DEVS-Graph and C++ atomic models can now be used to define atomic model behavior. Modeling and definition of new C++ atomic models is simplified by auto-generated code templates, which are kept synchronized with their graphical representation. In addition, the CDT Eclipse plug-in is used for highlighting of C++ code.

Issues about usability and modeling limitations have been overcome with new editors, a tool for easier model reuse, a coupled model editor with discovery, and new install and update mechanisms. In the future, we will synchronize the new right tool pane with the online CD++Repository, to extend the set of models to be reused and facilitate searching and uploading models.

Having all features integrated into the Eclipse environment allows for easy extensibility by adding new plug-ins. An example of this is the Virtual Laboratory of Model-Based Development for Network Processors, (NP) currently under construction by our group. The Lab is fully based on CD++Builder, and is targeted to design advanced embedded control algorithms for the Intel IXP family of NPs [24]. 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 an IXP chip. 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 Hardware In the Loop fashion.

References

[1] Zeigler, B; Praehofer, H; Kim, T. 2000, “Theory of Modeling and Simulation”, 2nd Edition. Academic Press,

[2] Wainer, G. 2002. "CD++: A Toolkit to Define Discrete Event Models". Software - Practice and Experience, Vol. 32, No.13, (November): 1261-1306.

[3] Chidisiuc, C.; Wainer G. 2007, “CD++Builder: An Eclipse-Based IDE for DEVS Modeling”. Proceedings of SpringSim 2007. Norfolk, VA. USA.

[4] Christen, G.; Dobniewski, A.; Wainer, G. 2004, “Modeling state-based DEVS models CD++”. Proceedings of Advanced Simulation Technologies, Arlington, VA.

[5] Budinsky, F; Steinberg, D.; Merks, E.; Ellersick, R.; Grose, T.. “Eclipse Modeling Framework”. Addison-Wesley Professional, 2003.

[6] Praehofer, H.; Pree, D. 1993, “Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs”. 25th Winter Simulation Conference, Los Angeles, CA.

[7] Sarjoughian, H; Zeigler, B. 1998, “DEVSJAVA: Basis for a DEVS-based collaborative M&S environment”. Proceedings of the International Conference on Web-based Modeling & Simulation, San Diego, CA.

[8] Sungung, K.; Sarjoughian, H.; Elamvazhuthi, V. 2009. “DEVS-Suite: A Simulator Supporting Visual Experimentation Design and Behavior Monitoring”. Spring Simulation Multi-conference, San Diego, CA.

[9] Sarjoughian, H.; Elamvazhuthi, V. 2009. “CoSMos: A Visual Environment for Component-based Modeling, Experimental Design, and Simulation”. Proceedings of SIMUTools 2009, Rome, Italy.

[10] Filippi, J-B.; Delhom, J.; Bernardi, F. 2002. “The JDEVS Environmental Modeling and Simulation Environment,” Proceedings of the first Biennial Meeting of iEMSs. Lugano, Switzerland.

[11] Traoré, M. 2008, “SimStudio: a next generation modeling and simulation framework”. Proceedings of SIMUTools 2008. Marseille, France.

[12] Quesnel, G.; Duboz, R.; Ramat, E.; Traoré, M. 2007, “VLE: a multimodeling and simulation environment”. Proceedings of Summer Computer Simulation Conference. San Diego, CA.

[13] Pagliero, E; Lapadula, M; Kofman, E. 2003, “PowerDEVS. An Integrated Tool for Discrete Event Simulation”. (in Spanish). Proceedings of RPIC, San Nicolas, Argentina.

[14] Wainer, G. 2009, “Discrete-Event Modeling and Simulation: a Practitioner’s approach”. CRC Press.

[15] OMG/XMI: XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998.

[16] Beck, K.“Test-Driven Development”. Addison-Wesley, 2003.

[17] Massol, V; Husted, T.. “JUnit in Action”. Manning Publications, 2003.

[18] Chreyh, R.; Wainer, G. 2009, “CD++ Repository: An Internet Based Searchable Database of DEVS Models and Their Experimental Frames”. Proceedings of SpringSim’09, San Diego, CA.

[19] Eclipse Consortium. 2009. “Eclipse Graphical Editing Framework (GEF) – Version 3.4”, available at http://www.eclipse.org/gef. [Accessed on Nov. 18, 2009].

[20] Eclipse Consortium. 2009. “Eclipse Graphical Modeling Framework (GMF)” http://www.eclipse.org/gmf. [Accessed on November 18, 2009].

[21] Ehrig, K; Ermel, C; Hansgen, S; Taentzer, G. 2005. “Generation of visual editors as eclipse plug-ins”. 20th IEEE/ACM International Conference on Automated software engineering, Long Beach, CA, USA.

[22] Bonaventura, M.; Wainer, G., Castro, R. 2009 “Advanced Environment for Discrete Event Simulation”. Internal Report, Carleton University, Ottawa (submitted).

[23] Shatalin, A; Tikhomirov, A. 2006, “Graphical Modeling Framework Architecture Overview”, Eclipse Modeling Symposium, 2006.

[24] Castro, R.; Kofman, E. and Wainer, G. 2009, "A DEVS-based End-to-end Methodology for Hybrid Control of Embedded Networking Systems", 3rd. IFAC Conference on Analysis and Design of Hybrid Systems, Zaragoza, Spain.