The DEVS-Driven Modeling Language and its Graphical Editor

Ufuoma Bright Ighoroje
Computer Science Stream
African University Of Science and Technology,
Abuja, Nigeria.
E-mail: b.ufuoma@gmail.com

Mamadou Kaba Traoré
LIMOS, CNRS UMR 6158
Université Blaise Pascal, Clermont-Ferrand 2
Campus des Cézeaux, 63173 Aubière,
E-mail: traore@isima.fr

Keywords: Discrete Event Simulation, DEVS, Graphical Notation, Eclipse, GMF.

Abstract

We propose DEVS-Driven Modeling Language (DDML), a graphical notation for DEVS modeling and an Eclipse-based graphical editor, Eclipse-DDML. DDML attempts to bridge the gap between expert modelers and domain experts making it easy to model systems, and capture the static, dynamic, and functional aspects of a system. At the same time, it unifies C-DEVS and P-DEVS models. DDML integrates excellent modeling concepts from powerful formalisms and glues them in one unique consistent framework. Eclipse-DDML provides enhanced graphical editing; further simplifying model construction and promoting good modeling practices. Integration with eclipse simplifies software development, installation and updates, while allowing extensibility.

1. INTRODUCTION

DEVS [1] is already established as a universal modeling and simulation formalism. Building DEVS models require the modeler to possess advanced skills in programming and/or mathematics. This makes it very difficult to discuss and verify these models with domain experts. There is also a need to integrate advanced modeling into generic simulation methodologies since both activities are wide apart. Hence, a generic approach that integrates software engineering, modeling and simulation expertise is required. And since there is an underlying simulation operational semantic, there is no need for paradigm/formalism transformation.

In order to realize this solution, an intermediate level of abstraction has to be adopted, which is high enough to be generalized (and accessible to a wide community) and low enough to reduce complexity of code synthesis. This representation needs to express the structural and behavioral characteristics described by declarative and functional models [2] and this must be inherently coherent. This integrative approach should allow the use of an easy language/notation with a potential for formal specification of data and operations, and therefore the simulation system.

The major properties to get in a DEVS-based modeling notation are:

a) **Highly communicable**: domain experts should be able to share and discuss the model easily with simulation specialists about the DEVS model's structure and behaviour captured.
b) **Amenable to formal analysis**: formal methods should be applicable (no ambiguity); so that the user can derive static and dynamic properties of the model and obtain simulation trajectories before running simulations.
c) **High expressive power**: all models that can be expressed in the DEVS paradigm should be easy to describe with the graphical notation. No situation should lead to a non specifiable system using the notation.
d) **Unifying framework**: the notation should provide a common basis for CDEVS and PDEVS.
e) **Supporting tool**: a software tool should allow the drawing of the graphical notation's concepts, as well as automated code synthesis and integrated support for formal analysis.

We develop DDML and our tool to satisfy all of these properties. The concrete syntax of DDML is based on the flowchart, State-Event-Chart, Flow-Trace, State-Event-Trace, and abstract data structure graph [3]. All of these elements are amenable to formal analysis and all of them have their exact DEVS equivalent (which provides the operational semantics). We leverage Eclipse's very rich infrastructure to develop a graphical editing tool for DDML. Our editor provides a rich palette of tools, with drag and drop facilities. Integration with Eclipse's platform also eases software development and makes our tool extensible.

2. BACKGROUND

The formal specification of DEVS provides a means for mathematical formulation of a model. DEVS permits independence of the language or methodology chosen to implement the models, which has allowed several simulation tools to be developed, tackling different needs and providing advantages in specific domains.

Related works address some visual notations and realizations for DEVS models. One approach proposes a framework capable of simulating a DEVS model via Unified Modeling Language (UML) state machines [4]. Then, some set of rules are enumerated to map the UML models to DEVS. The resultant UML models are executable

within simulation frameworks. This approach requires extra efforts to map UML models to DEVS. In addition, it does not satisfy properties (c), (d), and (e) identified in section 1. On the other hand, DDML does not require such advanced mapping to DEVS because it is a direct graphical representation of DEVS.

DEVSJAVA [5] is a DEVS-based modeling and simulation environment written in Java. It provides classes for the users to implement their own DEVS models. DEVS-C++ [6] is a DEVS-based modeling and simulation environment written in C++, which implements parallel execution and supports large-scale systems. While these approaches have been very applicable, they require advanced programming skills in Java/C++ to define DEVS models. Hence, they do not satisfy properties (a), (b), (d), and (e).

CD++ Builder [7] is an advanced IDE integrated with eclipse for modeling and simulating discrete event systems. CD++ Builder integrates a modeler that provides a graphical editor for coupled and DEVS-Graph atomic editors, and visualization of simulation results. Models can be visualized and C++ codes for the models can be generated for simulation. However, properties (b) and (d) are not met.

PowerDEVS [9] provides a graphical representation of coupled models and allows definition of atomic models in C++. It provides code assistance and a special model library enabling re-use of models in a drag and drop fashion. However, properties (b) and (d) are not met satisfactorily.

JDEVS [10] is a DEVS modeling and simulation environment implemented in Java. It allows general-purpose, component-based, object-oriented, visual simulation of models. It has modules for 2D and 3D visualization, built for natural systems. This framework however does not provide an expressive notation to define models. It does not meet properties (a), (b), (c), and (d).

The revised DEVS Diagram [11] is a structured diagram form of the DEVS formalism (C-DEVS) with many similarities with our earlier work [3]. It does not however satisfy properties (b), (d), and (e).

3. DDML

DDML is a direct graphical expression of DEVS Modeling. DDML provides a unifying framework for C-DEVS and P-DEVS modeling. At the same time, it provides a powerful means to specify DEVS-based models using an intuitive and an easy-to communicate specification, which in addition can be amenable to formal verification.

DDML methodology captures the functional aspects of a system using processes represented as flowcharts with ports. The behavioral aspects are captured using state transition diagrams.

3.1. DDML Specification for Atomic and Coupled Processes

DDML uses flowchart-like notations to represent system models or processes. We suggest in DDML to define processes as instances of classes. We also suggest to name ports and to define their domains. Comments should be attachable to include further details and description of the models.

An atomic model diagram represents a distinct system process that cannot be decomposed into sub-processes. A coupled model diagram represents a composite system process that can be decomposed into sub-processes. These sub-processes might be atomic models or coupled models.

Figure 1 shows the DDML graphical notations for coupled and atomic model.

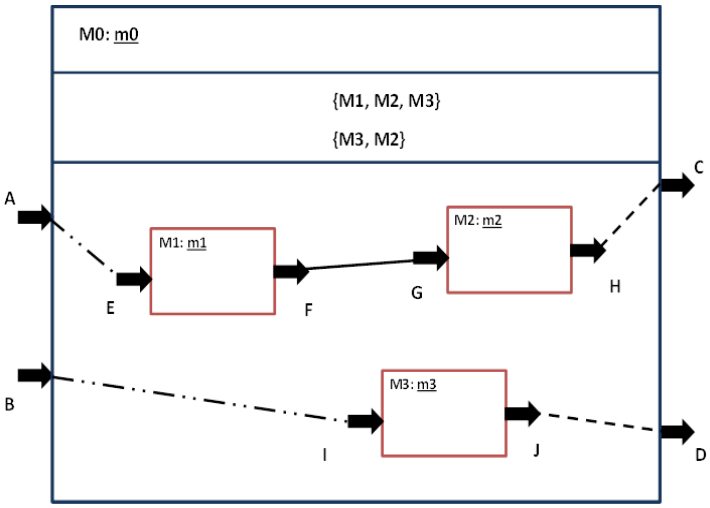

Figure 1: Coupled & Atomic Model Graphical Notation

From the figure, **m0** is a coupled model with sub-models **m1, m2, and m3.** The first compartment of the coupled model is used to represent the name of the process, in addition to the class name (e.g. **M0:m0**) (the class name indicates that the process is an instance of a class. The second compartment is used to represent the select flag which provides a tie-breaking mechanism for concurrency. There is also a compartment that contains sub-models and their couplings. Models (coupled and atomic) have input and output ports. A process receives signals from external processes via the input ports and sends out signals via the output port. For example, coupled model **m0** has input ports **A** and **B** and output ports **C** and **D.**

Processes in DDML are concurrent, asynchronous and they occur instantaneously.

Concurrency implies that processes are parallel. But in the case of a mutual exclusion, DDML uses the select flag (tie-breaker). The flag carries a list of processes sorted by decreasing priorities. From the figure, the select flag {**m1,**

m2, m3} means that if m1, m2, and m3 are simultaneously selected, m1 takes priority and it is executed first. A voting paradox (or *Condorcet's paradox*) may occur in a selection, for example {**m3, m2**} is a voting paradox. This implies that if only m3 and m2 are selected, then m3 is executed first.

The External Input Coupling (**EIC**) (represented by a line-dot-dotted style line) is any connection between a process's input port and a sub-process' input port. There are two EIC connections in the diagram above. They include {(**A—E**) and (**B—I**)}. The External Output Coupling (**EOC**) (represented by a dashed style line) is any connection between a process' output port and a sub-process's output port. There are two EOC connections in the diagram above. They include: {(**H—C**) and (**J—D**)}. Internal Coupling (**IC**) (represented by a solid line) is any connection between two sub-processes within a parent process. There only one IC connections in the diagram above. It is {(**F—G)}.**

3.1.1. Relation to C-DEVS

Recall that a coupled model can be defined in classic DEVS as

CM = <X, Y, D, {Md | d ∈ D}, EIC, EOC, IC, select>

This model is caught in the coupled model DDML diagram as follows:

- The coupled model (**CM**) corresponds to the DDML coupled model diagram
- Each input port p of **X** (e.g. **A** and **B**) of the **CM** is an input port of the DDML coupled model
- Each output port p of **Y**(e.g. **C** and **D**) of the **CM** is an output port of the DDML coupled model
- Each sub-model **d** of **D** (e.g. **m1**, **m2**, and **m3**) is a sub-process of the **CM** (Comments can be used to give additional details about the class to which the sub-process belongs).
- Each element in **EIC** (e.g. **(A—E)** and **(B—I))**, **EOC** (e.g. **(H—C)** and **(J—D))**, or **IC** (e.g. **(F—G)**) is a DDML port-to-port connection as shown above.
- The **select** function is translated into the select flag. Paradoxes might exist and multiple lines should be added to indicate paradoxes.

3.1.2. Relation to P-DEVS

Recall that a coupled model can be defined in parallel DEVS as

CM = <X, Y, D, {M_d | d ∈ D}, EIC, EOC, IC>

The DDML representation of such a model is done like with C-DEVS, but with the following changes:

- Inputs (and outputs) are all synchronized.
- There is no flag (hence the compartment for the select flag is left empty)

3.2. DDML Model of a Traffic System

Figure 2 shows a simple model of a traffic light system using DDML notation. The Traffic system has three sub-processes (Generator, Lights, and Display). The select flag, as shown has four lines to indicate priorities of processes when they are imminent simultaneously.

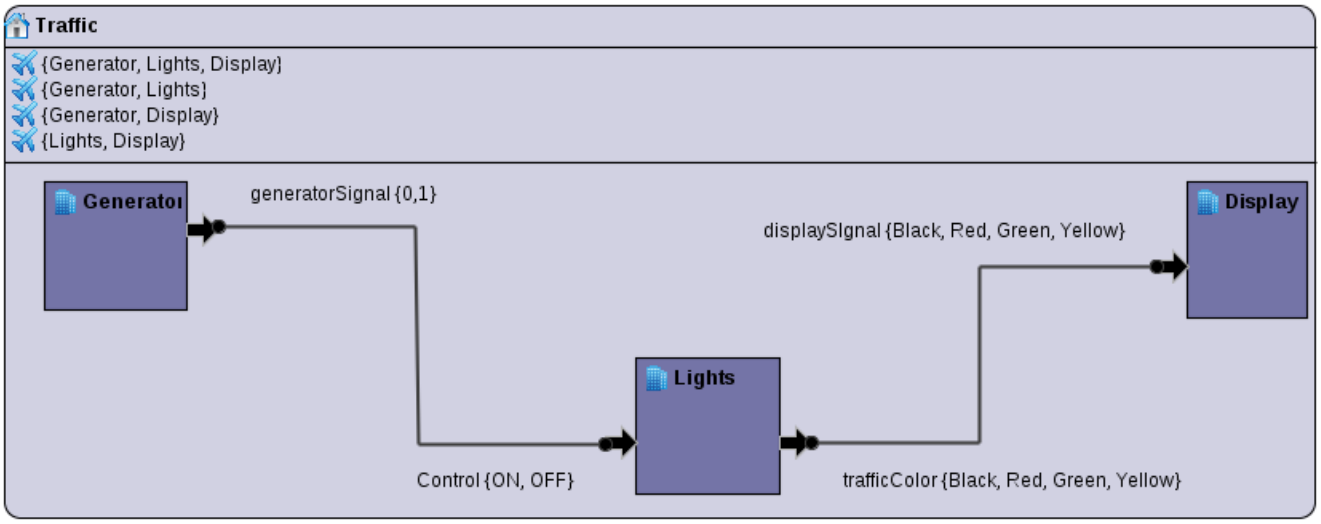

Figure 2: DDML Model for Traffic System

The **Generator** has an input port (**generatorSignal**) with domain (**{0, 1}**) defined. The generator generates a 0 or 1 signal. This signal is sent to the **Lights** process through its **Control** port. The domain of the Control is defined as **{ON, OFF}**, and this port acts as a switch to the Lights system. The Lights process also has an output port, **trafficColor**

which sends the color **{Black, Red, Green, Yellow}** to be displayed to the **Display** system through the **displaySignal** port.

3.3. DDML Specification for States and State Transitions

A process (atomic model) is in a state at a given time. Most systems possess an infinite number of states. In order to effectively capture these states, we use a finite number of state variables to group the infinite states into a finite number of state classes. Hence, in this paper, we define a "state" to be an ***equivalence class of states***, with a particular configuration of state variables. A state also has a duration (the time the system remains in that state)

States in DDML are classified according to the state duration, state activities, and configuration of state variables. A Finite State has a definite duration, a Passive state has an infinite duration, and a Transient state has zero duration. Each of these states is represented in DDML by a rectangle (see Figure 3). The rectangle has four compartments: the upper part is for the name of the state, the second part is for the state properties (values of the state variables, which defines the state), and the third part is for the activities performed whenever the process enters the state (usually in a do-block as shown in the figure) , and the lower part is for the time advance for the given state. Figure 3 shows the graphical notation for Finite state. Transient and Passive States use similar notation. But for Transient States, the time advance is set to **ZERO** and for Passive State, the time advance is set to **INFINITY**.

Figure 3: DDML Finite State Notation

The Initial state diagram represents the first/start state of an atomic model or process. This state is used to declare and initialize all the state variables and to define the subroutines that are used in other states. Variables creation and initialization activities are specified. State functions or sub-routines are also defined. The modeler can use any language to express data structures and algorithms. Figure 4 shows the graphical notation for an initial state. The state variables are defined in the second compartment; and functions (method definitions) of a process are defined in the last compartment.

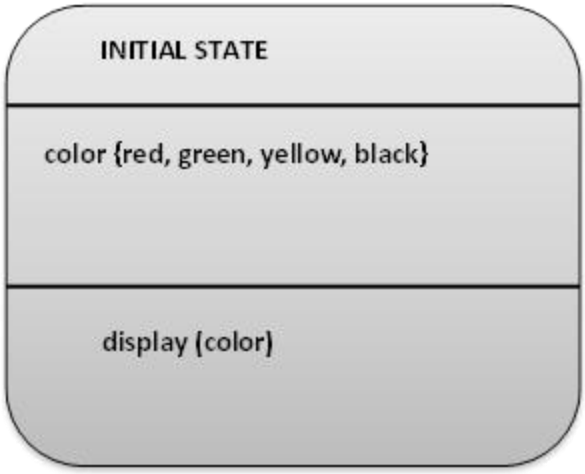

Figure 4: DDML Initial State Notation

States may undergo external, internal, or conflict transition. (Conflict transition is particular to P-DEVS).

The internal state transition is represented by a solid line with an arrow at the end as shown in figure 5. An internal state transition occurs automatically at the end of a definite state or an intermediate state. An action (Lamda), usually sending an output signal is performed at the beginning of the transition and a computation is done at the end (just before it enters the new state). Such a transition always goes from the right hand side of a state to the left hand side of another one. Infinite states do not undergo internal transitions.

The external state transition is represented by a broken line with an arrow at the end as shown in Figure 5. An external state transition occurs when a system receives an external input or disturbance that forces it to change its state. Such transition can occur at a time (elapse time, $\mathbf{e}$ ($\mathbf{0} \leq \mathbf{e} \leq \mathbf{ta}$)). An action (usually an input) is performed before the transition and a computation is done at the end of the transition (just before it enters the new state). External transitions go from the upper or the lower side of a state to the left hand side of another one.

The Conflict transition, which is a transition that goes from one of the right hand side corners of a state, showing that two situations occur simultaneously: the life-time of the state has expired while an external event occurs.

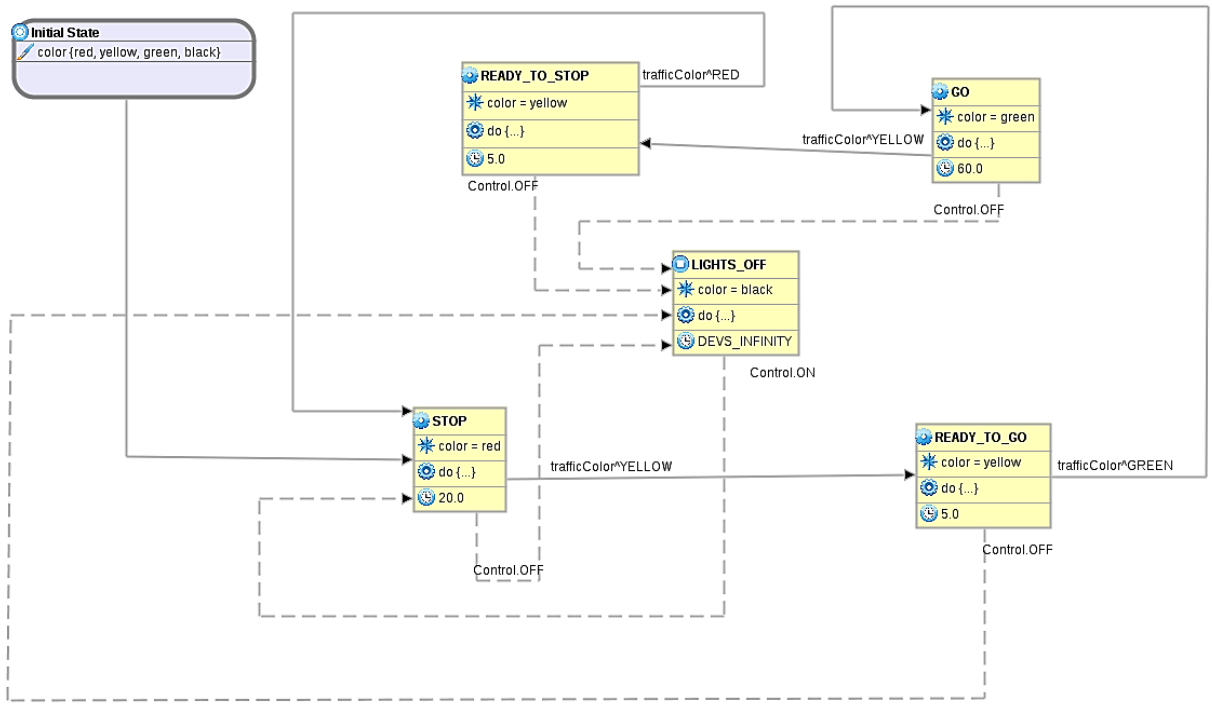

Figure 5: DDML Model for Lights Process Illustrating External and Internal Transitions

We illustrate the external and internal transition notations using the Traffic system example. Figure 5 shows the DDML state graph for Lights process. As shown in the figure, there are five states: **STOP, READY_TO_GO, READY_TO_STOP, GO,** and **LIGHTS_OFF**. Each of these states is defined with its state variables, activities and time advance. Note that state LIGHTS_OFF is a passive state (time advance is **INFINITY**), showing that it has infinite time duration. Hence, it does not undergo any internal transitions. From that state, when the process receives an ON signal through its Control port (indicated with **Control.ON** in the diagram), it undergoes an external transition to STOP state. The system remains in STOP state for 20 seconds (duration) before it undergoes an internal transition to the READY_TO_GO state. Before this internal transition, it sends out a Yellow signal through the trafficColor port (indicated by **trafficColor.^Yellow** in the figure). The system remains in the READY_TO_GO state for 5 seconds, but if it receives an OFF signal through the Control port, it transits to LIGHTS_OFF state (external transition). The system works in this fashion as shown by other states and transitions in the Figure.

DDML also has notation to define a conditional transition. The diamond shaped figure (Figure 5) is used to represent a decision node which indicates a conditional transition. A test is carried out before decision is made on which state to transit to. In the figure shown, the system transits to state **C** if Y ≠5 or transits to state **B** if Y == 5. Conditional transitions could also apply to external state transitions.

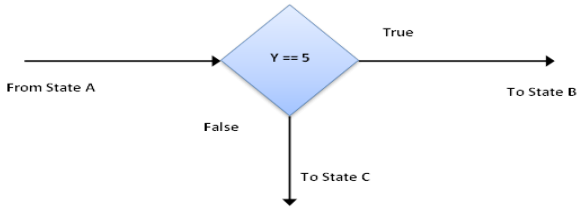

Figure 5: Conditional State Transition

3.3.1. Relation to C-DEVS

Recall, an atomic model is defined in C-DEVS as follows:

$M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$

The DDML representation of the model is an atomic process built as follows:

- X and Y are defined as defined in section 3.1.1.
- An initial state is defined, with declarations: $v \in S_v$. All other states are defined and their corresponding configurations of values for the variables specified. Also the value returned by the time advance (t_a) is indicated for each state at the bottom of the corresponding rectangle. Transient states are states with $t_a(s) = 0$ and infinite states are states with $t_a(s) = +\infty$.

- δ_{int} (s) is defined in the DDML representation as an internal transition from State A to state B, which carries λ (s), by indicating how it is distributed among output ports. Stochastic situations are depicted using decision nodes.
- δ_{int} (s) is defined in DDML representation as an external transition, which carries the input received and shows how this value is distributed among input ports. The associated guard (if mentioned) indicates the value of the elapsed time.

3.3.2. Relation to P-DEVS

An atomic model is defined in P-DEVS as:

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

Where,

X^b and Y^b are bags of inputs and outputs.

S, δ_{int}, δ_{ext}, λ, and ta are defined as in C-DEVS.

$\delta_{con}: Q \times X^b \rightarrow$ S is the conflict function;

The DDML representation is done here like in C-DEVS, with the following changes:

- Inputs (and outputs) are synchronized.
- Each relation $\boldsymbol{\delta_{con}}$ defines in the State-Event-Chart a conflict transition, which carries x and **λ** (s).

4. ECLIPSE DDML EDITOR

It has been shown that DDML presents an easy to use set of graphical notations for defining simulation models based on DEVS. To facilitate model construction using DDML, it is necessary to use a graphical editor with drag and drop features. This will also promote model reuse as models can be constructed, edited, saved, and shared among modelers. In this section, we present our DDML editor based on Eclipse.

We present two editors for DDML: the DDML Coupled Model Editor and the DDML Atomic Model Editor.

4.1. DDML Coupled Model Editor

The DDML coupled model editor contains tools to define DEVS coupled models and sub-models (coupled models or atomic models). Figure 6 below shows a snapshot of the DDML Coupled Model Editor.

The DDML coupled model editor has menu and tool bars, a project explorer, an outline view, a properties view, a rich palette of tools, and a diagram workspace.

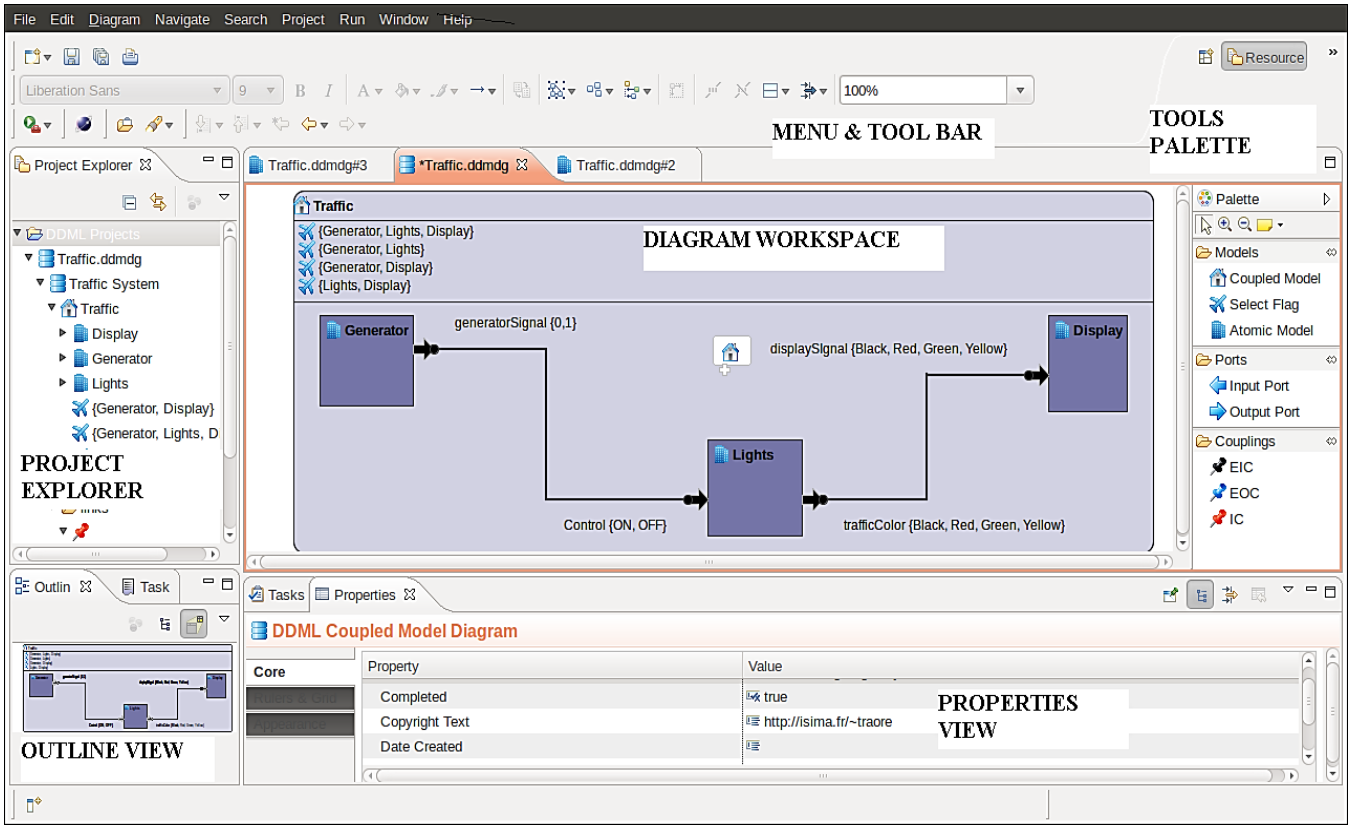

Figure 6: DDML Coupled Model Editor

Using the editor to define simulation models is very intuitive. The Menu Bar has the following menus: File, Edit, Diagram, Navigate, Search, Project, Run, Window, and Help. The toolbar contains common tools for formatting the model diagram. The project explorer view provides a hierarchical view of the project and resources in the Workbench. The outline view shows a graphical outline of the workspace. The palette contains the tools for defining a model.

The drawing workspace is where the model is created. States can be picked from the palette and dropped on the workspace. Ports can be added to a model (coupled or atomic) by simply picking the port tool (Input Port or Output Port) and dropping it on the model. Connections are made between ports. EIC, EOC, and IC connectors can be used in a drag-and-drop fashion.

The Eclipse-DDML workbench provides a properties view that displays the detailed properties for the selected element. Some details about a model element which can are not shown in the drawing workspace are shown in the properties view.

4.2. DDML Atomic Model Editor

The DDML Atomic Model Editor contains tools that can be used to define states and state transitions within a process model. See Figure 7 for a screenshot of the DDML Atomic Model Editor.

Just like the Coupled Model Editor, the Atomic Model Editor has a menu bar and tool bar; a project explorer, a properties view, an outline view, a diagram workspace, and a palette. Apart from the palette, the other sections are very much similar to the Coupled Model Editor.

Just like the Coupled Model Editor, the Atomic Model Editor has a menu bar and tool bar; a project explorer, a properties view, an outline view, a diagram workspace, and a palette. Apart from the palette, the other sections are very much similar to the Coupled Model Editor.

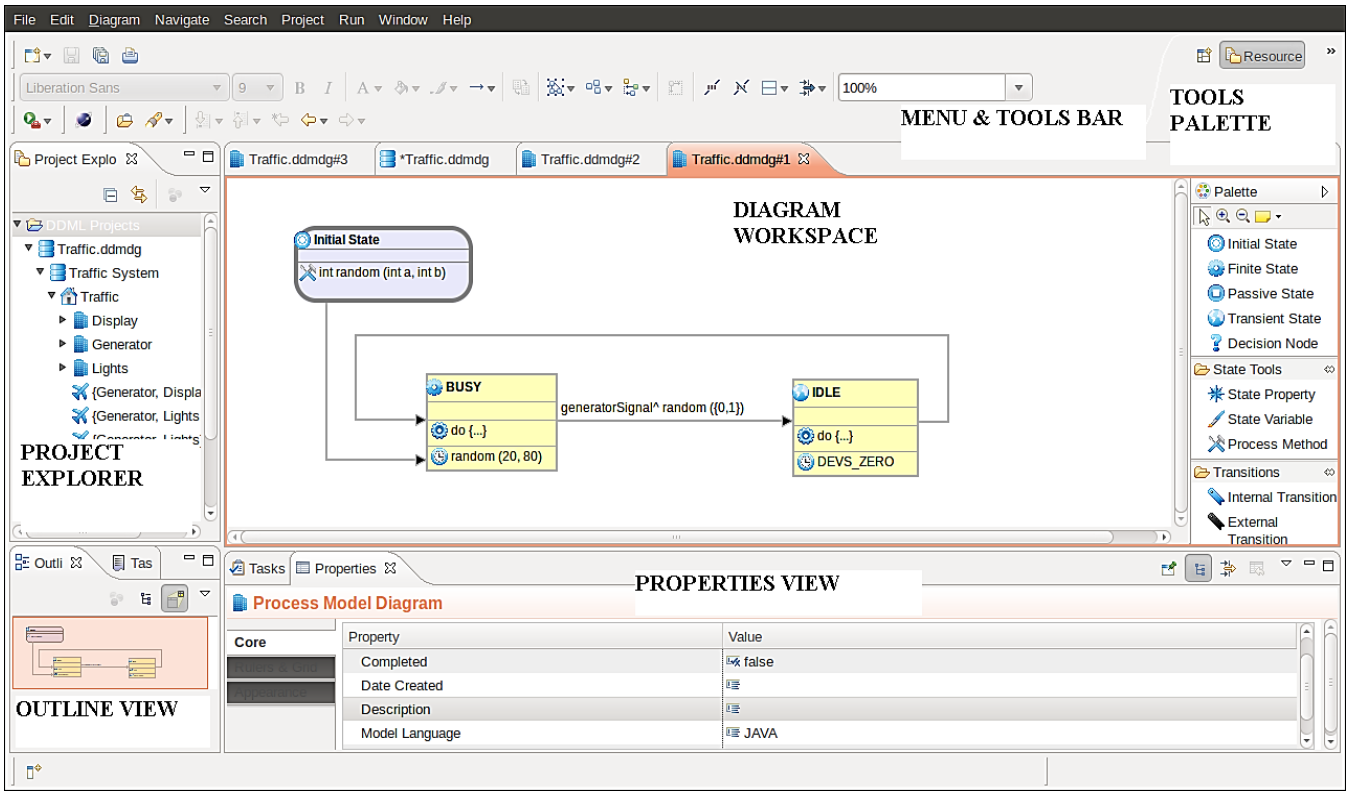

Figure 7: DDML Atomic Model Editor

This editor can be launched by simply double clicking on an atomic model within the DDML Coupled Model editor.

The Passive, Transient, and Finite States contain compartments for defining State Variables (which can be picked from the palette), and Activities. The State Activity is defined within the body of the **do {}** in the properties view and this must be done in the predefined language. The Time Advance for the Passive State and Transient State is set to infinity and zero respectively. External and Internal Transitions can be made by using the Transition tools. This can be done by simply picking the tool and connecting two

states. The Lambda and Computation must be defined for the internal transition while the Trigger and the Computation must be defined for the external transition (this can be done either graphically or in the properties view).

5. ARCHITECTURE AND TECHNOLOGY

In other to implement the graphical editors, several graphics utilities were considered. We considered the native Java Abstract Window Toolkit (AWT) and Swing Libraries. Eclipse also provides the Standard Widget Toolkit (SWT) and JFace library. These libraries are useful for defining form windows but do not possess the capability to build graphical editors. We also considered Draw2D provided by Eclipse. Draw2D was created particularly to handle figures. Eclipse Graphical Editing Framework (GEF) is a powerful library that is based on Draw2D. GEF is specifically created for building graphical editors. But in other to glue the graphical editor with an underlying model (defined by Eclipse Modeling Framework (EMF) [12] Ecore meta-modeling language), we choose to use Graphical Modeling Framework (GMF) [13].

GMF makes it easier to build graphical editors based on an underlying model defined in EMF. It provides a generative component and runtime infrastructure for developing graphical editors. GMF effectively implements the Model-View-Controller (MVC) design pattern, making it possible to define graphical components and model components separately. Four meta-model files have to be defined. The domain model in Ecore (.ecore); the graphical definition model (.gmfgraph) which describes the shapes and figures that are going to be used in the editor; The tooling definition model (.gmftool) which describes the palette tools; and the mapping model (.gmfmap) which maps the domain model, the graphical definition model, and the tooling definition model.

The mapping model is used to generate the generator model (.gmfgen) which is used to generate the editor code. Several tweaks were made to the generated code to fulfill the desired functionalities of our editors.

The graphical properties are separated from domain properties (stored in different XML files).

6. CONCLUSION

We presented DDML, a graphical notation for defining DEVS models. We showed how DDML maps to the formal DEVS specification and how it captures the dynamic, static and functional aspects of a system. DDML is a natural and intuitive approach to modeling. Its notation can easily be understood by both domain experts and modelers. We also presented Eclipse-DDML with rich editors for defining DEVS coupled models and atomic models.

DDML is a contribution towards making DEVS available to a wider community. Our tool is built as an eclipse plugin, hence it can integrate and be integrated into other utilities using Eclipse platform. This also means that it is extensible, easy to manage, and update.

Next steps include the following:

- The Eclipse-DDML tool should be integrated with some methods for formal analysis
- Our editor should be extended to include ability to automatically generate simulation code for DEVS libraries like SimStudio, DEVSJAVA, and DEVS-C++.
- Our tool should evolve into an integrated development environment for all simulation tasks (modeling, simulation, analysis of results, verification, and validation of simulation models, and visualization of simulation results.

References

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

[2] Fishwick, P. A. 1995. “Simulation Model Design and Execution: Building Digital Worlds," Prentice Hall.

[3] Traore, M. K. 2009. “A Graphical Notation for DEVS”. Proceedings from the Spring Simulation Multiconference 2009.

[4] Mooney J. 2008. DEVS/UML – A Framework for Simulatable UML Models. M.S. Thesis, Computer Science and Engineering Dept., Arizona State University, Tempe, AZ, USA.

[5] 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.

[6] Zeigler, B., Y. Moon, D. Kim, and D. Kim. 1996. DEVS-C++: A high performance modeling and simulation environment. Proceedings of 29th Hawaii International Conference on System Sciences, Honolulu.

[7] Wainer, G. 2009. “Discrete-event modeling and simulation: a practitioner's approach”. CRC Press.

[8] Kim, T. G. 1994. DEVSIM++ user’s manual. CORE Lab, EE Dept, KAIST, Taejon, Korea.

[9] Pagliero, E; Lapadula, M; Kofman, E. 2003, “Power-DEVS. An Integrated Tool for Discrete Event Simulation”. Proceedings of RPIC, San Nicolas, Argentina.

[10] Filippi, J. B., and P. Bisgambiglia. 2004. JDEVS: An implementation of a DEVS based formal framework. Environmental Modeling and Software 19:261–274.

[11] Song H. S. and Kim T. G.. 2010. DEVS Diagram Revised: A Structured Approach for DEVS Modeling. Proceedings from European Simulation Conference 2010, Belgium: 94 – 101.

[12] Steinberg D. et al. 2008. “Eclipse Modeling Framework.” 2nd Edition, Addison-Wesley Professional.

[13] Gronback R. C. 2009. Eclipse Modeling Project: A Domain-Specific Language Toolkit. Addison-Wesley.