A Unified Modeling Framework for Discrete Event Simulation

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, France
E-mail: traore@isima.fr

Keywords: Modeling, DEVS, Discrete event simulation, DDML.

Abstract

We propose a unified modeling framework for discrete event simulation. Our approach aims to provide a generic framework to simplify construction of simulation models, and at the same time provide the capability to capture the dynamic, static, and functional aspects of a system under study. We adopt a graphical modeling technique that integrates best practices from several powerful modeling paradigms and unify them in a consistent framework. The model when defined using this framework would be amenable to formal analysis. We also provide a graphical editor that makes these models easy to construct, reuse, and extend.

1. INTRODUCTION

1.1. Motivation

Building simulation models have become a very complex activity as a result of the wide variability between modeling activities and simulation activities. The primary issue in modeling is to develop multiple abstractions of a real/imagined system and capture these abstractions with algorithms that represent the static, dynamic and functional aspects of the system under study. These models have to be transformed into a simulation semantic which involves with the identification of timing aspects, data preparation and time management. Hence, there is a need to bridge advanced modeling techniques and generic simulation methodologies. This implies that modeling, software engineering, and simulation expertise should be integrated seamlessly into the modeling and simulation process.

Our proposed modeling framework satisfies these requirements. We adopt an intermediate level of abstraction, which is simple enough to be generalized (understandable by a wide community of modelers and users) and low enough to reduce the complexity of code synthesis for simulation purposes. At the same time, we can capture the functional, dynamic, and static aspects of a system under study. The underlying modeling paradigm is Discrete Event Simulation (DEVS) formalism.

1.2. Why DEVS?

[Zeigler et al. 2000] describes DEVS as a "unique form of representation that underlies any system with discrete event behavior". Models expressed using the Discrete Event System Specification (DEVS) represent a class of systems theoretic models that permit parallel event-based behavior to be expressed concisely and in a manner that lend themselves to formal verification. Although many different simulation formalisms have been advanced over the years, the DEVS formalism has emerged as the preferred formalism due to the fact that other formalisms have been proven to have an equivalent DEVS representation. DEVS support full range of dynamic system representation. In particular, a Differential Equation System Specification (DESS) can have an approximate Discrete Time System Specification (DTSS) by selection of a sufficiently small constant time interval (discretization). A DTSS model, in turn has an equivalent DEVS representation. Also, quantization of events in a DESS system can result in an approximate DEVS model. As such DEVS approach can be used to model discrete systems, continuous systems (approximate), and hybrid systems.

DEVS approach to modeling is appealing also because it separates modeling, simulation, and the experimental frame (EF). It eases verification. Its simulation approach allows parallel, distributed, or real-time execution of models thereby enhancing speed. DEVS promotes a user-oriented approach (incremental approach) to modeling and simulation.

Due to its generality, DEVS does not propose a means to specify systems in details. The mathematical foundations are defined but the specification details are left to the modeler. Hence, we propose a DEVS-Driven Modeling Language (DDML) to fill this void. DDML uses notations that unify C-DEVS and P-DEVS (two formalisms as the foundational frameworks for DEVS). DDML uses graphs that are amenable to formal analysis to define DEVS models.

1.3. Structure of Paper

In the following section, we present DDML and show how its graphical notations map to the DEVS formalism. In section 3, we present our Eclipse-DDML tool with

illustrative examples. In section 4, we conclude and suggest the future direction for our unified modeling framework.

2. DEVS-DRIVEN MODELING LANGUAGE (DDML)

DDML makes DEVS accessible to wide community of users and modelers by providing a means for defining DEVS-based models graphically. It uses a set of graphical notations to specify, visualize, analyse, verify, and document the characteristics and behaviour of some real or imagined system under development.

DDML uses processes to define the functional aspects of a system and this is described graphically using flowcharts with input and output ports. Dynamic aspects of a system are captured by using notations similar to state/activity diagrams. The static aspects are described using abstract structure graphs. The static aspects are automatically derived from the functional and dynamic aspects thereby clearing all ambiguities that might result if a modeller uses different diagrams to represent different views.

2.1. DDML Processes

A simulation model is analogous to a business process that interacts with its environment through input and output ports. It receives messages via its input ports and sends out messages via its output ports. In DDML, the processes are instance of classes and the ports have to be defined by the domain or a set of allowable signals.

A process which cannot be decomposed is said to be atomic. Processes that have sub-processes are coupled. Processes communicate with each other via couplings (constraints are usually attached to these couplings to provide more prescriptions about data that are transferred). These process couplings can either couple two input ports (from a process and a sub-process), two output ports (from a sub-process and a process), or an input port and an output port (from two distinct processes). These couplings are termed External Input Coupling (EIC), External Output Coupling (EOC), and Internal Coupling (IC) respectively.

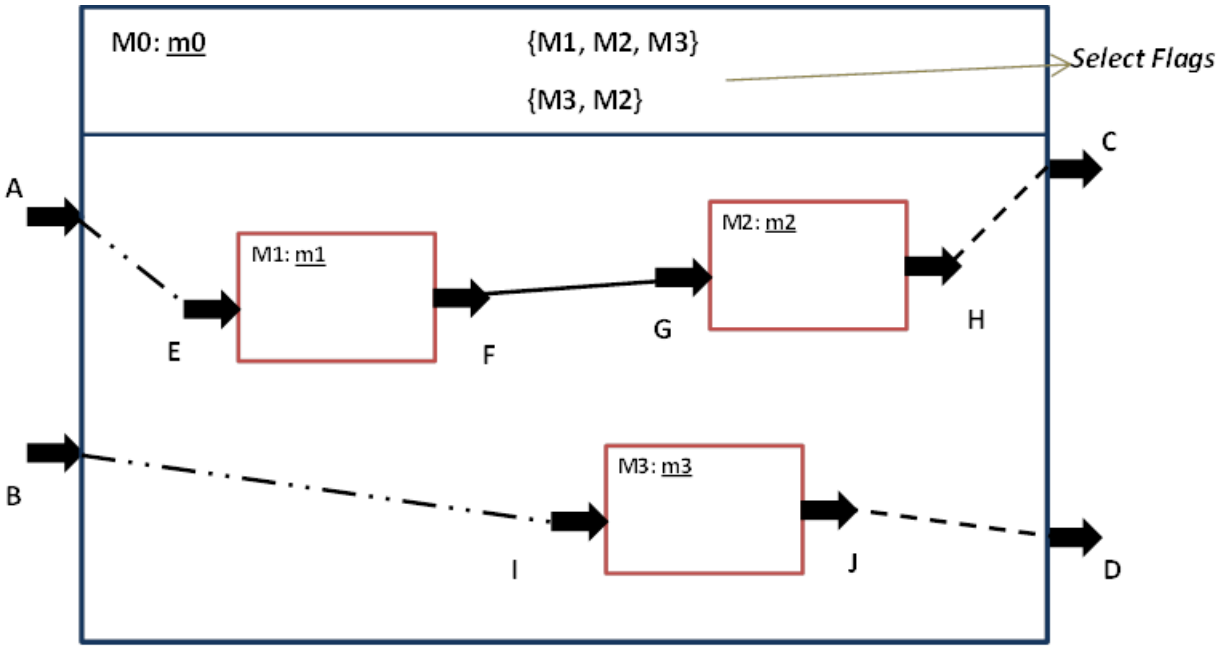

Figure 1: DDML Coupled Model and Atomic Model

Figure 1 above shows a coupled process (**m0**) with three sub-processes (**m1, m2, m3**). Process **m0** has input ports (**A** and **B**) and output ports (**C** and **D**). Each port has a port type which specifies the domain or set of allowed variables. The External Input Coupling (**EIC**) (represented by a line-dot-dotted style line) is any connection between the parent's input port and a child's' 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 the parent's output port and a child'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 processes. There only one IC connections in the diagram. It is **{(F—G)}.**

Processes usually occur concurrently, but in the case of a mutual exclusion, a flag is used to determine priorities. A

paradox could occur when determining priorities. Figure 1 has a compartment for specifying the tie breakers (Select Flags). From the figure, if **m1**, **m2**, and **m3** are concurrently activated, then **m1** is selected to be processed. But if only **m3** and **m2** are activated, then **m3** is selected. This kind of situation is known as *Condorcet's paradox* (or voting paradox). Several flags can be added to indicate paradoxes. Flows are also asynchronous and instantaneous.

2.1.1. Relation to Classical DEVS Theory

According to the DEVS theory, a coupled model can be defined in classic DEVS as

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

Where X and Y are input and output ports respectively. D refers to atomic models, EIC, EOC, IC are couplings as defined earlier. Select refers to the select function.

This model is represented in the coupled model DDML diagram (Figure 1) as follows:

- The coupled model corresponds to the DDML coupled model diagram
- Each input port p of X (e.g. A or B) of the CM is an input port of the DDML coupled model
- Each output port p of Y (e.g. C or D) of the CM is an output port of the DDML coupled model
- Each sub-model d of D (e.g. **m1**, **m2**, or **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 flags. A paradox may occur and there are as many lines as there are paradoxes.

2.1.2. Relation to Parallel DEVS Theory

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

CM = <X, Y, D, {Md | d ε 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)

According to the closure property, every coupled model can be regarded to be a DEVS atomic model. The closure property guarantees that the coupling of several class instances results in a model of a particular class, allowing hierarchical construction. This implies that we can have a coupled model (child) within another coupled model (parent).

2.2. DDML States and States Transition

At any given time, a process is in a particular state. A moderately sized system can have an unimaginable size of state spaces. Hence the size of the state space can become infinite leading to a problem of state explosion. We solve this problem by using a finite number of state variables to partition the infinite number of states into a finite number of state classes. Hence, we define a "state" here to be an *equivalence class of states*. Multiple individual states are said to be in the same equivalence class ("state" in DDML) if and only if they are equivalent under the given relation, which is defined by a configuration of state variables. For example, if we define a process by two state variables, X and Y, we can say that the individual states defined by {**X=4, Y=10**}, {**X=7, Y=9**}, and {**X=8, Y=11**}, are equivalent under the relation {**X>3, 7<Y<12**}. Hence, the configuration {**X>3, 7<Y<12**} is a state in DDML.

We classify states in DDML based on the duration of a state, configuration of state variables, and state activities. We have **Finite State** (to represent a state with a definite duration); **Passive State** (to represent a state with an infinite duration); and **Transient State** (to represent a state that transits instantaneously).

We use rectangles to represent these states in DDML (see Figure 2). The rectangle has four compartments: the upper part is for the name of the state, the second part is for the values of the state variables (which defines the state), and the third part is for the activities performed whenever the process enters the state, and the lower part is for the time advance for the given state.

The **Initial state** represents the first state for a process. This state is used to define all the state variables and to define the subroutines that are used in other states. Variables creation and initialization activities are specified (in a global way, any internal activity which is not a call to a subroutine can be specified in a "do" block). The modeler can use any language to express data structures and algorithms. Figure 2 also 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.

A state can be composite. Such a state is composed of sub-states that have common properties (every property of the composite state stands also for each of its sub-states, but sub-states can have their specific additional properties, and these can be specified in the sub-state graph). The duration of a composite state can be explicit or not (in the later case, sub-states have their own durations). We call this a **state cluster**. Figure 2 illustrates a state cluster in DDML.

State transitions occur between states in a process. As a result of grouping of states using state variables, these transitions should be seen as a transition between state groups rather than transitions between definite states.

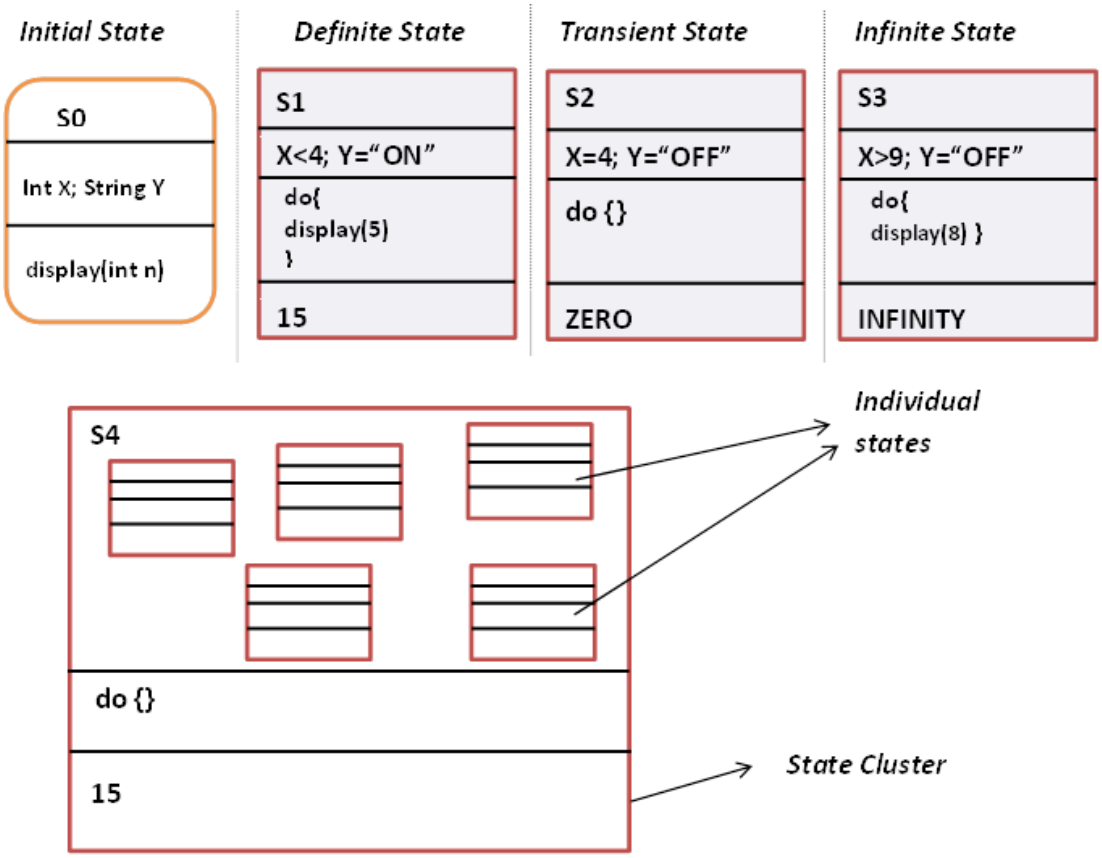

Figure 2: State Notations in DDML

The **internal state transition** is represented by a solid line with an arrow at the end as shown in Figure 3 (**S5—S6**). An internal state transition occurs automatically at the end of a definite state or an intermediate state. An action (usually sending an output signal, e.g. ***Board^.Red***) is performed at the beginning of the transition and a computation (e.g. ***Y="OFF"***) 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.

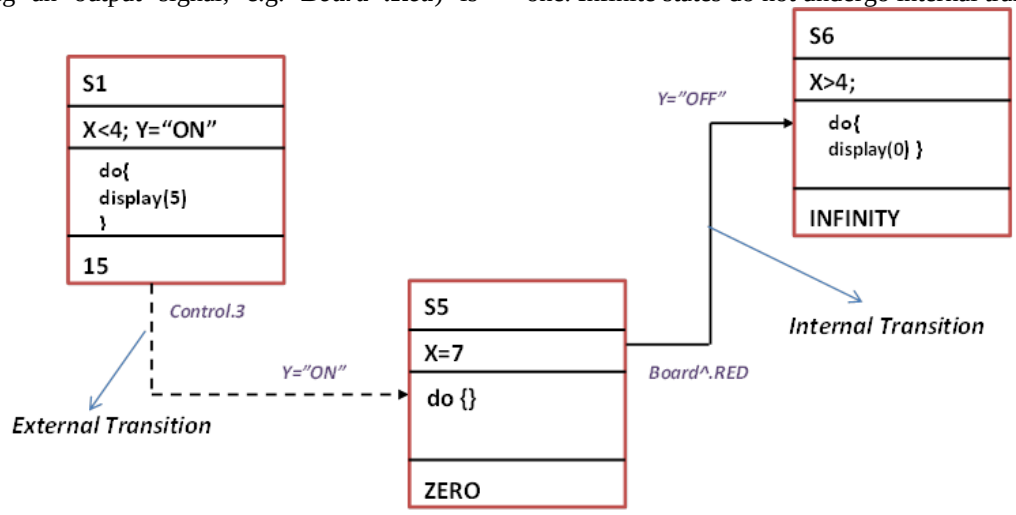

Figure 3: External and Internal State Transitions

The **external state transition** is represented by a broken line with an arrow at the end as shown in Figure 3 (**S1—S5**). An external state transition occurs when a system receives an external input or disturbance that forces it to change its state (in the diagram, Control port receives a signal with value 3, depicted as **Control.3**). Such transition can occur at a time (elapse time, **e** (**0 ≤ e ≤ ta**)). A computation is done at the end of the transition (just before it enters the new state e.g. (**Y="ON"**) as shown in Figure 3. In DDML notation, 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. This is illustrated in Figure 4. A conflict transition also has an action and computation.

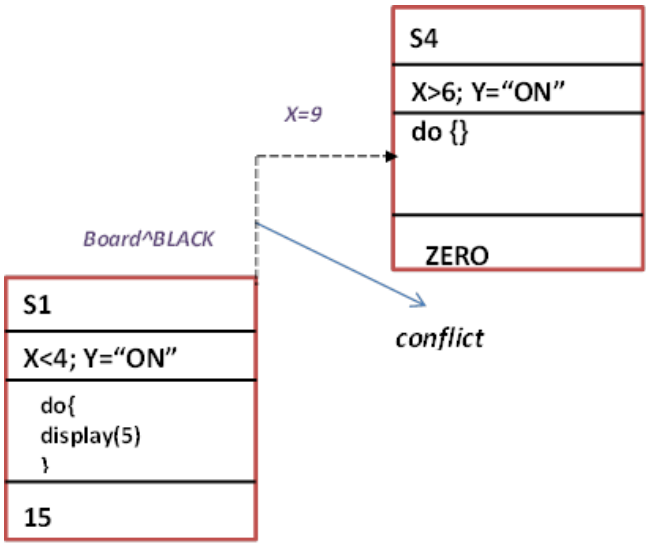

Figure 4: Conflict Transition

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.

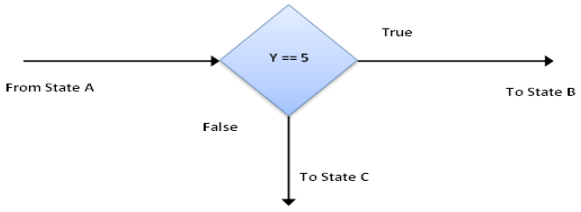

Figure 5: Conditional Transition

2.2.1. Relation to Classical DEVS Theory

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

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

Where X, Y are input ports and output ports respectively. S is the set of states. δ_{int}, δ_{ext} are internal and external states transitions respectively. **λ** is the output function and ta is the time advance function.

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

- X and Y are defined as defined in section 3.1.4.
- An initial state is defined, with declarations: $v \varepsilon S_v$. 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) (output), 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.

2.2.2. Relation to Parallel DEVS Theory

An atomic model is defined in P-DEVS as:

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

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 conflict transition (Figure 4), which carries X and **λ** (s).

3. DDML MODELING TOOL

In this section, we present our graphical modeling software for constructing DDML models. Our software has two editors, the DDML Coupled Model Editor (Figure 6) and the DDML Atomic Model Editor (Figure 7). The former is used to define DDML processes and sub-processes whereas the latter is used to further define the structure and behaviour of a process by constructing states and states transitions. Both editors are Eclipse plug-ins and they provide powerful and intuitive graphical capabilities for constructing DDML models. The domain information for a model is saved as an XML file.

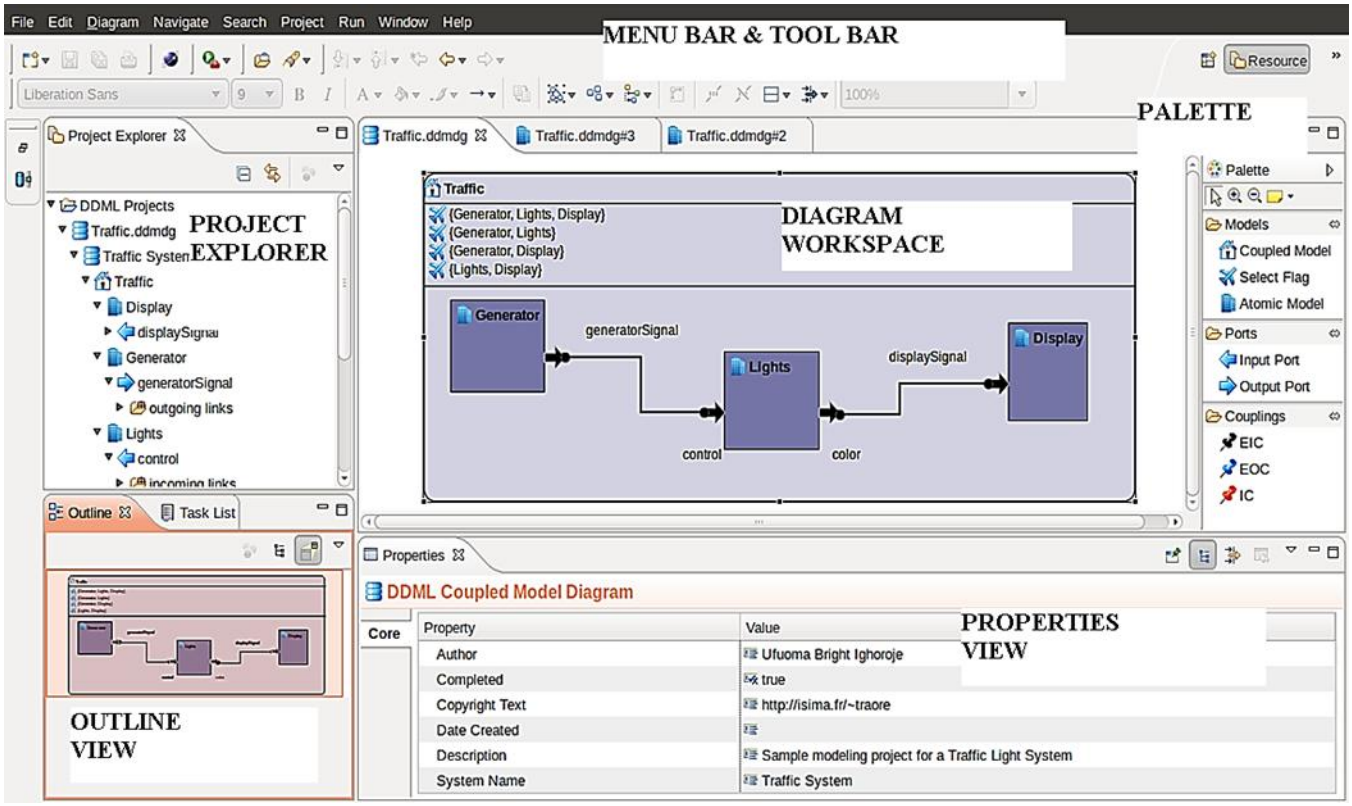

Figure 6: DDML Coupled Model Editor

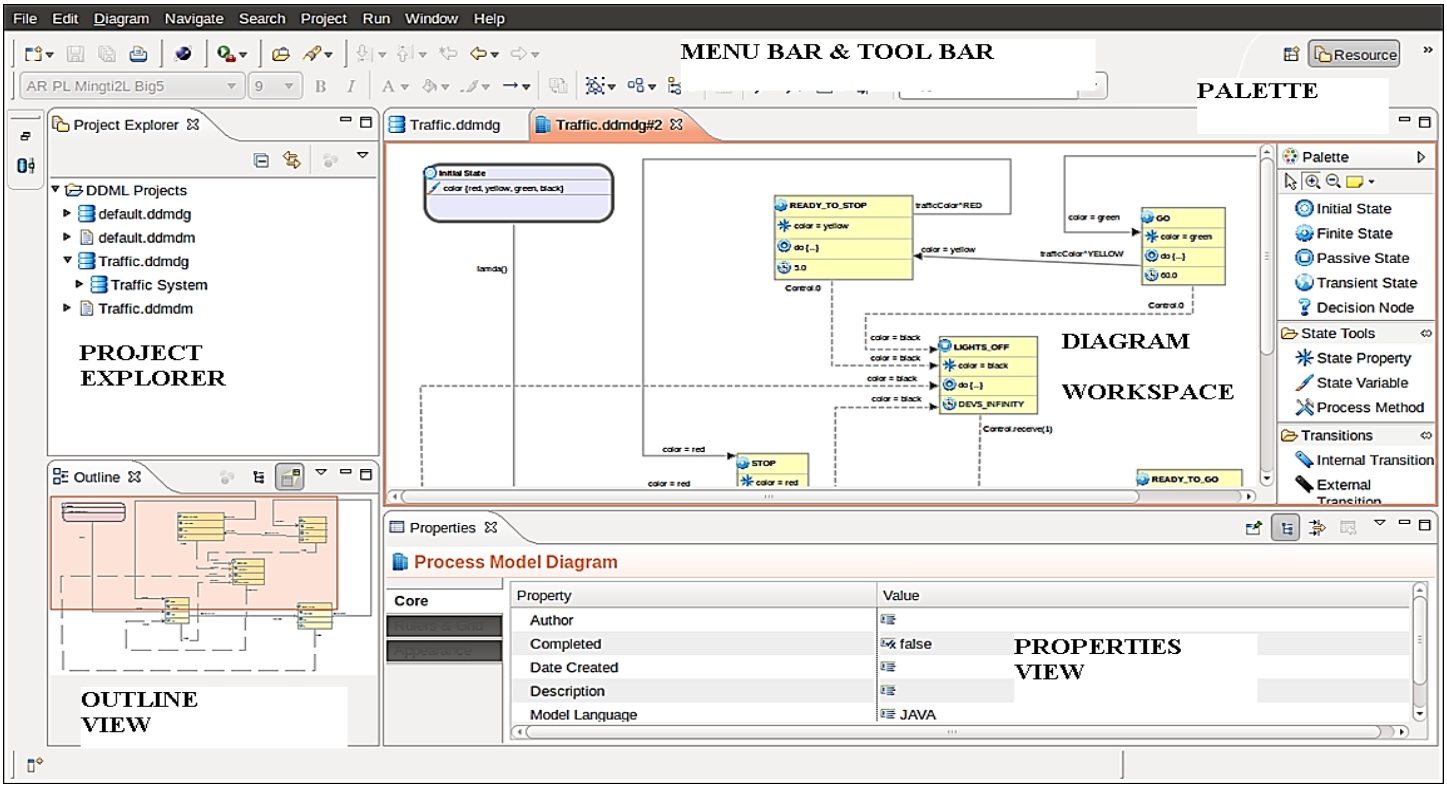

Figure 7: DDML Atomic Model Editor

4. CONCLUSION

In this paper, we proposed a generic approach to bridge advanced modeling and generic simulation methodologies. Our approach involves unifying all aspects of simulation modeling into one unique framework. DEVS has provided the underlying formalism for this.

Similar works have proposed using familiar methodologies to glue modeling and simulation. [Mooney 2008] proposes a framework capable of simulating a DEVS model via Unified Modeling Language (UML) state machines. A set of rules is enumerated for the creation of UML models. Adherence to these rules results in models that are both DEVS and UML compliant. Resultant UML models are executable within DEVS simulation frameworks such as [Sarjoughian and Zeigler 2008]'s DEVSJAVA. While this approach is beneficial, it requires additional efforts to map the modeling paradigm (UML) to the DEVS simulation framework.

Other proposals are based on state-based notation. [Christen et al. 2004] proposes State-Based DEVS models for CD++. [Risco et al. 2007] uses UML state diagrams to construct models and transforms these diagrams to DEVS state machines using XML.

Recently, [Song and Kim 2010] revised DEVS Diagram, a structured diagram form of the DEVS formalism (C-DEVS) with many similarities with our earlier work [Traore 2009]. DEVS Diagram uses the concepts of ports and messages for structuring sequential events and it introduces the concepts of phase transition diagram to simply represent state transitions. It does not however provide a means to represent P-DEVS models.

DDML does not require advanced mapping to DEVS as it is purely based on DEVS. This also makes it amenable to formal analysis and automatic code generation for several DEVS libraries. DDML provides a unifying framework for both C-DEVS and P-DEVS using the same graphical notations as we have shown in this paper.

DDML has the following goals:

- To provide a unified modeling framework for discrete event simulation.
- To provide users with a ready-to-use, expressive visual modeling language for building simulation models.
- To provide a basis of communicating via DEVS models.
- To be independent of any particular programming language and development process
- To provide a formal basis for understanding the DEVS formalism and to make DEVS accessible to the entire computer simulation community
- To support higher-level development concepts such as collaborations, frameworks, patterns and components.
- To integrate best practices from various powerful modeling paradigms.

We also presented a graphical editing tool to further simplify the construction of DDML models. Our editor is integrated into Eclipse, hence it leverages Eclipse's powerful development environment. Eclipse also provides additional advantages of extensibility, easy installation and updates, and integrated software development environment.

Next steps would involve integrating methods of formal analysis into the Eclipse-DDML editor and generation of simulation codes from DDML models for DEVS libraries. This would be integrated into SimStudio (a collaborative simulation infrastructure) proposed by [Touraille et al. 2009].

References

[Christen et al. 2004] Christen G., Dobniewski A. & Wainer G. Modeling State-Based DEVS Models in CD++. Proceedings of MGA, Advanced Simulation Technologies Conference (ASTC'04). Arlington, VA. U.S.A.

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

[Risco et al. 2007] Risco-Martín J.L., Mittal S. & Zeigler B.P. From UML Statecharts to DEVS State Machines Using XML. *Multi-paradigm Modeling, IEEE/ACM International Conference on Modeling Languages and System*, Nashville, Sept.

[Sarjoughian and Zeigler 2008] Sarjoughian, H; Zeigler, B. 1998. "DEVSJAVA: Basis for DEVS-based collaborative M&S environment". Proceedings of the International Conference on Web-based Modeling and Simulation, San Diego, CA.

[Song and Kim 2010] Song H. S., Kim T. G. "DEVS Diagram Revised: A Structured Approach for DEVS Modeling. Proceedings from European Simulation Conference, ESM '10.

[Touraille et al. 2009] Touraille L., Traoré M.K. and Hill D.R.C. 2009. SimStudio: Proposition d'une Infrastructure Collaborative de Modélisation, Simulation et Analyse de Systèmes Dynamiques Complexes. 13ème JSED SPI, ISSN 0249-7042. Modélisation de systèmes

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

[Zeigler et al. 2000] Zeigler B.P., Praehofer H. & Kim T.G. Theory of Modeling and Simulation. Integrating Discrete Event and Continuous Complex Dynamic Systems. *Academic Press*.

BIOGRAPHY

Ufuoma B. Ighoroje is a graduate (M.Sc) student in Computer Science at the African University of Science and Technology, Abuja (Nigeria). His research interests are in computer modeling and simulation, discrete event systtems, software engineering, and formal specification.

Mamadou K. Traore is Associate Professor in Computer Science at the Blaise Pascal University of Clermont-Ferrand (France). His research focuses on formal specification, symbolic manipulation and automatic code generation of simulation models.