DDML: a support for communication in M&S

Oumar Maïga
Université des Sciences,
Techniques et Technologies
Bamako, Mali
maigabababa78@yahoo.fr

Ufuoma Bright Ighoroje
African University
of Science and Technology
Abuja, Nigeria
uighoroje@aust.edu.ng

Mamadou Kaba Traoré
CNRS UMR 6158, LIMOS
Université Blaise Pascal
Clermont-Ferrand, France
traore@isima.fr

Abstract
This paper presents a new approach to simulation modeling for discrete event systems that integrates two system design perspectives: object-oriented paradigm and system theoretic approach. The resulting formalism is a highly communicable visual language called DDML (the DEVS Driven Modeling Language) which federates modeling concepts from UML (Unified Modeling Language) and DEVS (Discrete Event System Specification). Such a combination facilitates the modeling process and improves communication between experts in the system domain and experts in modeling and simulation domain. A case study of urban traffic modeling is given to illustrate the use of DDML as a support for communication in Modeling and Simulation.

Keywords: DEVS, DDML, UML.

I. INTRODUCTION

Modeling is a process of abstract knowledge representation of a real or conceived system. In modeling and simulation of complex systems, it is important that the abstract model should accurately capture the structural and behavioral aspects of the system. It is often difficult to develop a simulation model in the early stages of system development because this would require knowledge of the system domain, modeling methodology, and model execution techniques. Domain experts are concerned with system characteristics, problems and behavior. Simulation experts use mathematical formalisms, algorithms, and/or computer programs to develop abstractions of systems. These abstractions must be translated into a simulation semantic to investigate system properties. Due to the difference in concerns and expertise between domain engineers and simulation experts, it is required to utilize a framework that supports communication and cooperation between domain experts and experts in modeling and simulation. What is needed to achieve this is an intermediate notation which is highly communicable, expressive, and low enough to reduce the complexity of code synthesis for simulation and analysis. This representation should be able to express the structural and behavioral characteristics of complex systems without ambiguities. This is the case in the area of software, systems, and data engineering where Unified Modeling Language (UML) [1], System Modeling Language (SysML) [2], and Entity Relationship Diagrams (ERD) [3] respectively have been used as specifications that provide easy visual constructs that facilitates cooperation between domain engineers and technical experts. Employing similar constructs for simulation modeling would facilitate the system development process.

Our objective is to propose a visual language to increase communication between domain experts and M&S experts and to allow automated simulation code generation.

The approach combines the use of UML and the DEVS formalism [4]. UML is used for domain analysis; therefore domain experts can bring the accurate knowledge about the system. DEVS is used at the simulation stage, allowing simulation specialists to carry on their expertise. The same way UML is well recognized as a modeling standard for software development, the DEVS paradigm is recognized as a computation model that unifies discrete event formalisms [13]. Also, it has been proven that continuous systems can have an approximated representation in DEVS, either by discretizing the simulation time or by using the quantization method [4].

A major issue is the gap between the representation obtained from domain analysis and the simulation model generated for this system. Domain experts and simulation experts need to communicate, sharing a common language to get from the domain knowledge to the application of simulation techniques. We have defined the DDML (DEVS Driven Modeling Language) for this purpose. It is a visual language that augments the UML class diagram with DEVS concepts. So, UML class diagrams can be built first, and then augmented to produce DDML classes. The operational semantics of DDML is given by the well-established DEVS simulation algorithms, so the simulation code can be automatically generated from the DDML classes.

The rest of the paper is organized as follows: after a brief review of the state of the art in Section II, we present our approach in Section III where DDML syntax and semantics are exposed. An application is shown in Section IV; it focuses on the urban traffic of the city of Bamako (Mali). Concluding remarks are given in Section V.

II. STATE OF THE ART

In the literature, there are two broad classes of approaches for defining a domain specific language based on UML. One class of approaches involves the use of UML profiles [1] using stereotypes and user-defined properties (tagged values). This approach allows benefiting from existing tools that support UML notation for modeling and code generation. It also allows a precise definition of the concrete syntax of the language but does not allow the clear definition of the abstract syntax of the domain-specific language because it also inherits problems of UML (i.e., the lack of precise semantics). The other class of approaches uses MOF (Meta Object Facility) to extend the UML metamodel or define directly the language metamodel independently of UML. The problem with this approach is the need to develop a new supporting tool for the language. It allows a clear definition of the abstract syntax, concrete syntax and semantics. Other approaches use UML profiles to define the concrete syntax and a metamodel for defining the abstract syntax of the language.

Related works address some visual notations and realizations for DEVS models. Some notable graphical approaches integrating UML to DEVS modeling include DEVS/UML [5], Executable UML with DEVS (eUDEVS) [6], and the object-oriented co-modeling methodology [7]. DEVS/UML provides a representation of DEVS models as UML state machines. A simulator has been proposed for such models. The issue here is the difficulty of expressing DEVS concepts in UML state machines. eUDEVS approach is doing the opposite of DEVS/UML, i.e. it transforms UML models into DEVS models but the models obtained are in a restricted class of DEVS called FD-DEVS (Finite and Deterministic DEVS). Other approaches such as CD++ Builder [8], PowerDEVS [9], and DEVS graph [10] are based on the definition of new languages. In [7] the authors present an approach using a subset of UML to support the process of object-oriented modeling based on DEVS. This approach uses UML to model parts of the behavior of objects and DEVS to complete the missing discrete event information. They use the use case diagrams, class diagrams and sequence diagrams of UML. The procedure to transform a given sequence diagram model in DEVS is adapted only to finite sets of states.

Our approach is close to the latter, apart from that we address also systems with infinite state sets. We provide a clear definition of the abstract syntax of DDML by a metamodel in UML with OCL (Object Constraint Language) constraints. We associate a concrete syntax and a well-defined operational semantics. Our approach presents two advantages: it gives a complete methodological approach allowing various experts in the loop, and the resulting visual language allows the automated generation of simulation code. DDML is also amenable to formal analysis (this is a very important aspect though not discussed in this paper).

III. DDML APPROACH

The methodology is based on two major stages:

(1) Domain analysis, where the domain class diagram is built according to the knowledge provided by domain experts. All entities are identified and described in terms of attributes and methods.

(2) Identification of dynamic components as atomic or coupled models, and specification of their structure and behavior. Simulation experts must closely discuss with domain experts to drive this stage. Indeed, domain experts provide knowledge on components behavioral rules and simulation experts translate them in terms of state sets, state transitions, output functions and time advance functions (according to the DEVS-driven semantics of DDML). The visual aspect of DDML makes these exchanges easier since the mathematical descriptions are hidden behind DDML's intuitive and user-friendly nature.

DDML has an abstract syntax and an associated concrete syntax which borrows visual elements from UML. The main element in DDML is a model, which is an augmented UML class (Figure 1). Additional information are input and output ports through which the model interacts with other models and dynamics that translate the way the model behaves in time. While a classical UML modeling would include ports in the attributes and would use a UML state chart to express the dynamics, we prefer highlighting these elements differently. Ports are public attributes that carry data and they require their corresponding data structure be clearly specified (as UML classes, primitive types or even DDML models in their turn). UML state chart are not adequate to describe models behavior for simulation, due to the fact that the simulation time must explicitly be taken into account. In addition, it is important that such behavior is visually linked to the model; indeed, the attributes, methods, and more specifically the ports are used in the specification of the dynamics. More about ports and dynamics are detailed in the language's abstract syntax.

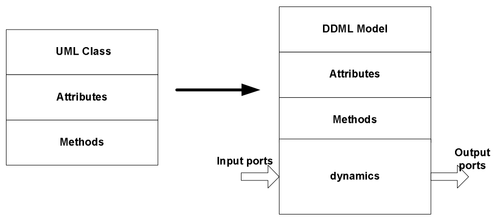

Figure 1. DDML model

A. DDML abstract syntax

The metamodel defining DDML abstract syntax is given in Figure 2. A *Model* is either an *AtomicModel* (i.e., it cannot be decomposed) or a *CoupledModel* (i.e., it is decomposed into smaller models, which in their turns can be atomic or coupled). *Models* interact with each other through *IOInterfaces* (or ports), some are *InputPort* (for the receipt of data) and some are *OutputPort* (for the storage of data generated by the model). All data are viewed as *AbstractEvents*, some are *events*, and some are *Bags* (i.e., a group of events received at the same time).

At any given time, an *AtomicModel* is in a particular state. The state of a system is defined using assignments on *StateVariables* and these are defined by their name and domain. A moderately sized system can have an unimaginable size of state spaces. The size of the state space can even become infinite leading to a problem of state explosion. In order to represent a system with a large number of states, we use a finite number of state variables (after abstracting the key variables that can give a reasonable description) to partition the state space into a finite number of *Configurations*. A C*onfiguration* is a partition or subdivision of the state space into non-overlapping and nonempty subsets of states. It is defined by a set of properties upon the state variables. Each

Configuration has a *TimeAdvance* function that maps each member state to a real time advance. We classify *Configurations* in DDML based on their *timeAdvance* and state activities. The activities are the operations that are carried out by the system when in that state. Based on this, we have *Finite* (to represent a *Configuration* with a definite duration); *Passive* (to represent a *Configuration* with an infinite duration); and *Transient* (to represent a *Configuration* that transits instantaneously). State transitions occur between states in an atomic model. As a result of grouping of states using state variables, these transitions should be seen as transitions between *Configurations* rather than transitions between definite states. State transitions could be internal, external, or confluent. Before entering the next *Configuration*, a reconfiguration of the state variables occurs (*computation*). The *Internal* transition occurs automatically when the life time (defined by the *timeAdvance* function) has elapsed. The system automatically transits into a new *Configuration*. At the beginning of such transition, an output (*lamda*) is sent out through an output port. Since the life time of an *Infinite Configuration* is infinite, it does not undergo internal transition. The *External* transition occurs when a system receives an external input (*trigger*) or disturbance that forces it to change its *Configuration*. The Conflict transition corresponds to the confluent function of DEVS. It is used as an alternative to the select function in the coupled network to resolve conflicts within the system.

The *CoupledModel* and its components (*Models)* are connected via *Couplings.* The couplings are split into *EIC* (External Input Coupling, couplings of coupled model input ports to input ports of some components), *EOC* (External Output Coupling, couplings of component output ports to output ports of the coupled model), and *IC* (Internal Coupling, couplings of component output ports to input ports of components). Components in a coupled model are concurrent and asynchronous. Concurrency implies that they are parallel but in the case of mutual exclusion, a *Select* (corresponding to the select function in DEVS specification) is used for arbitration*.* It is simply a list of components sorted by decreasing priorities.

Times Traces describes the *History* of the system under consideration within a time frame. This is defined as a trace of the Model. *AM_FootPrint* is a trace of an *Atomic_Model (IOS)* and *CM_FootPrint* is a trace of *Coupled_Model (CN).* They are composed of *tuples* in succession from the first to the last.

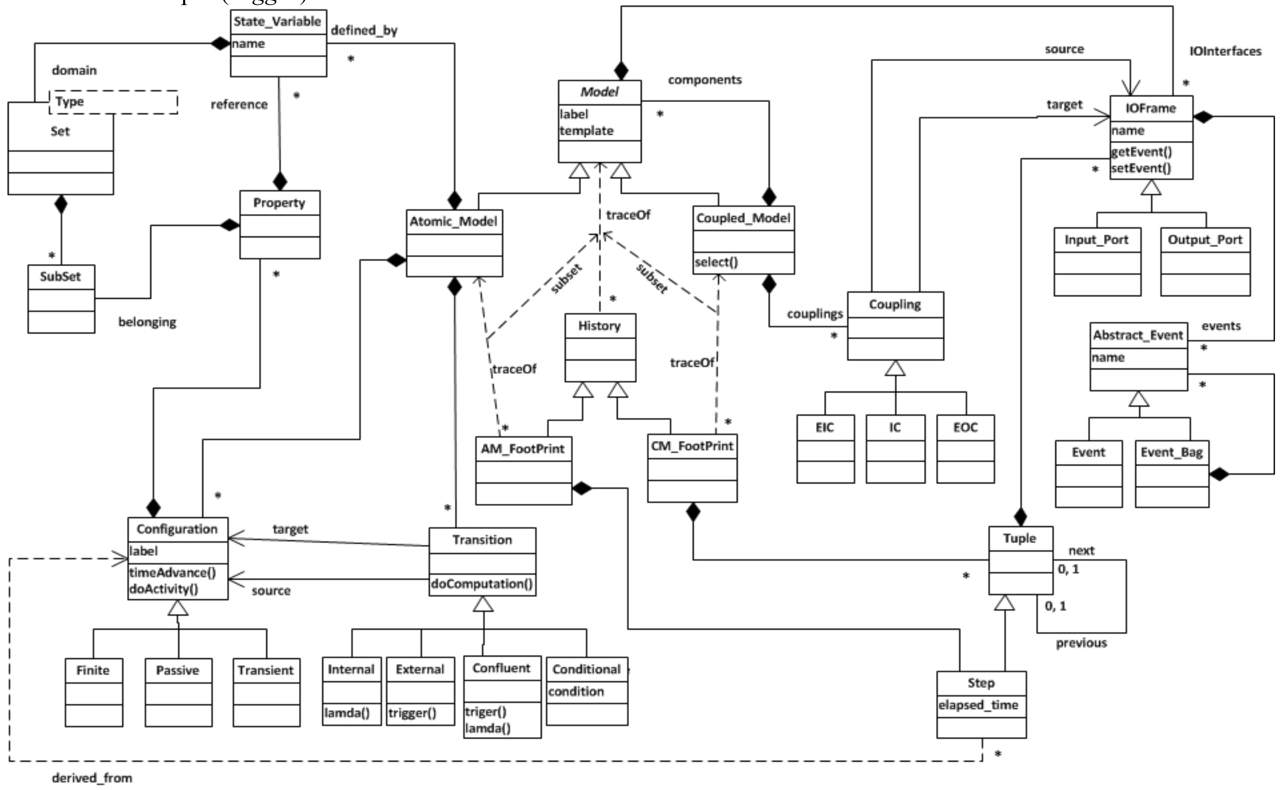

Figure 2. DDML abstract syntax

B. DDML concrete syntax

The metamodel of the concrete syntax associated to the notions is given in Figure 3. *Models* (Atomic & Coupled) are represented like UML classes and as such they possess properties like having attributes (including complex attributes realized by composition, association, and aggregation), functions (or subroutines), and being subject to inheritance. Models have input and output ports indicated with arrows as shown. These ports are labeled with their name and type (name: type). The last compartment is used to draw either the state transitions chart for an atomic model or the coupling network for a coupled model. In realizing these transitions, some function calls might be made to some functions defined in the third compartment.

The notation for *Configuration* is a box with four compartments for label, properties, activities, and time advance. For *External* transition, we use a broken arrow and this transition must appear before the edge of the box (top or bottom) and directed towards the lateral sides. There are also labels for the input event causing the transition and computation (re-configuration of state variables) at the end of the transition. For *Internal* transition, we use a straight line with arrow and this must appear at the right edge of the box and directed towards the left edge of another box. There are labels for *lamda* (output event at the beginning of the transition) and computation at the end of the transition. Confluent transition originates from the top right corner and is directed towards the back. Some transitions may be directed towards different Configurations depending on some conditions. This transition is a conditional transition represented with a diamond with the condition specified within the diamond and arrows pointing to different Configurations.

The notation for EIC is a dashed line, IC is a continuous line, and EOC is a line-dot-dotted line (Figure 4). It is also important to label these components.

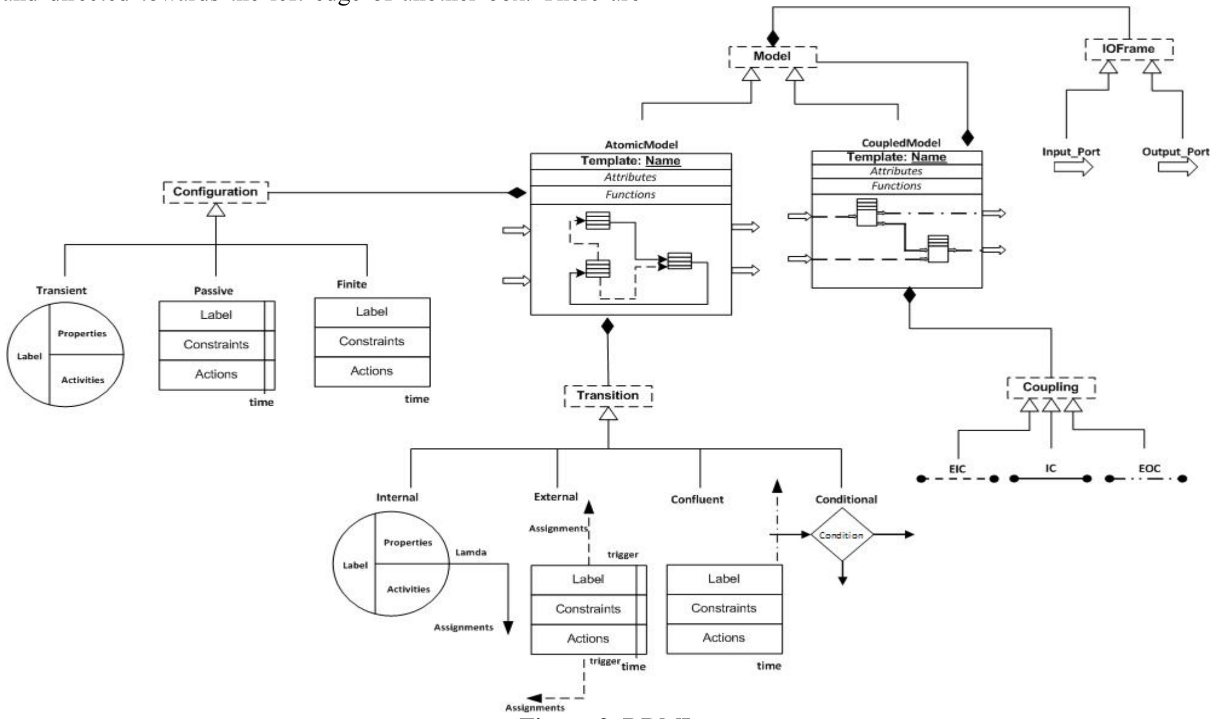

Figure 3. DDML concrete syntax

IV. APPLICATION TO URBAN TRAFFIC

We have applied our methodology and the use of DDML to the modeling and simulation of the urban traffic in Bamako, the capital city of Mali. We limited ourselves to the zone drawn on Figure 4 which is a very critical area, due to its importance in the transportation network of the city and to numerous traffic jams that occur there in a day. The zones marked with circled GEN are those from which users enters the area. The zones with circled ACC absorb the traffic.

It was commonly admitted that simulation is a powerful means for urban planning agents to understand the traffic in this area, to forecast various futures under various assumptions and to build strategies for developing the city's transportation system. But the urban traffic experts alone could not synthesize the simulation code which could serve for this purpose. We then applied our methodology.

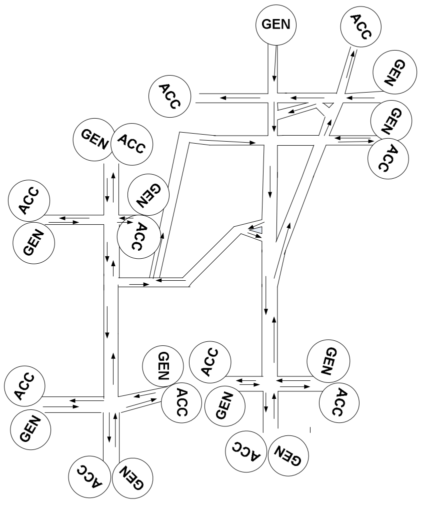

Figure 4. Area under study

The static structure of the traffic network obtained from domain analysis is shown in Figure 5. Blue classes are directly identified from domain experts' knowledge. These classes are refined into a technical class diagram (we do not present these details here) after thorough discussions with these experts.

Red classes are generic DDML models which relationships with blue classes (inheritance) are established by DEVS experts. Then, these experts can go into more specification using DDML and focusing on all dynamic entities recognized as atomic or coupled models.

Two such specifications are presented hereafter. We cannot be exhaustive in presenting all models of the entire specification. We only aim at showing how simulation concerns can easily be introduced with DDML, starting with the UML class diagram.

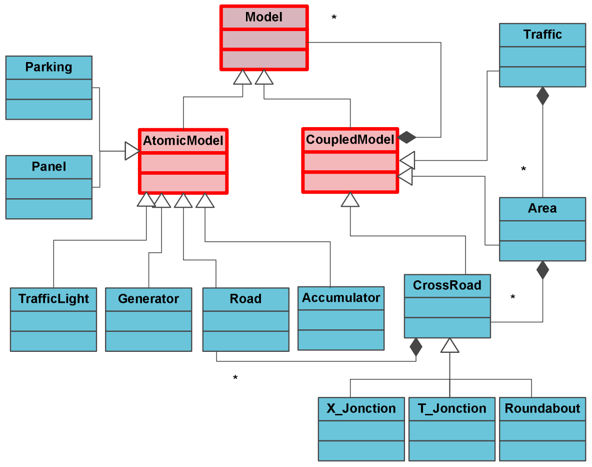

Figure 5. Domain knowledge of the urban traffic

A. *Traffic light*

A traffic light (Figure 6) successively display three colors (Green, Yellow, Red) when it is on. If the traffic light is off, the posted color is Black. The light can receive through *Control* port an order to turn on (1) or off (0). The *Signal* port stores the last color displayed by the traffic light (in the shape of icons). Configurations are: OFF, GO, READY-TO-STOP, STOP, tempOFF (for temporary state before going off) and tempON (for temporary state before going on). A single state variable called *Status* is used to capture these configurations. Being in any of them the display function is called. This function, as well as setters and getters are defined since the building of the UML technical class diagram.

In a steady situation, the traffic light successively goes from GO to READY-TO-STOP (after 5 time units), then to STOP (after 1 time unit), and then again to GO (after 3 time units). During the transition from GO to READY-TO-STOP the light sends Yellow in its *Signal* port; during the transition from READY-TO-STOP to STOP it sends Red; during the transition from STOP to GO it sends Green.

For any state different from OFF, when the light receives 0 in the *Control* port, then it operates an external transition to tempOFF (just for getting the opportunity to output Black), before operating an internal transition to the OFF passive configuration. If 1 is received by the light at any time in the OFF state (therefore the only constraint mentioned is that the elapsed time e in this state should be greater than 0), the latter operates an external transition to tempON (for getting the opportunity to output Green), before operating an internal transition to GO.

B. *Modeled area*

Crossroads as well as the whole area are coupled models. Crossroads are composed of traffic lights, roads, platforms (single places connecting roads), generators (of cars) and accumulators (to collect leaving cars). We split the whole area in five zones: Badalabougou, King Fad's Bridge, ENSUP, Dabanani market and Administrative city. Each zone is a coupled model and can receive and send cars to other zones under authorized circumstances (authorizations are also events sent between coupled components, e.g. a car can be sent to a zone if the latter is not already full; therefore the receiver should notify the sender of its current situation; the *Authin* and *Authout* ports are used for that purpose). The traffic model is shown in Figure 7.

The Eclipse-DDML tool [16] has been used to build the DDML models. This tool is integrated into the SimStudio platform which has a Java implementation of the DEVS simulation protocol [12]. The Eclipse-DDML converts the visual model obtained by drag-and-drop into an XML file. This latter one is translated into SimStudio classes by the means of an XSLT parser.

V. CONCLUSION

We presented in this work a methodology for the design of discrete events simulation models based on the DDML language. This methodology allows a hierarchical and modular construction of complex models starting from the UML domain class diagram, and then augmenting them with DEVS simulation concepts. While DEVS mathematical specifications are difficult to manipulate, this approach enables a visual design as well as a level of details which is closer to implementation. The resulting language is intended to provide a unified support for discrete event system modeling and simulation, by taking advantages of UML and by tailoring it specifically for the DEVS paradigm. This helps balance the design strength of UML for object modeling and expressiveness of the DEVS formalism for developing discrete event simulation models in a simple and intuitive way, and therefore improves communication between experts during the design stage. Moreover, supporting tools exist for DDML modeling and automated Java code synthesis.

One must notice that collaboration is constant throughout the DDML process between domain experts and simulation experts, but some tasks are mostly driven by one or the other group. Another remark is that we have intuitively shown here the mapping of the semantic domain of DDML into DEVS. A more formal work has been done about this mapping, which we cannot exhibit in this paper due to the lack of space.

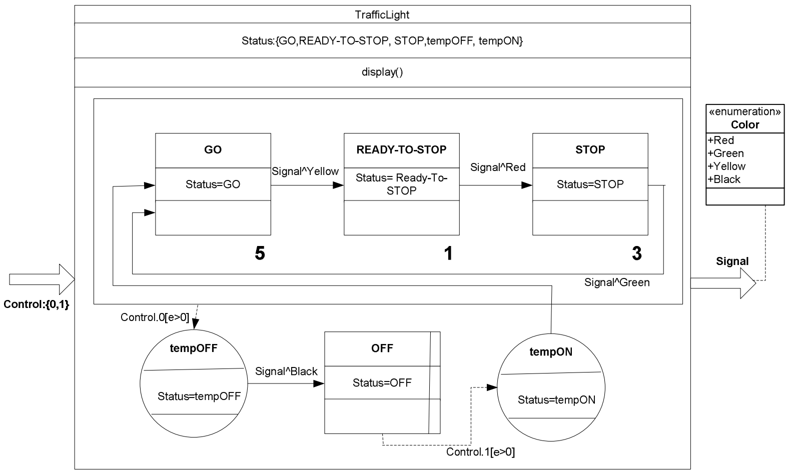

Figure 6. Traffic light atomic model

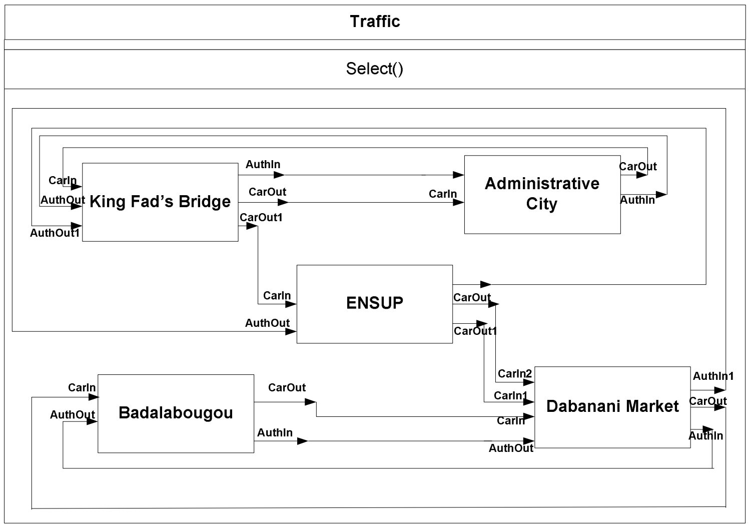

Figure 7. Traffic coupled model

References

[1] OMG, UML. http://www.omg.org/spec/UML/2.3/

[2] OMG, System Modeling Language TM (SysML) Version 1.2, June 2010. http://www.omg.org/spec/SysML/1.2/

[3] P. Chen and P. Pin-Shan, The Entity-Relationship Model – Toward a Unified View of Data, in: ACM Transactions on Database Systems Volume 1 Issue 1 (March 1976) 9-36.

[4] B. Zeigler, H. Praehofer, T. Kim, Theory of Modeling and Simulation 2nd Edition, Academic Press (2000).

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

[6] J.L. Risco-Martin, J.M. De La Cruz, S. Mittal, and B.P. Zeigler, eUDEVS: Executable UML with DEVS, Simulation Volume 85 Issue 11-12 (November 2009) 750-777.

[7] C.H. Sung and T.G. Kim, Object-Oriented Co-Modeling Methodology for Development of Domain Specific Models, G.A. Wainer and P.J. Mosterman (Eds.), Discrete-Event Modeling and Simulation: Theory and Applications, CRC Press (2010).

[8] B. Matias, G. Wainer, and R. Castro, Advanced IDE for Modeling and Simulation of Discrete Event Systems, in: Proc. Symposium on Theory of Modeling and Simulation, (SCS Spring Simulation Multiconference, Orlando, FL. 2010) #125.

[9] E. Pagliero, M. Lapadula, and E. Kofman, Power-DEVS: An Integrated Tool for Discrete Event Simulation (in Spanish), in: Proc. RPIC (San Nicolas, Argentina, 2003).

[10] H.S. Song and T. G. Kim, DEVS Diagram Revised: A Structured Approach for DEVS Modeling, in: Proc. European Simulation Conference (Eurosis, Belgium, 2010) 94 – 101.

[11] B.I. Ufuoma., M.K. Traoré. A Graphical Editor for the DEVS Driven Modeling Language. Proceedings of The ESM 2011.

[12] M.K. Traoré, SimStudio: a Next Generation Modeling and Simulation Framework, in Proc. ACM/IEEE 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (Marseille, France, March 3-7 2008) #67.

[13] H. Vangheluwe, DEVS as a Common Denominator for Multi-Formalism Hybrid Systems Modeling, in: Proc. IEEE International Symposium on Computer-Aided Control System Design (IEEE Computer Society Press, Alaska, 2000) 129-134.