AN ECLIPSE-BASED GRAPHICAL MODELING TOOL FOR DISCRETE EVENT SIMULATION

A THESIS SUBMITTED TO
AFRICAN UNIVERSITY OF SCIENCE AND TECHNOLOGY
ABUJA-NIGERIA

IN PARTIAL FUFILMENT OF THE REQUIREMENT FOR

MASTER DEGREE IN COMPUTER SCIENCE

BY

UFUOMA BRIGHT IGHOROJE

SUPERVISOR:

PROF. MAMADOU KABA TRAORE

African University of Science and Technology
Abuja
Knowledge is Freedom

DECEMBER 2010

To my sweet mum

Acknowledgement

There are many people who have helped me directly or indirectly with this thesis work. I am particularly grateful to my supervisor, Prof. Mamadou Kaba Traore for introducing me to this subject and many instructions on this subject. I appreciate his guidance, encouragement, and support during the period I worked on this thesis.

I owe deep gratitude to Dr. Boubou Cisse and Prof. Charles Chidume for their invaluable assistance, especially for their support during my visit to the ISIMA laboratory in Clermont-Ferrand for intensive research on this work. I would also like to thank Nafisa Abdullahi and Tracey Odigie for their logistical support. Thanks also to Dr. Ekpe Okorafor and Dr. Guy Degla for their advice and assistance.

My colleagues at African University of Science and Technology have been very wonderful. Special thanks to Doyin for several discussions on my thesis work and for her excellent suggestions. Thanks also to Henry, Toyin, Omena, Joachim, Osayawe, Aliu, Aminu, Dorothy, Dwumfour, Emmanuel, Rosine, Gaba, Meredith, Hans, and Kehinde.

I would also like to thank my dear friend, Emamurho for maintaining my mental well-being with numerous encouragement and talks. Thanks also to Emmanuel Osahon, Eric, Stanley, Esiwo, and Alex.

My great family has been very supportive. Thanks to my dad, mum, and big brother (Maroh) for their financial assistance, prayers and encouragement. Thanks to my lovely sisters, Precious and Best, their names describe what they have been to me. Thanks also to Wisdom for his wise thoughts and support.

My greatest thanks go to the Almighty God, to whom I owe my life, for His benevolence, mercy, and love.

This thesis work was funded by the African University of Science and Technology.

Abstract

We propose DEVS-Driven Modeling Language (DDML), a graphical notation for DEVS [1] 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. This also makes the editor extensible.

Table of Contents

List of Figures

Chapter 1. Introduction

1.1. Motivation

To model is to abstract from reality a description of a dynamic system. Modeling serves as a language for describing systems at some level of abstraction, or additionally, at multiple levels of abstraction. Modeling is a way of thinking and reasoning about systems.

Several methodologies and tools have been developed for modeling of dynamic systems. Most of these tools require the modeler to be an expert in programming and/or mathematics. Verifying models built with these tools with domain experts is difficult because a wide knowledge gap exists between the domain expert and the expert modeler. Also, modeling and simulation activities are wide apart. Modeling involves developing multiple levels of abstractions of a system and capturing these abstractions with algorithms that represent the static, dynamic and functional aspects of the system under study. The primary issues in simulation are timing aspects identification and time management. A generic approach is recommended to integrate advanced modeling into generic simulation methodologies. Hence, both simulation and software engineering domain expertise should be integrated in the modeling and simulation process. 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 and this must be inherently coherent. This integrative approach should allow the use of a user-familiar language/notation with a potential for formal specification of data and operations, and therefore the simulation system.

We define the DEVS-Driven Modeling Language (DDML) to be such a notation. DDML presents a graphical notation to effectively realize DEVS models. 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). DDML provides graphical notations for defining coupled models and atomic models. Furthermore, atomic models can be defined using state transition charts (internal and external state transitions), while taking cognizance of simulation timing.

To facilitate defining simulation models using DDML, we leverage Eclipse's very rich infrastructure to develop a graphical editing tool that realizes the graphical notations 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 while simplifying software development, installation and updates.

1.2. Objective

The objective of this work is to present the DEVS Driven Modeling Language (DDML), a graphical notation for modeling dynamic systems based on DEVS and an Eclipse-based graphical modeling tool based on DDML. The model, when defined in DDML using this tool would be amenable to formal analysis and automated code synthesis.

1.3. Structure of the Work

In the next chapter, we review the theory of DEVS and several tools that have been used to define models based on DEVS. In chapter 2, we present the DEVS formalism. In chapter 3, we present the DEVS Driven Modeling Language (DDML) with an illustrative example and we show how its graphical notations relate to DEVS. In chapter 4, we present the eclipse platform and discuss the tools that eclipse provides for building graphical editors. In chapter 5, we present the architecture of the Eclipse-based graphical editor for DDML and how it was built. In chapter 6, we present the Eclipse-DDML graphical editor and how to use it.

Chapter 2. Discrete Event System Specification (DEVS)

DEVS [1] is a modular and hierarchical formalism for modeling and analyzing general systems that can be discrete event systems. DEVS defines system behavior as well as system structure. System behavior in DEVS formalism is described using system input and output events as well as states.

According to the DEVS theory, the system of interest is seen as a source of behavioral data for study within a given experimental frame (EF). The EF is a restricted set of elements observed in the system and the conditions under which they are observed. These data are used to create an abstract representation of such a system (a model). Using a set of instructions, rules or mathematical equations, the model tries to replicate the behavior of the system of interest under experimental conditions.

A model represents a simplified version of reality and its structure. A model is built in consideration of the conditions of experimentation of the system of interest, including the work conditions of the real system and its application domain. The model is subsequently used to build a simulator (a device capable of executing the model's instructions) generating the model's behavior.

DEVS was created for modeling and simulation of discrete event dynamic systems (DEDS); thus it defined a way to specify systems whose states change either upon the reception of an input event or due to the expiration of a time delay. In order to attack the complexity of the system under study, the model is organized hierarchically (i.e. it is organized in such a way that every element is higher than its precedent), and the higher level components of the system are decomposed into simpler elements. The second tool used to attack complexity is information hiding, through the provision of a modular interface for each of the models.

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.

2.1. The DEVS Formalism

A real system modeled by DEVS can be described as a composition of atomic and coupled components. An atomic model is specified as:

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

Where

$X = \{(p, v) \mid p \in IPorts, v \in X_p\}$ is the set of *input* events, where *IPorts* represents the set of input ports and X_p represents the set of values for the input ports;

$Y = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ is the set of *output* events, where *OPorts* represents the set of output ports and Y_p represents the set of values for the output ports;

S is the set of sequential *states*;

$\delta_{\text{ext}}: Q \times X \rightarrow S$ is the *external* state transition function,

With $Q = \{(s, e)/s \in S, e \in [0, ta(s)]\}$ and *e* is the elapsed time since the last state transition;

$\delta_{\text{int}}: S \rightarrow S$ is the *internal* state transition function;

$\lambda: S \rightarrow Y$ is the *output* function; and

$ta: S \rightarrow R_0^+ \cup \infty$ is the *time advance* function.

At any given moment, a DEVS model is in a state $s \in S$. In the absence of external events, it remains in that state for a lifetime defined by $ta(s)$. When $ta(s)$ expires, the model outputs the value $\lambda(s)$ through a port $y \in Y$, and it then changes to a new state given by $\delta_{int}(s)$. A transition that occurs due to the consumption of time indicated by $ta(s)$ is called an internal transition. On the other hand, an external transition occurs due to the reception of an external event. In this case, the external transition function determines the new state, given by $\delta_{ext}(s, e, x)$, where *s* is the current state, *e* is the time elapsed since the last transition, and $x \in X$ is the external event that has been received.

The time advance function can take any real value between 0 and ∞. A state for which $ta(s) = 0$ is called a ***transient or intermediate*** state (which will trigger an i*nstantaneous* internal transition). In contrast, if $ta(s) = \infty$, then *s* is said to be a ***passive or infinite*** state, in which the system will remain perpetually unless an external event is received (can be used as a termination condition).

A DEVS *coupled model* is composed of several atomic or coupled sub-models. It is formally defined by

$$\mathbf{CM = <X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC, select>}$$

Where

X and Y are defined as previously;

D is the set of the component names and for each $d \in D$;

M_d is a DEVS (i.e., atomic or coupled) model;

EIC is the set of external input couplings (i.e., how the inputs of the coupled model are linked to the inputs of its sub-components)

EOC is the set of external output couplings (i.e., how the outputs of the coupled model are linked to the outputs of its sub-components)

IC is the set of internal couplings (i.e., how the outputs of any the sub-components are linked to the inputs of other sub-components), and

Select: $2^D \rightarrow D$ is the tiebreaker function

Coupled models group several DEVS into a composite model that can be regarded, due to the ***closure under coupling property***, as a new DEVS model. This closure property guarantees that the coupling of several class instances results in a model of the same class, allowing hierarchical construction.

Because multiple subcomponents can be scheduled for an internal transition at the same time, ambiguity could arise because it is not clear which transition this second component should execute first. There are two alternatives for this:

- To execute the external transition first and then the internal transition, with $e = ta(s)$;
- To execute the internal transition first, followed by the external transition, with $e = 0$.

The *select* function provides a simple way to solve this ambiguity. The function defines an ordering over all the components of the coupled model so that only the first model to execute in the case of simultaneous internal events can be chosen.

2.1.1. Parallel DEVS

In the previous section, we saw that whenever two models are scheduled for state transitions at the same time, a DEVS coupled model will pick the one specified by the *select* function to execute first. This tie-breaking strategy is rigid. The *select* function introduces serialization in the execution of components when many interconnected atomic models are imminent (which could be considered as parallel in a multiprocessor environment).

Parallel DEVS (or PDEVS) is a variant to DEVS that provides a more flexible way of dealing with these ambiguities. Atomic models provide an additional *confluent* function to specify collision behavior for events that might be scheduled simultaneously and a mechanism for receiving multiple external events at the same time and processing them together. An atomic PDEVS model is defined as

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

Where

X, Y and S are defined as previously;

$\boldsymbol{\delta_{ext}: Q \times X^b} \rightarrow S$ is the external transition function;

$\delta_{int}: S \rightarrow S$ is the *internal* state transition function;

$\boldsymbol{\delta_{con}: S \times X^b} \rightarrow S$ is the confluent transition function;

$\lambda: S \rightarrow Y^b$ is the *output* function; and

$ta: S \rightarrow R_0^+ \cup \infty$ is the *time advance* function.

PDEVS models use bags (multisets) of events for receiving inputs and collecting outputs (X^b *and* Y^b) instead of a single event. This allows multiple events to be processed simultaneously. Because external input events received by the component are added to the bag, external transition functions can combine the functionality of a number of external transitions into a single one, and simultaneous events (like the departure of a vehicle and a collision in the intersection) can be treated simultaneously. Also, PDEVS allows a better way to deal with collisions: the model specification includes a confluent transition function (δ_{con}). When a collision between the internal and external functions occurs, the confluent function determines the new state of the model.

The semantics of PDEVS for internal/external transition functions is similar to DEVS. If one or more external events $X^b = \{x_1 \ldots x_n / x_i \in X\}$ occur before $ta(s)$ expires (i.e., while the system is in total state (s, e) with $e < ta(s)$), the new state will be given by the model's external transition function, $\delta_{ext}(s, e, X^b)$. If the external events X^b are received when $e = ta(s)$, the new state of the model will be given by the confluent function (δ_{con}). If multiple components in a coupled model are imminent, all their outputs are first collected and mapped to their influences in parallel. Then the corresponding transition function is executed for every model.

In PDEVS, coupled models are defined as in DEVS, without the need for a *select* function. Formally, a coupled model is defined as

$$\mathbf{CM = \langle X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC \rangle}$$

Where definitions for the set of input and output events (X and Y), components (D and M_d), and couplings (EIC, EOC, and IC) follow the specifications of DEVS coupled models presented earlier in this chapter.

2.1.2. DEVS simulation algorithms

DEVS simulators are based on the abstract simulation techniques presented in Zeigler, Praehofer, and Kim [1]. These simulation algorithms are guaranteed to execute the hierarchical DEVS specifications correctly. It has been proven that these algorithms are correct to simulate DEVS models. This includes cases of hierarchical composition, individual atomic model execution, and detection of termination conditions (when all the models in the simulation are passive, the simulation can end).

The main idea of DEVS abstract simulation algorithms is to create a hierarchy of execution engines based on the modeling hierarchy created by the user. We call these entities *Processors*. Atomic/Coupled *Models* define the structure and behaviour of the system of interest, while their corresponding *Processors* implement the simulation dynamics (using an abstract mechanism hidden from the models), as sketched in Figure 1. This figure shows the different kinds of Processors: *Simulators* are associated with atomic models and *Coordinators* with coupled models. The *Root Coordinator* drives the global aspects of the simulation; it maintains the global time, starts/finishes the simulation (when a termination condition is detected), and is related to the Coordinator of the top-level coupled model (collecting the outputs from it and feeding it with external input events).

Simulation is driven by passing messages among the Processors; each represents an event to process. The messages include information about the event origin/destination, the time of the event the message represents, and its content. Four kinds of messages are used:

- * messages signal the occurrence of internal events
- X messages carry information about external input events
- Y messages transmit the model's output events
- Done messages carry scheduling information for future events, including that a model has finished with its current task.

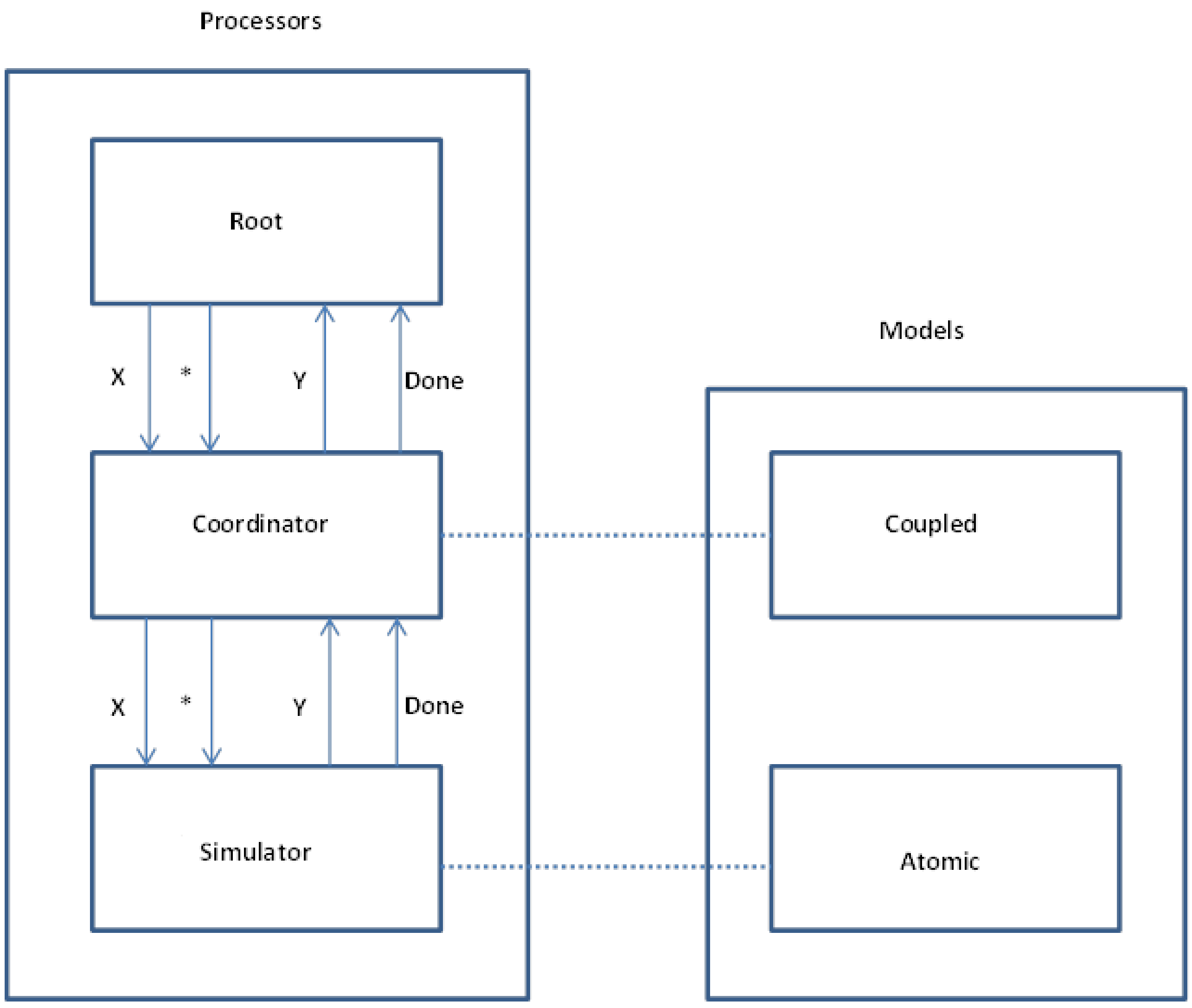


Figure 1. *Relationship between models and processors.*

As discussed earlier, PDEVS was introduced to solve serialization problems with the simultaneous events in classic DEVS. The main difference is that PDEVS processes input bags and generates output bags for the model, and the confluent transition function (δ_{con}) is activated when internal and external events occur simultaneously.

As with the original definition of the abstract simulator, PDEVS processors are specialized into two different engines, Simulator and Coordinator. Five kinds of messages are used and can be categorized into synchronization messages (@, *, and done) and content messages (y and q) (Figure 2):

- *Synchronization messages* are sent from a parent *Processor* to its imminent children. All imminent models' output functions must be executed before any transition function. All outputs are collected and only after they have been sorted the transition functions can be activated. Message @ is used to request all imminent children to execute their output functions and to route the outputs to the corresponding inputs according to the coupling scheme. Message * tells the children to invoke their transition functions (whether it is an internal, external, or confluent transition).
- *Data messages* are sent from parent/child *Processors.* All outputs produced by a model are translated to *y* messages between a child *Processor* and its parent. External messages are sent as *q* messages.

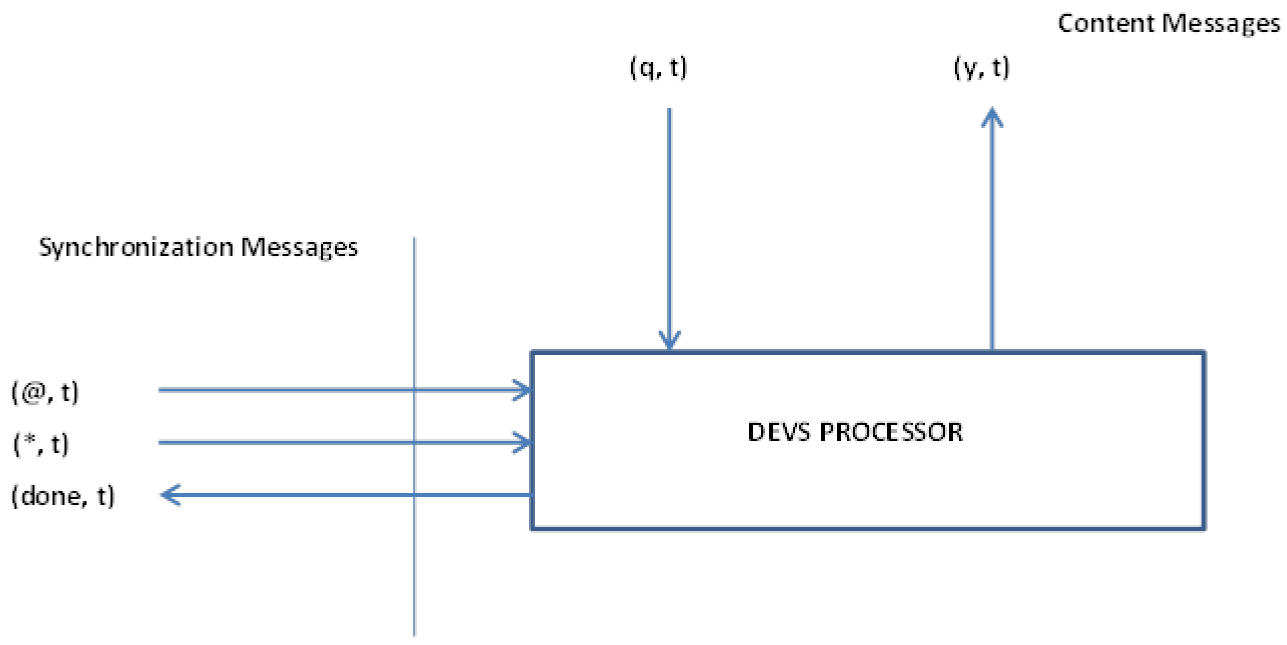


Figure 2. Messages that a DEVS processor sends and receives

For details of the simulation algorithms for DEVS (simulator, coordinator and root) and for PDEVS and efficient realizations of these algorithms, refer to [2].

2.2. Tools for DEVS

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.

The following is a list of some tools that have been developed (or currently undergoing development) for modeling and simulation based on the DEVS formalism.

- CD++ Builder [3][4][5] is an Eclipse plugin that integrates varied applications and utilities that aids in creating CD++ DEVS models, simulating and analysing results.

CD++ modeller 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. The CD++ tool has been developed following the specification of DEVS and Cell-DEVS.

- ADEVS [6] provides a C++ library based on DEVS, which developers can use to build their own models, and supports integration with other simulation environments.
- DEVS-Ada/TW was the first attempt to combine DEVS and the Time Warp parallel simulation algorithm over a multiprocessor environment. DOHS, the distributed optimistic hierarchical simulation scheme, combines DEVS and Time Warp, implemented in D-DEVSim++. This alternative presents a more general approach for distributed optimistic execution of DEVS models, while addressing some restrictions introduced in DEVS-Ada/TW [7].
- DEVS-C++ [8] is a DEVS-based modeling and simulation environment written in C++, which implements parallel execution and supports large-scale systems.
- DEVS-Scheme [9] [10] is a knowledge-based environment for modelling and simulation based on the Scheme functional language (a variation of Lisp).
- DEVS/HLA [11] [12] is based on the high-level architecture (HLA) [13]. It was used to demonstrate how an HLA-compliant DEVS environment could improve the performance of large-scale distributed modeling and simulation.
- DEVSJAVA [14] is a DEVS-based modeling and simulation environment written in Java. It provides classes for the users to implement their own DEVS models.
- DEVSim++ [15] is an object-oriented DEVS simulator implemented in C++. The tool defines basic classes that can be extended by users to define their own atomic and coupled DEVS components.
- GALATEA [16] is a simulation platform that offers a language to model multi-agent systems using an object-oriented architecture.
- JAMES [17] implements DEVS theory to model and simulate agent systems. The toolkit supports software-in-the-loop simulation to test agents in virtual environments.
- JDEVS [18] is a DEVS modeling and simulation environment written in Java. It allows general-purpose, component-based, object-oriented, visual simulation of models.
- PyDEVS [19] uses the ATOM3 tool [20] to construct DEVS models and to create the code to be executed. Models are represented as a state graph used to generate Python code and then interpreted by PyDEVS.
- SimBeams [21] is a component-based software architecture based on Java and JavaBeans. The idea is to provide a set of layered components that can be used in model creation, result output analysis, and visualization using DEVS.

The list of DEVS tools continues to grow. Common properties missing in these tools include:

- They are not amenable to formal analysis.
- Most methodologies are not highly communicable. It is difficult for domain experts to express their models using these advanced methodologies.
- They lack high expressive power.
- They fail to provide a framework for unification of C-DEVS and P-DEVS.
- Most of these tools lack a supporting tool that aids simulation modeling.

We develop DDML and our supporting tool to address these issues. The work in this project is based on the DDML (DEVS Driven Modeling Language) graphical notation. DDML has been pre-proposed. In the following chapter, we present DDML and show how easy it is to build a DEVS model with DDML. We also show how it maps to the DEVS formalism.

Chapter 3. DEVS Driven Modelling Language (DDML)

As we have seen in chapter 2, DEVS formalism is very generic and several DEVS tools and techniques adopt this mathematical style making it DEVS inaccessible to a very wide community. 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.Since DDML is visual, it is easier to discuss, understand, modify, and diagnose problems. Graph algorithms can be applied to a DDML model to evaluate the model and resolve problems.

3.1. 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 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 3 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.

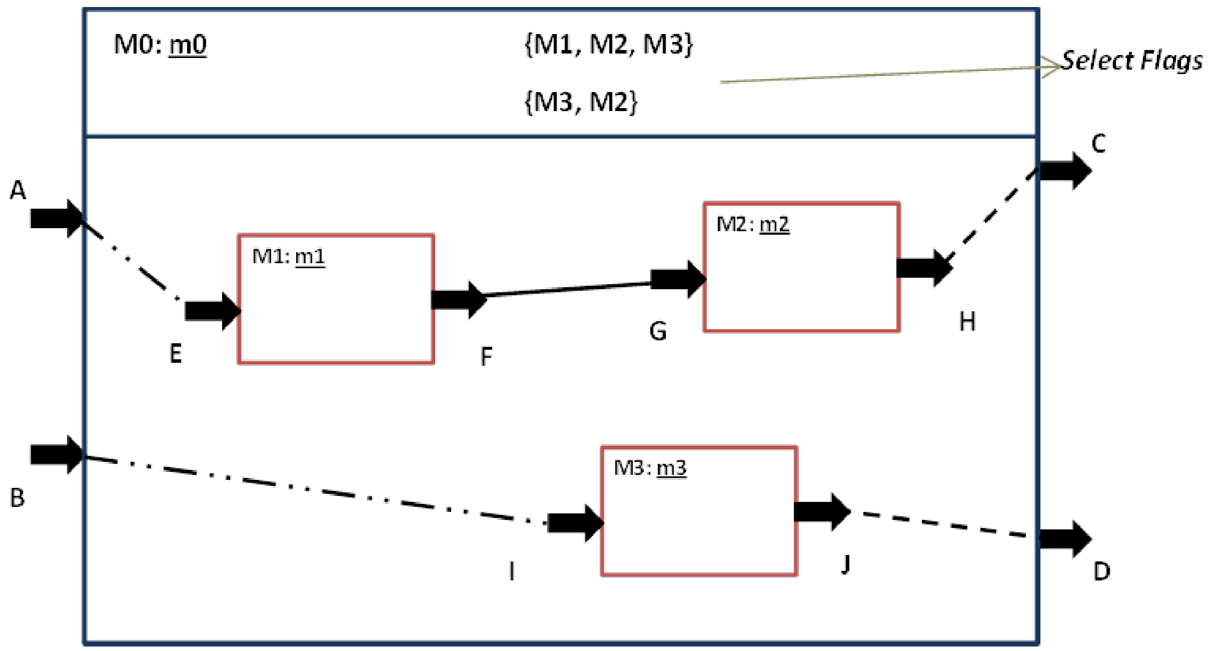


Figure 3. DDML Coupled Model and Atomic Model

3.1.1. 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 ∈ 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 3) 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.

3.1.2. Relation to Parallel DEVS Theory

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

$$\mathbf{CM = <X, Y, D, \{Md \mid d \in D\}, EIC, EOC, IC>}$$

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).

3.2. 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*. From set theory, we can show that a partition on a given set of states is defined under a given relation, and each subset in the partition is 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 4). 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 4 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 latter case, sub-states have their own durations). We call this a **state cluster**. Figure 4 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.

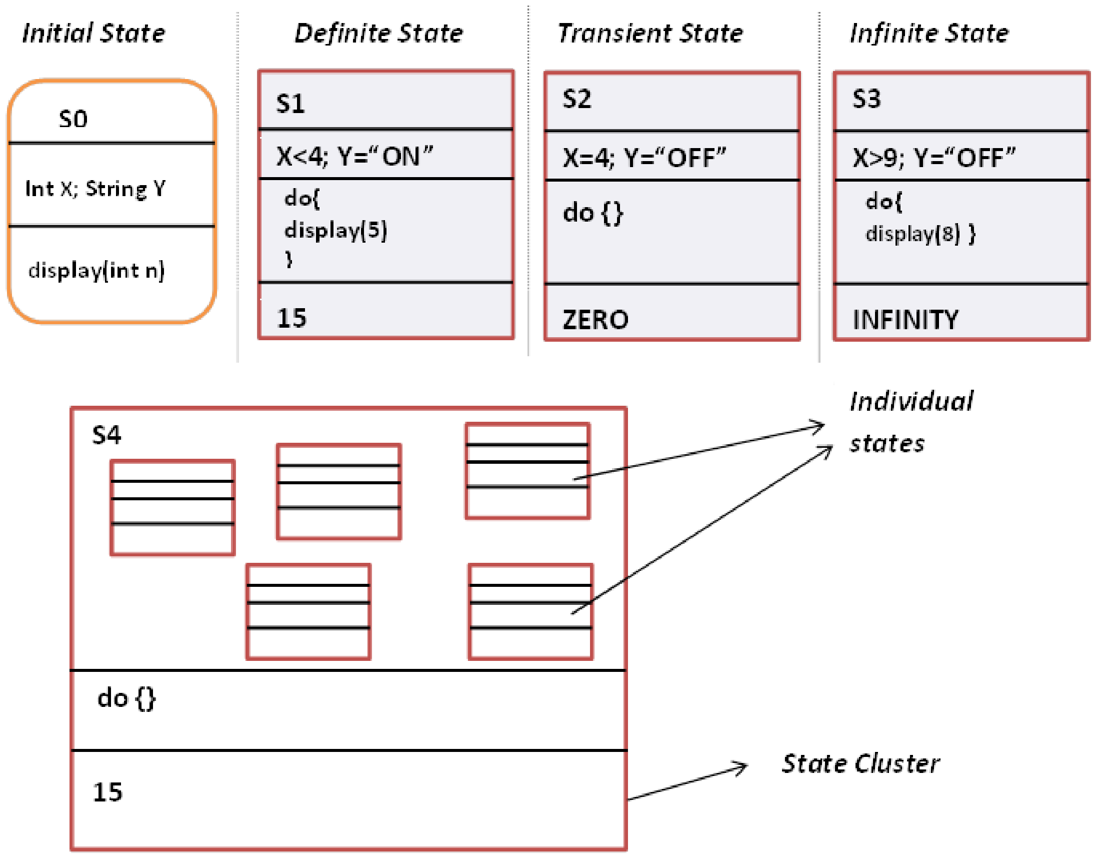


Figure 4. State Notations in DDML

The **internal state transition** is represented by a solid line with an arrow at the end as shown in Figure 5 (**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.

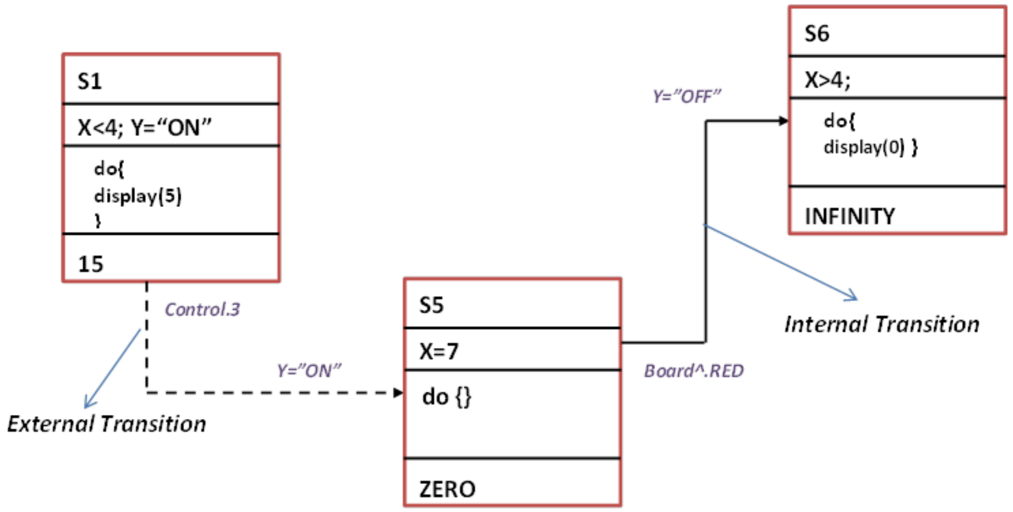


Figure 5. External and Internal State Transitions

The **external state transition** is represented by a broken line with an arrow at the end as shown in Figure 5 (**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 6. 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 6. A conflict transition also has an action and computation.

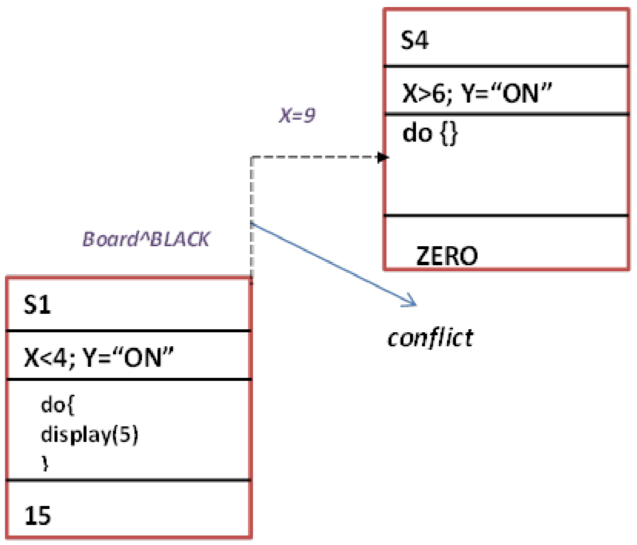


Figure 6. Confluent Transition

DDML also has notation to define a conditional transition. The diamond shaped figure (Figure 7) 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 $Y \neq 5$ or transits to state **B** if Y == 5. Conditional transitions could also apply to external state transitions.

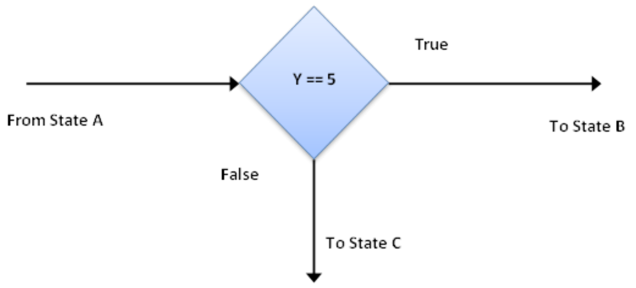


Figure 7. Conditional Transition

3.2.1. Relation to Classical DEVS Theory

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

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

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 3.1.1.

- 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 (ta) is indicated for each state at the bottom of the corresponding rectangle. Transient states are states with ta(s) = 0 and infinite states are states with ta(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.

3.2.2. Relation to Parallel DEVS Theory

An atomic model is defined in P-DEVS as: $\mathbf{M = <X^b, Y^b, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta>}$

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

- Inputs (and outputs) are synchronized.
- Each relation δ_{con} defines in the conflict transition (Figure 6), which carries X and $\boldsymbol{\lambda}$ (s).

3.3. Modeling a Traffic Light System Using DDML

In this section, we shall demonstrate how to model dynamic systems using DDML with the example of a traffic light system.

Figure 8 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.

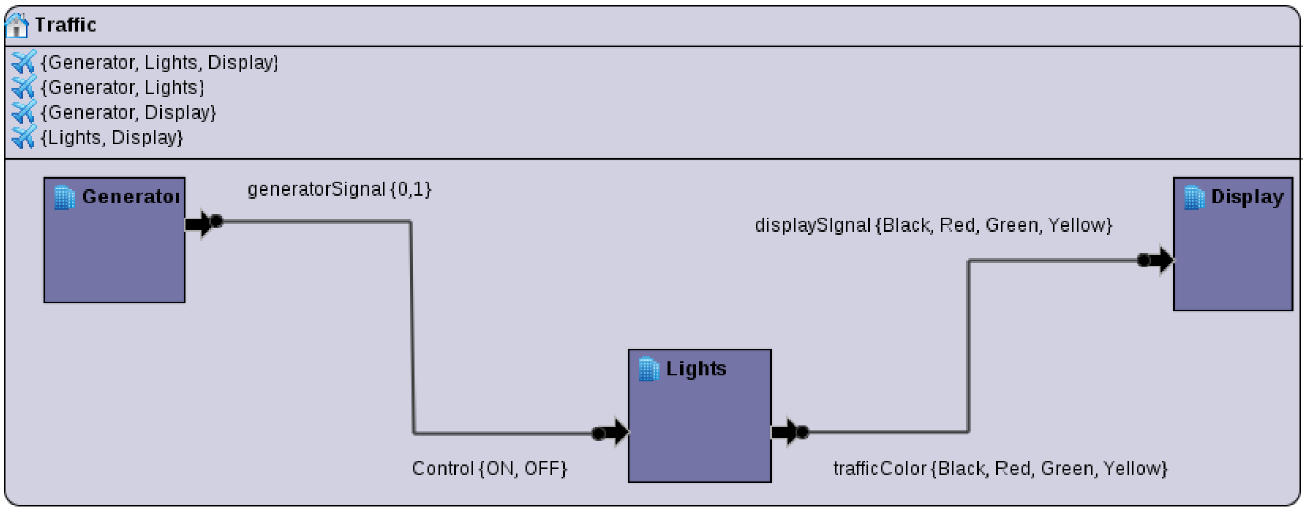


Figure 8. DDML Model for Traffic System

The Generator randomly switched ON/OFF the 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.

The Generator can be further analyzed using DDML state notations and state transition notations. Figure 9 shows a description of the Generator process.

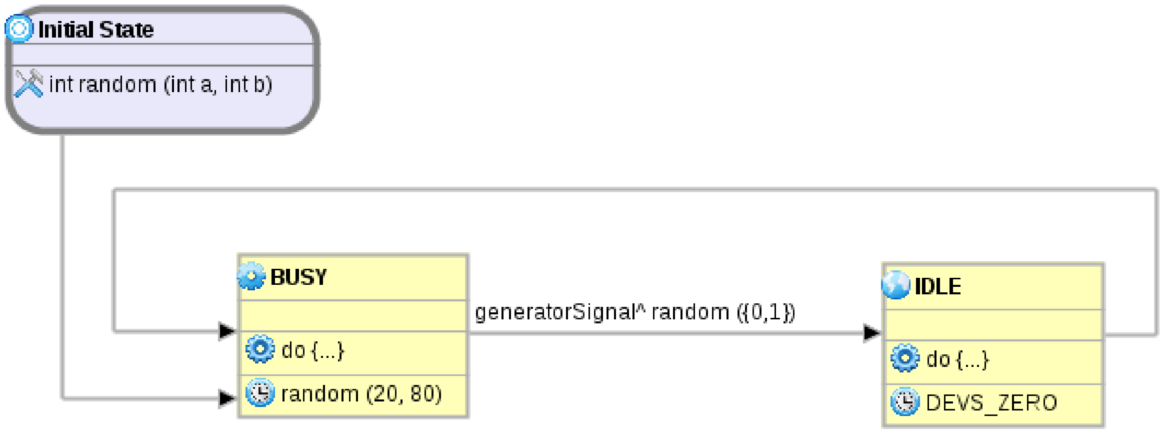


Figure 9. DDML model for the Generator Process showing states and transitions

The Generator system randomly switches OFF/ON the traffic system. From the Figure above, we can see that the generator has two states; the **BUSY** and **IDLE** states. The Generator generates a random signal (0 or 1) at the end of its **BUSY** state, which lasts for a random time between 20 and 80 seconds. A signal of 0 switches off the traffic system; whereas a signal of 1 indicated that the traffic system is ON. Note that IDLE process is a transient process (time advance is zero).

Figure 10 below shows the DDML state graph for the Light system.

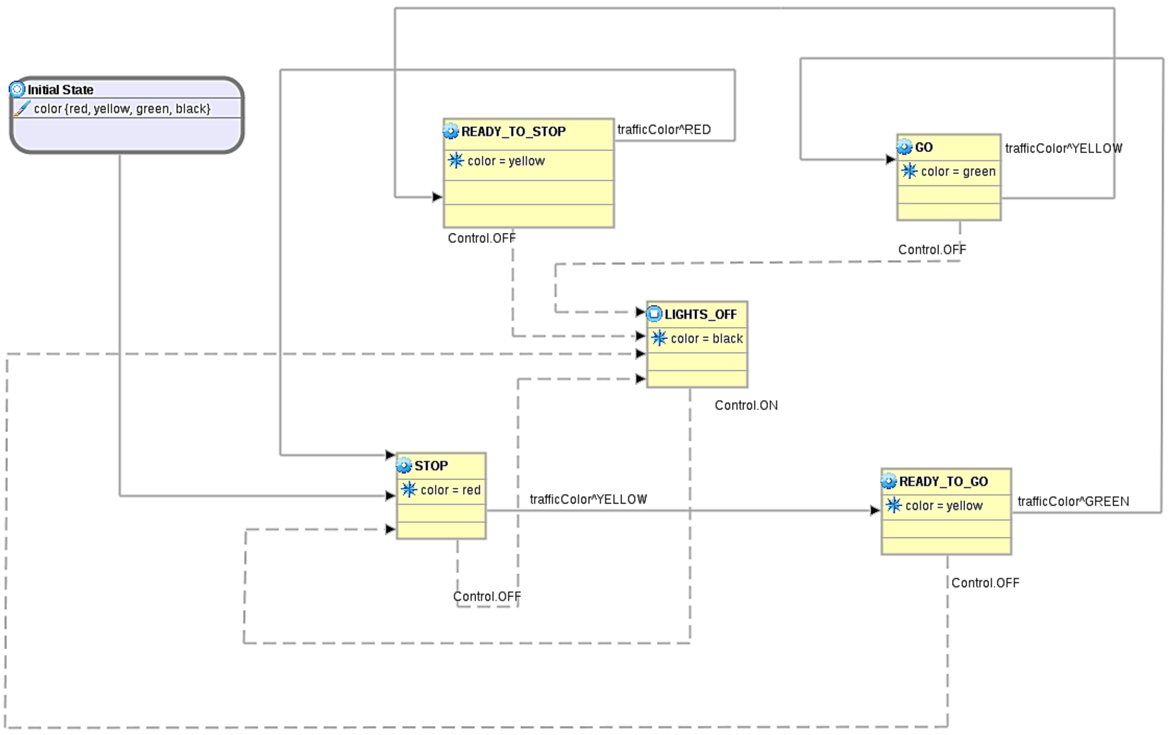


Figure 10. DDML Model for Lights Process

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 values for 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.

The DDML state graph for the Display system is shown in Figure 11 below

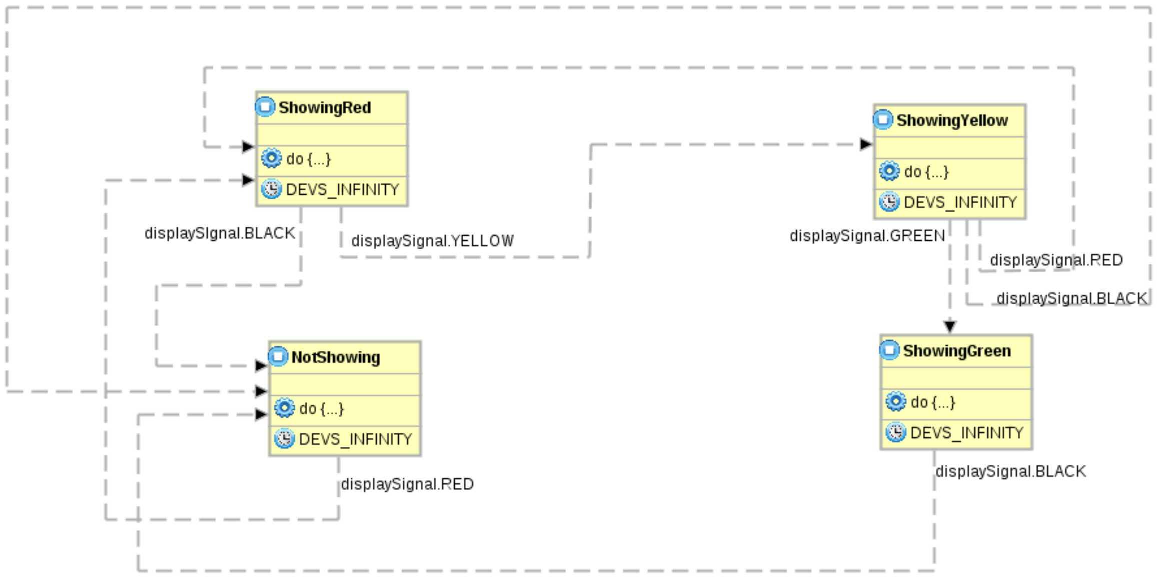


Figure 11. DDML State Graph for the Display Process

Chapter 4. Building Graphical Editors with Eclipse

In order to implement the graphical editor for DDML, several frameworks have been considered. Some of them include the basic Java graphic libraries: the Abstract Window Toolkit (AWT) and Swing. Others frameworks considered are the Eclipse graphic utilities: the Standard Widget Toolkit (SWT), JFace, Draw2D, and the Graphical Editing Framework (GEF). AWT, Swing, and SWT are based on Java and they provide general GUI controls useful for building form windows. Furthermore, SWT is a widget set and graphics library integrated with the native window system but with an OS-independent API. JFace is a platform-independent toolkit built on SWT. It provides convenience classes for many typical application features and simplifies a number of common UI tasks. It enhances and works with SWT without ever hiding it from the developer. Nevertheless, AWT, SWT, Swing, and JFace 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 of Draw2D that builds on top of the SWT library. The Draw2D tool lets you render GUI components with whatever appearance and functionality you prefer. It provides this capability by creating a high-level drawing region that operates independently from the native platform. GEF allows generating a graphical editor based on an existing application model. Eclipse provides Eclipse Modelling Framework (EMF) for building application models and Graphical Modeling Framework (GMF) to glue EMF models with GEF. Due to these reasons, Eclipse's GMF has been chosen because this library acts as a bridge between GEF and EMF; and it specifically tackles the creation of graphical Eclipse-based editors.

4.1. Overview of Eclipse

Eclipse [23] is an open source software development platform, simple to understand, yet robust enough to support integration with a lot of powerful tools. The Eclipse SDK comes with a Java Development Toolkit (JDT) for writing and debugging Java programs and the Plugin Development Environment (PDE) for extending Eclipse. The Eclipse Platform's purpose is to provide the services necessary for integrating software development tools, which are implemented as plugins. To be useful, the Platform has to be extended with plugins; even the JDT is a plugin. The beauty of Eclipse's design is that, except for a small runtime kernel, everything is a plugin or a set of related plugins. This plugin design makes Eclipse extensible. More important, however, the platform provides a well-defined way for plugins to work together (by means of extension points and contributions), so new features can be added not only easily, but seamlessly.

Eclipse is built to meet the following requirements:

- Support the construction of a variety of tools for application development.
- Support an unrestricted set of tool providers, including independent software vendors (ISVs).
- Support tools to manipulate arbitrary content types (e.g., HTML, Java, C, JSP, EJB, XML, and GIF).
- Facilitate seamless integration of tools within and across different content types and tool providers.

- Support both GUI and non-GUI-based application development environments.
- Run on a wide range of operating systems, including Windows® and Linux.
- Capitalize on the popularity of the Java programming language for writing tools.

Eclipse platform architecture consists of a platform runtime kernel, Workbench, workspace, help, and team components. Other tools plug in to this basic framework to create a usable application.

4.2. Eclipse Plugin Environment

An Eclipse plugin is the smallest unit of Eclipse Platform function that can be developed and delivered separately. Usually a small tool is written as a single plugin, whereas a complex tool has its functionality split across several plugins. Except for a small kernel known as the Platform Runtime, all of the Eclipse Platform's functionality is located in plugins.

An Eclipse Plugin is coded in Java. A typical plugin consists of Java code in a JAR library, some read-only files, and other resources such as images, web templates, message catalogs, native code libraries, etc. Some plugins do not contain code at all. One such example is a plugin that contributes online help in the form of HTML pages. A single plugin's code libraries and read-only content are located together in a directory in the file system, or at a base URL on a server. There is also a mechanism that permits a plugin to be synthesized from several separate fragments, each in their own directory or URL. This is the mechanism used to deliver separate language packs for an internationalized plugin.

Each plugin has a manifest file declaring its interconnections to other plugins. The interconnection model is simple: a plugin declares any number of named extension points, and any number of extensions to one or more extension points in other plugins.

A plugin’s extension points can be extended by other plugins. For example, the workbench plugin declares an extension point for user preferences. Any plugin can contribute its own user preferences by defining extensions to this extension point. An extension point may have a corresponding API interface. Other plugins contribute implementations of this interface via extensions to this extension point. Any plugin is free to define new extension points and to provide new API for other plugins to use.

On start-up, the Platform Runtime discovers the set of available plugins, reads their manifest files, and builds an in-memory plugin registry. The Platform matches extension declarations by name with their corresponding extension point declarations. Any problems, such as extensions to missing extension points, are detected and logged. The resulting plugin registry is available via the Platform API. Plugins cannot be added after start-up.

Plugin manifest files contain XML. An extension point may declare additional specialized XML element types for use in the extensions. This allows the plug-in supplying the extension to communicate arbitrary information to the plugin declaring the corresponding extension point. Moreover, manifest information is available from the plugin registry without activating the contributing plugin or loading of any of its code. This property is key to supporting a large base of installed plugins only some of which are needed in any given user session. Until a plugin’s code is loaded, it has a negligible memory footprint and impact on start-up time. Using an XML-based plugin manifest also makes it easier to write tools that support plugin creation. The Plugin Development Environment (PDE), which is included in the Eclipse SDK, is such a tool.

A plugin is activated when its code actually needs to be run. Once activated, a plugin uses the plugin registry to discover and access the extensions contributed to its extension points. The contributing plugin will be activated when the user selects a preference from a list. Activating plugins in this manner does not happen automatically; there are a small number of API methods for explicitly activating plugins. Once activated, a plugin remains active until the Platform shuts down. Each plugin is furnished with a subdirectory in which to store plugin-specific data; this mechanism allows a plugin to carry over important state between runs.

The Platform Runtime declares a special extension point for applications. When an instance of the Platform is launched, the name of an application is specified via the command line; the only plugin that gets activated initially is the one that declares that application.

By determining the set of available plugins up front, and by supporting a significant exchange of information between plugins without having to activate any of them, the Platform can provide each plugin with a rich source of pertinent information about the context in which it is operating. This context cannot change while the Platform is running, so there is no need for complex life cycle events to inform plugins when the context changes. A lengthy start-up sequence is avoided, as is a common source of bugs stemming from unpredictable plugin activation order.

The Eclipse Platform is run by a single invocation of a standard Java virtual machine. Each plugin is assigned its own Java class loader that is solely responsible for loading its classes (and Java resource bundles). Each plugin explicitly declares its dependence on other plugins from which it expects to directly access classes. A plugin controls the visibility of the public classes and interfaces in its libraries. This information is declared in the plugin manifest file; the visibility rules are enforced at runtime by the plugin class loaders.

The plugin mechanism is used to partition the Eclipse Platform itself. Indeed, separate plugins provides the workspace, the workbench, and so on. Even the Platform Runtime itself has its own plugin. Non-GUI configurations of the Platform may simply omit the workbench plugin and the other plugins that depend on it.

The Eclipse Platform's update manager downloads and installs new features or upgraded versions of existing features (a feature being a group of related plugins that get installed and updated together). The update manager constructs a new configuration of available plugins to be used the next time the Eclipse Platform is launched. If the result of upgrading or installing proves unsatisfactory, the user can roll back to an earlier configuration.

The Eclipse Platform Runtime also provides a mechanism for extending objects dynamically. A class that implements an “adaptable” interface declares its instances open to third party behavior extensions. An adaptable instance can be queried for the adapter object that implements an interface or class. For example, workspace resources are adaptable objects; the workbench adds adapters that provide a suitable icon and text label for a resource. Any party can add behavior to existing types (both classes and interfaces) of adaptable objects by registering a suitable adapter factory with the Platform. Multiple parties can independently extend the same adaptable objects, each for a different purpose. When an adapter for a given interface is requested, the Platform identifies and invokes the appropriate factory to create it. The mechanism uses only the Java type of the adaptable object (it does not increase the adaptable object's memory footprint). Any plugin can exploit this mechanism to add behavior to existing adaptable objects, and to define new types of adaptable objects for other plugins to use and possibly extend.

4.3. Eclipse Modeling Framework (EMF)

EMF [24] is a Java framework and code generation facility for building tools and other applications based on a structured model. EMF helps to rapidly turn models into efficient, correct, and easily customizable Java code.

EMF model requires just a small subset of the kinds of things that can be modeled in UML, specifically simple definitions of the classes and their attributes and relations, for which a full-scale graphical modeling tool is unnecessary.

EMF unifies the three important technologies: Java, XML, and UML. Regardless of which one is used to define the model, an EMF model is the common high-level representation that "glues" them all together. The model used to represent models in EMF is called Ecore (an XML Metadata Interchange (XMI) file). Ecore is itself an EMF model, and thus is its own meta-model. While EMF uses XMI (XML Metadata Interchange) as its canonical form of a model definition, you have several ways of getting your model into that form:

- Create the XMI document directly (using Ecore) , using an XML or text editor
- Export the XMI document from UML using a modeling tool such as Rational Rose
- Annotate Java interfaces with model properties
- Use XML Schema Definition (XSD) to describe the form of a serialization of the model

Once an EMF model is specified, the EMF generator can create a corresponding set of Java implementation classes. These generated classes can be edited to add methods and instance variables and still regenerate from the model as needed: the additions will be preserved during the regeneration.

In addition to simply increasing productivity, building an application using EMF provides several other benefits like model change notification, persistence support including default XMI and schema-based XML serialization, a framework for model validation, and a very efficient reflective API for manipulating EMF objects generically. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

EMF consists of two fundamental frameworks: the core framework and EMF.Edit. The core framework provides basic generation and runtime support to create Java implementation classes for a model. EMF.Edit extends and builds on the core framework, adding support for generating adapter classes that enable viewing and command-based (undoable) editing of a model, and even a basic working model editor.

4.4. Graphical Editing Framework (GEF)

The Graphical Editing Framework, GEF [25] allows us to easily develop graphical representations for existing models. It is possible to develop feature rich graphical editors using GEF. All graphical visualization is done via the Draw2D framework, which is a standard 2D drawing framework based on SWT from eclipse.org. The editing possibilities of the Graphical Editing Framework allow you to build graphical editors for nearly every model. With these editors, it is possible to do simple modifications to a model, like changing element properties or complex operations like changing the structure of your model in different ways at the same time.

All these modifications to the model can be handled in a graphical editor using very common functions like drag and drop, copy and paste, and actions invoked from menus or toolbars. As GEF is based on MVC architecture, every GEF-based application uses a model to represent the state of the diagrams being created and edited. GEF allows us to use any objects as model objects within an application, however, using an EMF model provides some advantages over using arbitrary objects:

- EMF's code generation facilities can be used to produce consistent, efficient and easily customizable implementations of a model's objects. If your model evolves during development, you can regenerate the code to reflect changes to the model, while preserving your customizations.
- The MVC architecture used by GEF relies on controllers that listen for model changes and update the view in response. If you use an EMF model, notification of model change is already in place, as all EMF model objects notify change via EMF's notification framework.
- The implementations generated for the model's objects ensure that the model remains consistent; for example, when a reference is updated, the opposite reference is also updated.
- EMF provides support for persisting model instances, and the serialization format is easily customizable.
- Your applications can use the reflective API provided by EMF to work with any EMF model generically.

Although EMF.Edit-based editors can be generated from EMF models using the org.eclipse.emf.codegen.ecore plugin, these editors use JFace viewers, such as the TreeViewer to display model instances, and typically provide a view that has a one-to-one correspondence with the model. Sometimes editors where the view is more loosely coupled with the model might be created. This is often the case when we want to use a graphical notation that may hide some of the detail of the underlying model objects, or may impose additional or a different structure to the model, for visualization purposes. We can think about using GEF and EMF together from two different perspectives; using an EMF model within a GEF application, and augmenting EMF.Edit-based editors using GEF.

4.5. Graphical Modeling Framework (GMF)

GMF [26] combines both GEF and EMF for designing powerful graphical editors. GEF is used to define the figures representing the models created using EMF. GMF sits on GEF and utilizes all the features of EMF and also provides its users with an editor with many re-usable components.

GMF consists of two main components: a runtime and a tooling framework. The runtime handles the task of bridging EMF and GEF while providing a number of services and Application Programming Interfaces (API) to allow for the development of rich graphical editors. The tooling component provides a model-driven approach to defining graphical elements, diagram tooling, and mappings to a domain model for use in generating diagrams that leverage the runtime.

GMF also relies on the Model-View-Controller (MVC) architectural pattern to separate the model from its graphical representation, which has been used successfully in other DEVS editor (notably CD++ Builder).

4.5.1. Runtime Framework

The production of an editor plugin based on the generator model will target a final model; that is, the diagram runtime model. The runtime is bridge between the notation and domain model(s) when a user is working with a diagram. It also provides for the persistence and synchronization of both. The runtime not only allows easier integration between EMF and GEF, but provides additional services like: transactions support, extended meta-modeling facilities, notation meta-model, variability points used for runtime extensibility of generated code, etc.

4.5.2. Tooling Framework

Figure 12 below illustrates the main components and models used on the tooling side of GMF.

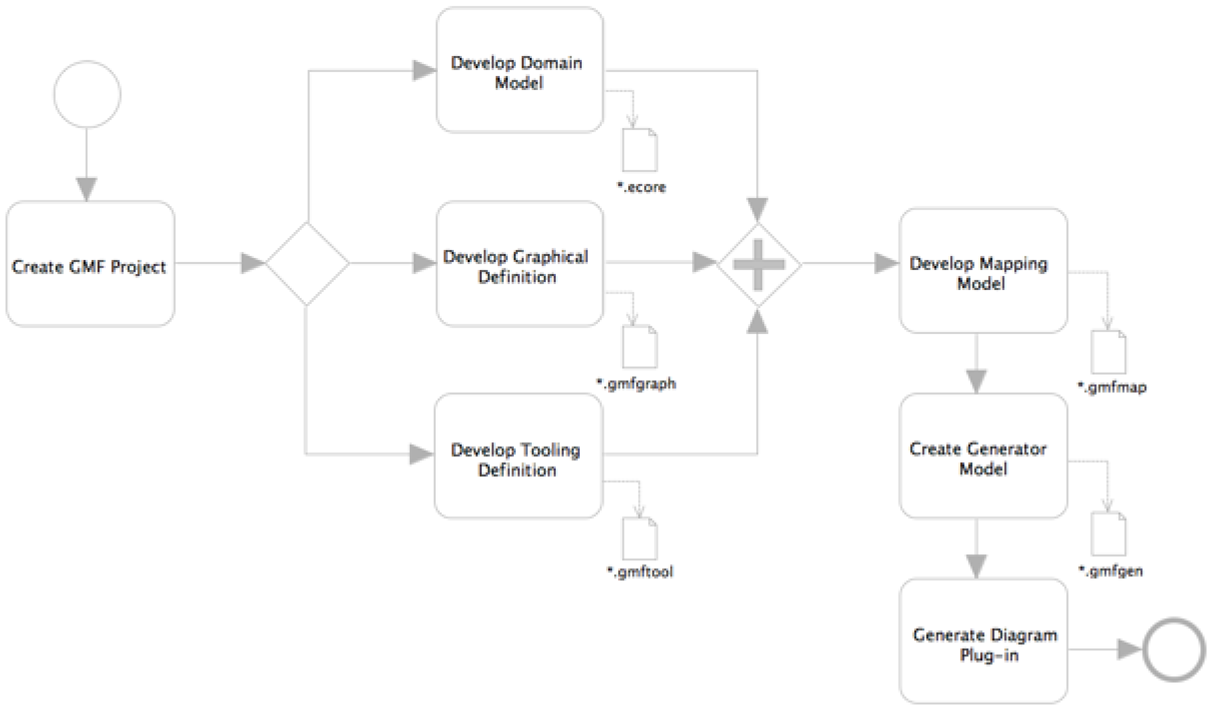


Figure 12. GMF Tooling Framework Main Components

4.5.3. Graphical Definition Model

The graphical definition model consists of two parts and defines the graphical elements found on a diagramming surface. The first part is a Figure Gallery, which defines figures (shapes, labels, lines, and so on) that the Canvas elements later reference to define nodes, connections, compartments, and diagram labels. An important point is that figure galleries can be reused. Many diagrams require similar-looking elements, such as a rounded rectangle with center label, or connections that are a solid line with open arrowhead decoration on the target end.

The mapping model references figures defined in the gmfgraph model. When the mapping model is transformed to the generator model, figure code is generated and included within the gmfgen model itself. When code is generated, edit parts will contain figures as inner classes. This is the default behavior when working with GMF, although it is not necessarily the recommended approach.

Another feature of the graphical definition model is the capability to export figures to a standalone figure plugin. This can also satisfy reuse because these plugins can be shared by several diagrams and among a community as a binary form of the figure gallery.

4.5.4. Tooling Definition Model

Diagrams typically include a palette and other periphery to create and work with diagram content. The purpose of the tooling definition model is to specify these elements. The tooling model currently includes elements for the palette, the toolbar, and various menus to be defined for a diagram.

4.5.5. Mapping Definition Model

Perhaps the most important of all models in GMF is the mapping model. Here, elements from the diagram definition (nodes and links) are mapped to the domain model and assigned tooling elements. The mapping model represents the actual diagram definition and is used to create a generator model. Typically a one-to-one mapping exists among a mapping model, its generator model, and a particular diagram.

The mapping model uses Object Constraint Language (OCL) in many ways, including initializing features for created elements, defining link and node constraints, and defining model audits and metrics. Audits identify problems in the structure or style of a diagram and its underlying domain model instance, and metrics provide measures of diagram and domain model elements.

4.5.6. Generator Model

As mentioned in the overview, the generator model adds information used to generate code from the mapping model and is somewhat analogous to the EMF genmodel. Both can be reproduced and reloaded from their source models, although the EMF genmodel is a true decorator model. The GMF generator model is more of a many-to-one model transformation than a decorator model. As a mapping model is transformed into a generator model, it loses knowledge of the graphical definition and gains knowledge of the runtime notation model. This minimizes the number of dependencies linked from the generator model and separates concern among the models. The code generated should be modified to provide custom behavior and functionality that is required of the graphical editing tool.

GMF comes with a dashboard that streamlines the workflow of dealing with the collection of models. Each model can be selected, created, or edited within the dashboard, including EMF *.ecore and *.genmodel models.

Chapter 5. Architecture of Eclipse-DDML Graphical Editor

Figure 13 shows the conceptual architecture of the Eclipse-based DDML Editor.

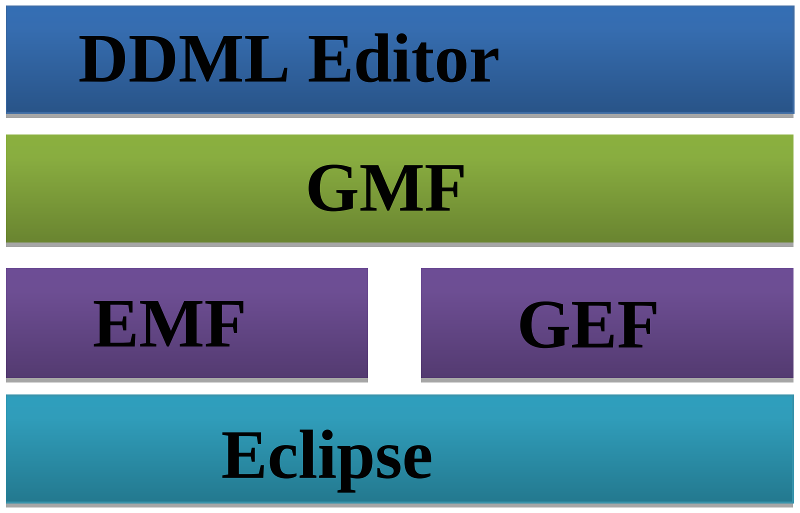


Figure 13. Architecture of Eclipse DDML Graphical Editor

EMF was used to define the model, as it provides several utilities for specifying entities. The model was specified in an Ecore graphical editor which was translated into an Ecore model (XMI format). EMF uses this model to generate Java classes and interfaces. The generated classes implement the observable design pattern, providing methods that notify whenever one of their properties has changed. Custom code and methods to provide extra behaviour and functionality was added to the generated classes. EMF recognises special code comments in the added or customized methods not to overwrite them when the model is regenerated. EMF also provides persistence and validation services for the generated models. Figure 14 below shows the Ecore Graphical Definition for DDML

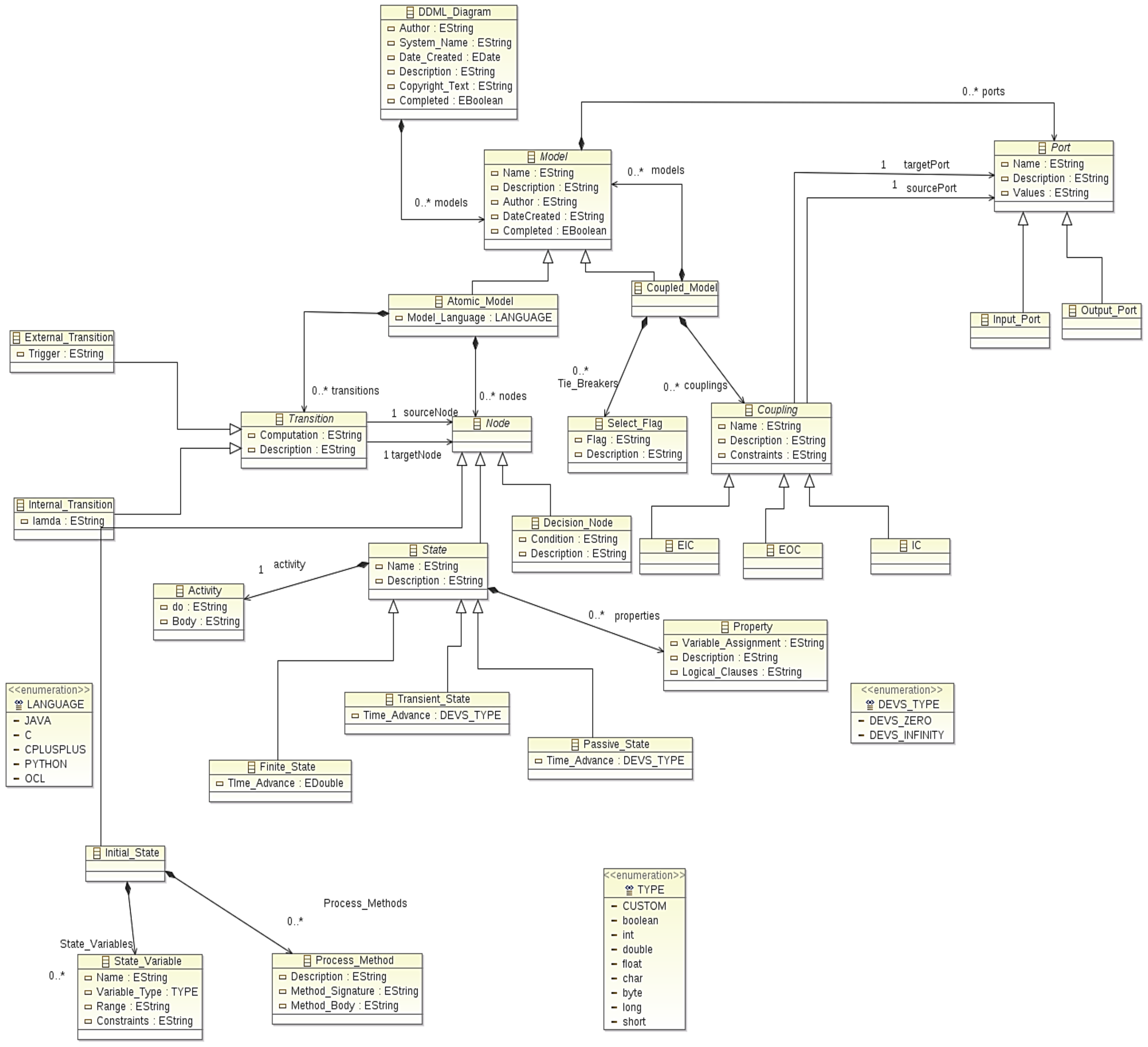


Figure 14. Ecore (Meta-Model) Model for DDML (Graphical)

DDML is hierarchical. First, we defined an editor for defining coupled models and atomic models with ports and connections (EIC, EOC, and IC). We call this editor the DDML Coupled Model Editor. Next, we defined an editor for defining state transitions for an atomic model. We call this editor the DDML Atomic Model Editor. Our idea here is that, the user would first define the coupled model and atomic models within a coupled model. Thereafter, the user can further define the state transitions for an atomic model by double clicking on an atomic model within the

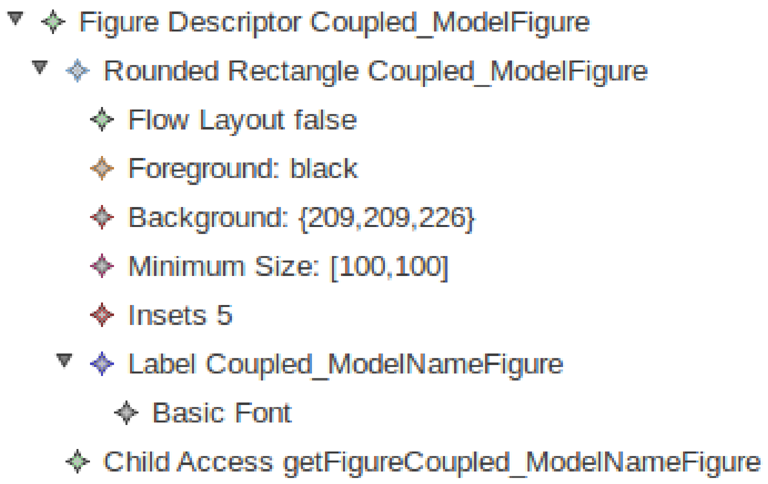
DDML Coupled Model Editor. This would launch the DDML Atomic Model Editor for state transitions definition.

5.1. The DDML Coupled Model Editor

5.1.1. Graphical Definition (DDML_Model.gmfgraph)

Building on the Ecore model defined in Figure 14, we created the GMF graphical definition was defined for DDML. This includes graphical definitions for Coupled Model, Atomic Model, Input Port, Output Port, External Input Coupling (EIC), External Output Coupling (EOC), and Internal Coupling (IC).

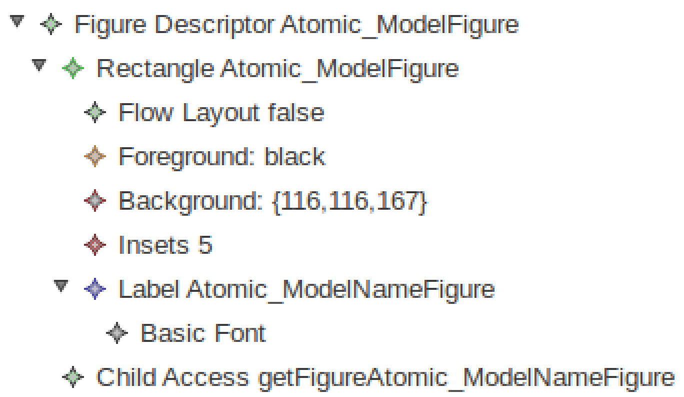
- We defined a Canvas at the root with a Figure gallery containing figure descriptors for our model elements (we call the diagram canvas ddml).
- We defined a set of figures to represent the domain model elements. The editor for this definition has dimension and color attributes such as line widths, foreground and background color attributes and static decorations. We constructed all the figures by adding them as a New Child to the present figure that represents our class.
- We defined the figure for the Coupled Model which is a Rounded Rectangle with a label (for the name of the atomic model) and other properties as shown in Figure 15 below:



```
▼ Figure Descriptor Coupled_ModelFigure
  ▼ Rounded Rectangle Coupled_ModelFigure
      Flow Layout false
      Foreground: black
      Background: {209,209,226}
      Minimum Size: [100,100]
      Insets 5
    ▼ Label Coupled_ModelNameFigure
        Basic Font
    Child Access getFigureCoupled_ModelNameFigure
```

Figure 15. Figure Descriptor for Coupled Model

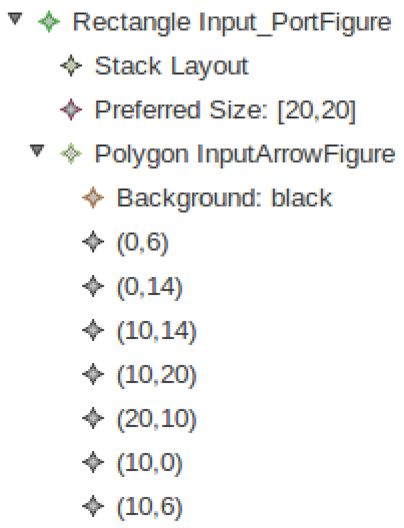
- We defined the figure for the Atomic Model which is a rectangle with a label (for the name of the atomic model) and other properties as shown in Figure 16.



```
▼ Figure Descriptor Atomic_ModelFigure
  ▼ Rectangle Atomic_ModelFigure
      Flow Layout false
      Foreground: black
      Background: {116,116,167}
      Insets 5
    ▼ Label Atomic_ModelNameFigure
        Basic Font
    Child Access getFigureAtomic_ModelNameFigure
```

Figure 16. Figure Descriptor for Atomic Model

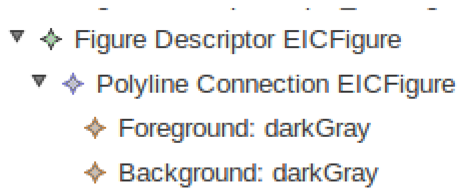
- We define figures for the input and output ports which is a polygon (with template points as shown in Figure 17 below



```
▼ Rectangle Input_PortFigure
    Stack Layout
    Preferred Size: [20,20]
  ▼ Polygon InputArrowFigure
      Background: black
      (0,6)
      (0,14)
      (10,14)
      (10,20)
      (20,10)
      (10,0)
      (10,6)
```

Figure 17. Figure Descriptor for Input Port

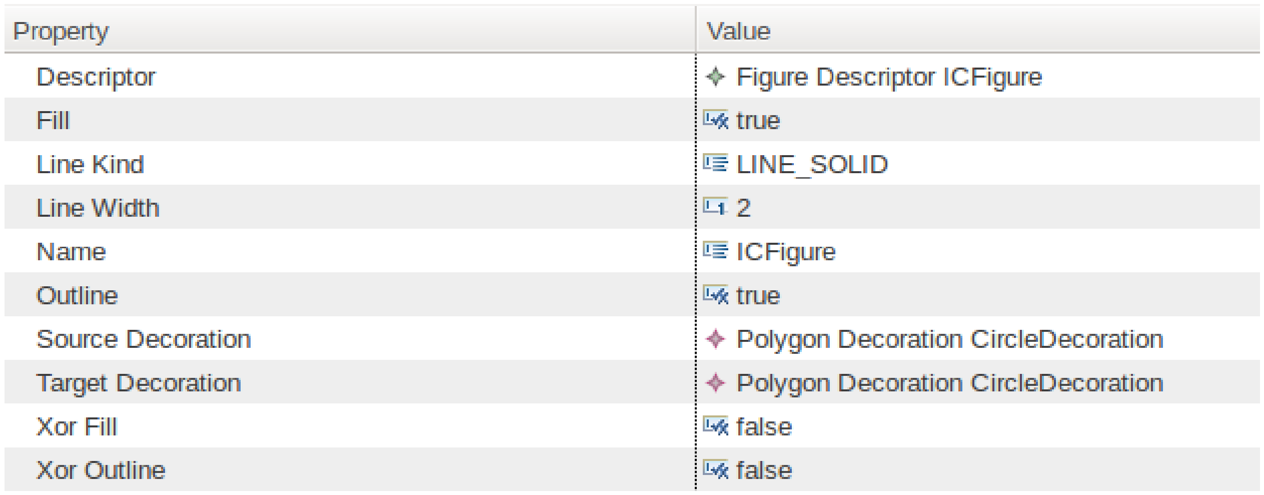
- We define figures for EIC, EOC, and IC which are polyline connections with their particular line types, source and target decorations (circular decoration as defined below) as shown in the Figure 18 below



```
▼ Figure Descriptor EICFigure
  ▼ Polyline Connection EICFigure
      Foreground: darkGray
      Background: darkGray
```

Figure 18. Figure Descriptor for EIC

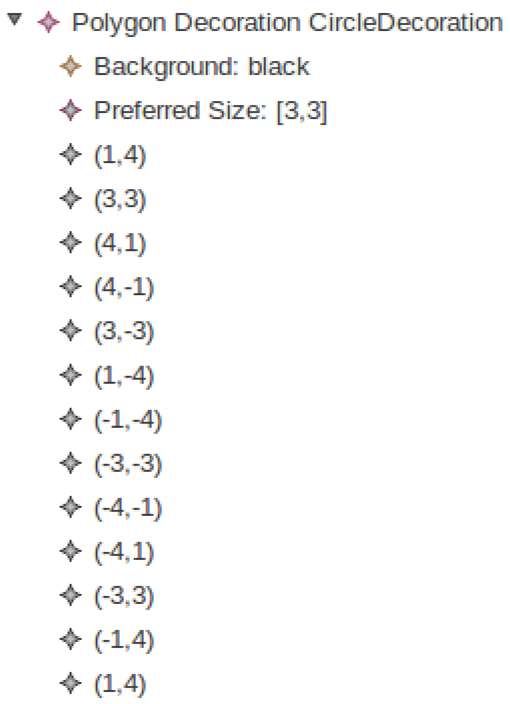
For each of the Connection Figures, we made tweaks to the Properties to define the line properties. The properties of the IC Figure are shown in Figure 19.



Property	Value
Descriptor	Figure Descriptor ICFigure
Fill	true
Line Kind	LINE_SOLID
Line Width	2
Name	ICFigure
Outline	true
Source Decoration	Polygon Decoration CircleDecoration
Target Decoration	Polygon Decoration CircleDecoration
Xor Fill	false
Xor Outline	false

Figure 19. Properties Definition for IC

- We defined a Polygon Decoration (circular decoration) for the EIC, EOC, and IC connections as shown in Figure 20



```
▼ Polygon Decoration CircleDecoration
    Background: black
    Preferred Size: [3,3]
    (1,4)
    (3,3)
    (4,1)
    (4,-1)
    (3,-3)
    (1,-4)
    (-1,-4)
    (-3,-3)
    (-4,-1)
    (-4,1)
    (-3,3)
    (-1,4)
    (1,4)
```

Figure 20. Figure Descriptor for Circular Decoration

- We defined graph nodes and connections. The domain model elements that are to be placed on the diagram editor canvas as nodes were defined as Node. The nodes in our DDML_MODEL.gmfgraph definition are shown in Figure 21:

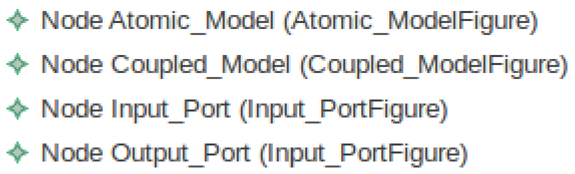
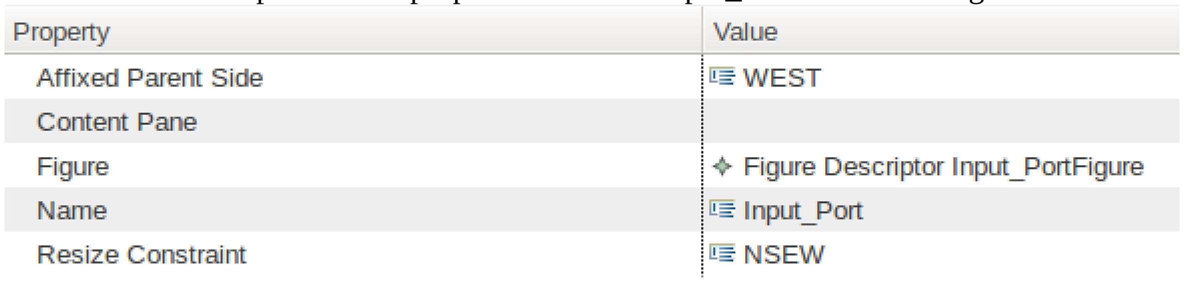

Node Atomic_Model (Atomic_ModelFigure)
Node Coupled_Model (Coupled_ModelFigure)
Node Input_Port (Input_PortFigure)
Node Output_Port (Input_PortFigure)

Figure 21. Graph Nodes for DDML Model

- We also associated each of these nodes to the corresponding figure descriptor.
- We also made tweaks to the properties of the Input Port Node and Output Port Nodes. These nodes would be affixed to the side of a parent node (a Model node). We defined the Input Port to be affixed to the West side of the parent and the Output Port to be affixed to the East side of the parent. The properties of Node Input_Port are shown Figure 22.



Property	Value
Affixed Parent Side	WEST
Content Pane	
Figure	Figure Descriptor Input_PortFigure
Name	Input_Port
Resize Constraint	NSEW

Figure 22. Properties Definition for Input Port

- We defined connections. Domain elements that are to be specified as connections to link the domain elements were defined as Connection. The connections in our graphical definition are shown in Figure 23:

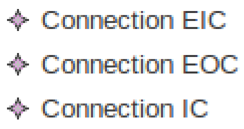

Connection EIC
Connection EOC
Connection IC

Figure 23. Connections for DDML Model

We also associated each connection to the corresponding figure description.

- We defined Compartments. Compartments are sections in nodes that can be collapsed and themselves can contain other nodes or list of elements. We specify two compartments in the Coupled_ModelFigure. They are shown in Figure 24.

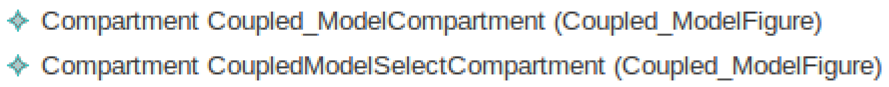

Compartment Coupled_ModelCompartment (Coupled_ModelFigure)
Compartment CoupledModelSelectCompartment (Coupled_ModelFigure)

Figure 24. Coupled Model Compartments

- Finally, we defined diagram labels to show the text associated with the graphical elements. The labels are shown in Figure 25.

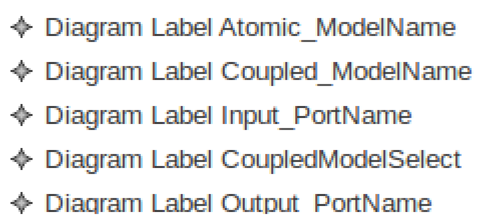
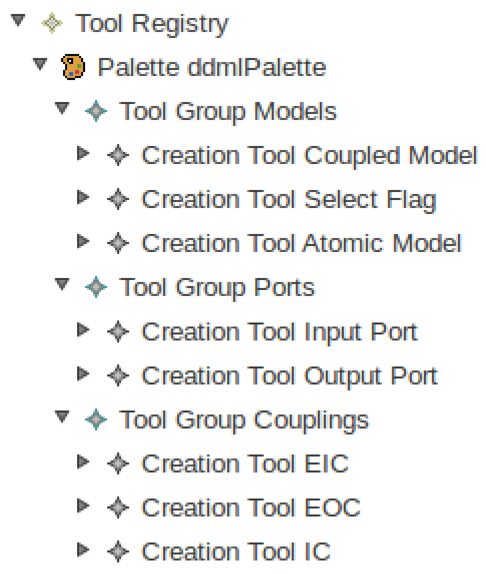


Diagram Label Atomic_ModelName
Diagram Label Coupled_ModelName
Diagram Label Input_PortName
Diagram Label CoupledModelSelect
Diagram Label Output_PortName

Figure 25. Diagram Labels for DDML

5.1.2. DDML Tooling Definition

The DDML_MODEL.gmftool model defines a set of palette entries. The palette is the set of figure buttons (on the right of the editor) that allows model elements to be added to the domain model instance. As shown in Figure 26, we defined tools for Models, Ports and Couplings. Then we defined an icon for each of the tools.



```
Tool Registry
  Palette ddmlPalette
    Tool Group Models
      Creation Tool Coupled Model
      Creation Tool Select Flag
      Creation Tool Atomic Model
    Tool Group Ports
      Creation Tool Input Port
      Creation Tool Output Port
    Tool Group Couplings
      Creation Tool EIC
      Creation Tool EOC
      Creation Tool IC
```

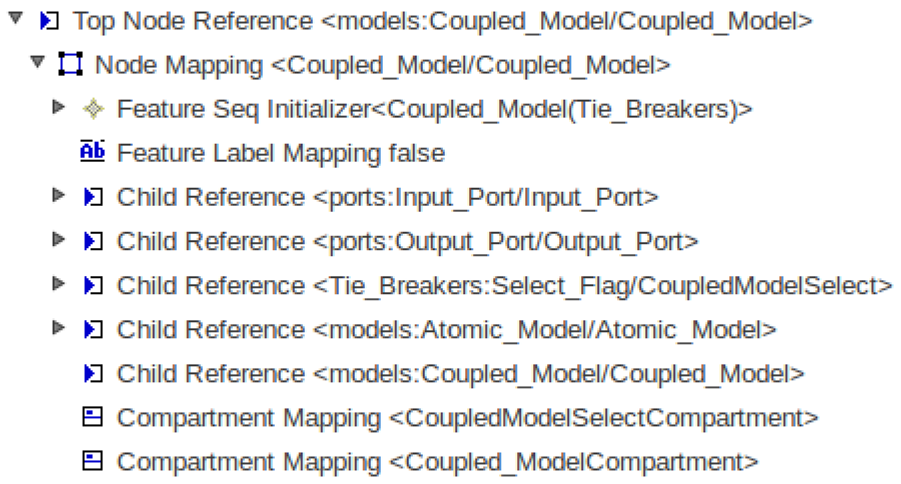
Figure 26. Tooling Definition for DDML Model

5.1.3. DDML Mapping Definition

This is the most complex model. Here we mapped the tooling definition and the graphical definition to the domain model. For each domain model that we want to map directly onto the diagram surface (the editpattern) we have to first define a Top Node Reference. Below that, we add a normal Node Mapping. This contains information about the model element to map to the tooling and graphical definitions. We did the following to define our mapping definition:

- We created a Top Node Reference for Coupled Models (as shown in the Figure 27). To this we added a Node Mapping to map the map the Coupled Model Domain Element, the Coupled Model Graphical Definition, and the Coupled Model Creation Tool. We then added several Child References to this node. As shown in the figure, we added Child

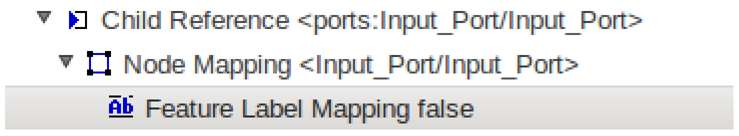
References for Input Ports, Output Ports, Tie Breakers (Select Function), Atomic Models, and a Child Reference which references the Node Mapping for the Coupled Model Node Mapping. We also added Compartment Mappings for the SelectCompartment (a compartment for the Select Flag, with Child as the Child Reference for Select Flag) and ModelCompartments (with Children as the Child References for the Atomic Model and Coupled Model). We also defined the Containment Features and the Children Features for the Child References.



```
▼ Top Node Reference <models:Coupled_Model/Coupled_Model>
  ▼ Node Mapping <Coupled_Model/Coupled_Model>
    ▶ Feature Seq Initializer<Coupled_Model(Tie_Breakers)>
      Feature Label Mapping false
    ▶ Child Reference <ports:Input_Port/Input_Port>
    ▶ Child Reference <ports:Output_Port/Output_Port>
    ▶ Child Reference <Tie_Breakers:Select_Flag/CoupledModelSelect>
    ▶ Child Reference <models:Atomic_Model/Atomic_Model>
      Child Reference <models:Coupled_Model/Coupled_Model>
      Compartment Mapping <CoupledModelSelectCompartment>
      Compartment Mapping <Coupled_ModelCompartment>
```

Figure 27. Mapping Definition for Coupled Model

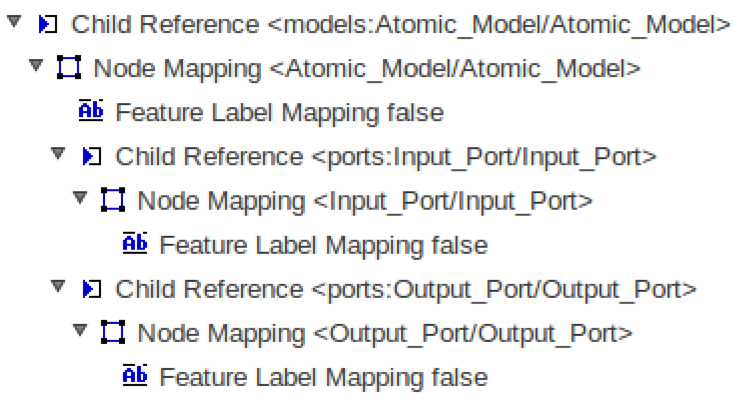
- For each of the Child References, we defined a Node Mapping to map the Domain Element, the Graphical Definition for the element and the Creation Tool. In some cases, we made a reference to an existing Node Mapping, for example, the Child Reference for Coupled Model references the Node Mapping for the Top Node Reference (Coupled Model).
- We also defined Label Mappings and Feature Label Mapping for each Node Mapping. For example, for Input Ports, we mapped the Input Port Name to the Model. This has been defined by an External Label in our Graphical Definition. This is illustrated in Figure 28.



```
▼ Child Reference <ports:Input_Port/Input_Port>
  ▼ Node Mapping <Input_Port/Input_Port>
      Feature Label Mapping false
```

Figure 28. Adding Label Mapping

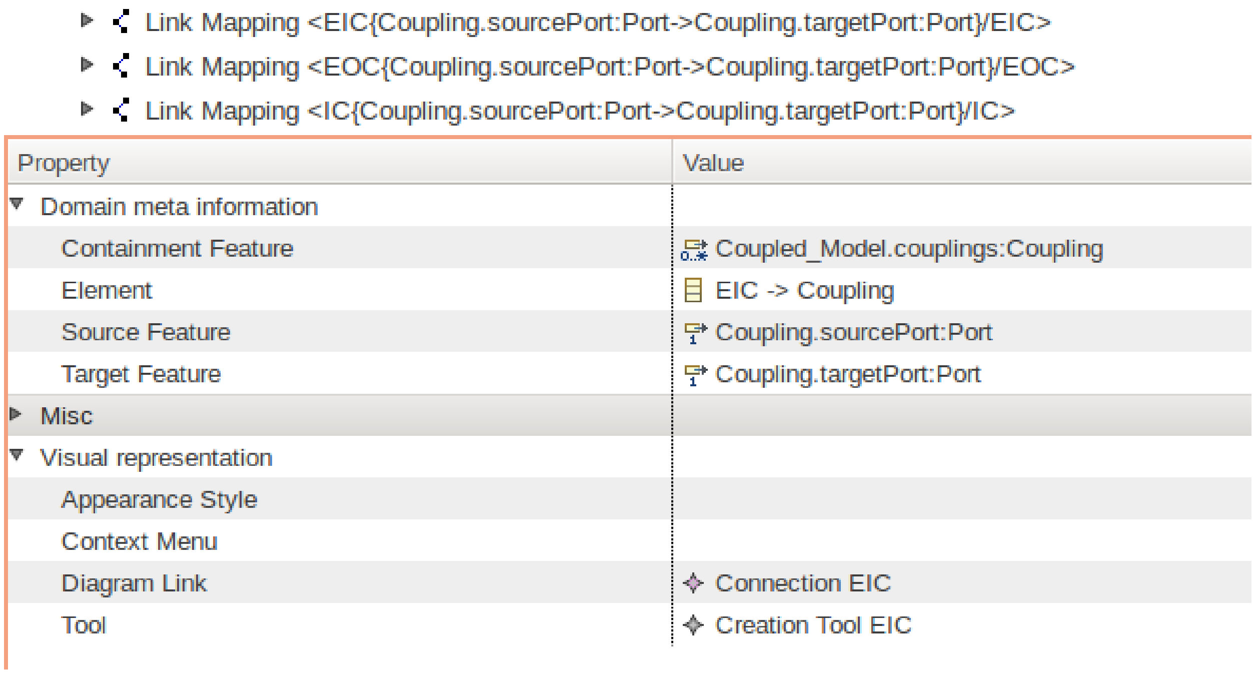
- For the Node Mapping for the Child Reference to Atomic Model (within the Node Mapping for Coupled Model to the Top Node Reference for Coupled Model), we define Children Child References (i.e. Child References within a Child Reference) for Input Ports and Output Ports. This ensures that Our Atomic Model can also contain Input Ports and Output Ports. See Figure 29 for the illustration:



```
▼ Child Reference <models:Atomic_Model/Atomic_Model>
  ▼ Node Mapping <Atomic_Model/Atomic_Model>
      Feature Label Mapping false
    ▼ Child Reference <ports:Input_Port/Input_Port>
      ▼ Node Mapping <Input_Port/Input_Port>
          Feature Label Mapping false
    ▼ Child Reference <ports:Output_Port/Output_Port>
      ▼ Node Mapping <Output_Port/Output_Port>
          Feature Label Mapping false
```

Figure 29. Node Mapping for Child Reference Atomic

- We defined Link Mappings for the EIC, EOC, and IC links. For each link, we mapped the Domain Element, Graphical Definition, and Creation Tool. This is illustrated in Figure 30.



```
▶ Link Mapping <EIC{Coupling.sourcePort:Port->Coupling.targetPort:Port}/EIC>
▶ Link Mapping <EOC{Coupling.sourcePort:Port->Coupling.targetPort:Port}/EOC>
▶ Link Mapping <IC{Coupling.sourcePort:Port->Coupling.targetPort:Port}/IC>
```

Property	Value
▼ Domain meta information	
Containment Feature	Coupled_Model.couplings:Coupling
Element	EIC -> Coupling
Source Feature	Coupling.sourcePort:Port
Target Feature	Coupling.targetPort:Port
▶ Misc	
▼ Visual representation	
Appearance Style	
Context Menu	
Diagram Link	Connection EIC
Tool	Creation Tool EIC

Figure 30. Link Mappings (above); Properties for Link Mapping

From the properties (as shown in the Figure 30), we defined the Containment Feature, Domain Element, Source Feature, Target Feature, Diagram Link, and Tool.

- We defined constraints for each link using Object Constraint Language (OCL). Link Constraints are used to validate the links upon creation between any two elements. The constraints can be specified using OCL as Source End Constraint. and Target End Constraint. (Refer to Figure 31)
  - For EIC, we define Source Constraint (not self.oclIsKindOf(Output_Port) and (self <> oppositeEnd)) and Target Constraint (not

self.oclIsKindOf(Output_Port) and (self <> oppositeEnd)). This ensures that an EIC connects only an Input Port from the Parent Coupled Model and an Input Port in the Child Model (Coupled Model or Atomic Model).

- For EOC, we define Source Constraint (not self.oclIsKindOf(Input_Port) and (self <> oppositeEnd)) and Target Constraint (not self.oclIsKindOf(Intput_Port) (self <> oppositeEnd)). This ensures that an EOC connects only an Output Port from the Parent Coupled Model and an Output Port in the Child Model (Coupled Model or Atomic Model).
- For IC, we define Source Constraint(not self.oclIsKindOf(Input_Port) (self <> oppositeEnd)) and Target Constraint(not self.oclIsKindOf(Output_Port) (self <> oppositeEnd)). This ensures that an EOC connects only an Output Port from a Model and an Input Port in another Model (Both Models within a parent Coupled Model).

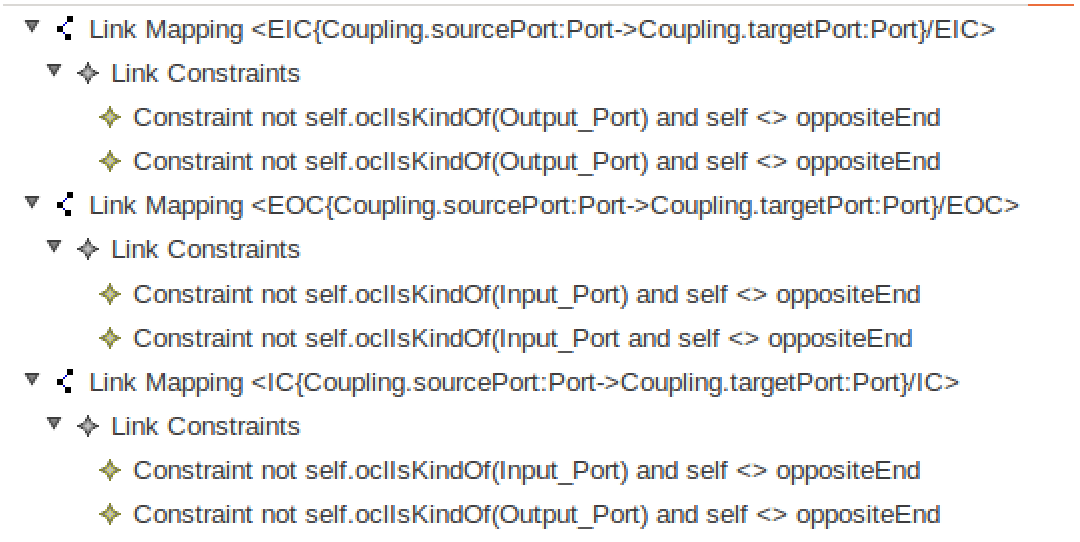
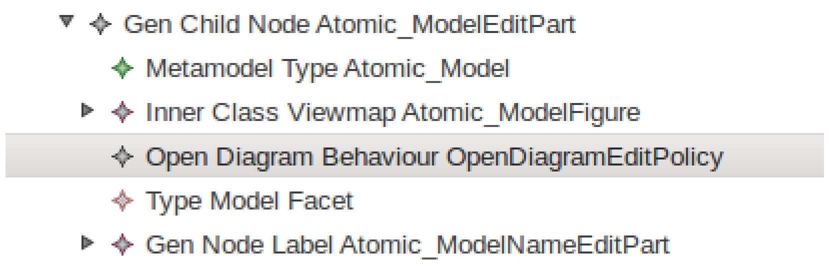

```
▼ Link Mapping <EIC{Coupling.sourcePort:Port->Coupling.targetPort:Port}/EIC>
  ▼ Link Constraints
      Constraint not self.oclIsKindOf(Output_Port) and self <> oppositeEnd
      Constraint not self.oclIsKindOf(Output_Port) and self <> oppositeEnd
▼ Link Mapping <EOC{Coupling.sourcePort:Port->Coupling.targetPort:Port}/EOC>
  ▼ Link Constraints
      Constraint not self.oclIsKindOf(Input_Port) and self <> oppositeEnd
      Constraint not self.oclIsKindOf(Input_Port and self <> oppositeEnd
▼ Link Mapping <IC{Coupling.sourcePort:Port->Coupling.targetPort:Port}/IC>
  ▼ Link Constraints
      Constraint not self.oclIsKindOf(Input_Port) and self <> oppositeEnd
      Constraint not self.oclIsKindOf(Output_Port) and self <> oppositeEnd
```

Figure 31. Defining Link Constraints

5.1.4. DDML Generator Model

We transformed the .gmfmap model to the generator model (.gmfgen) using the facilities that Eclipse provides. During the transformation, reference was made to the .genmodel file that was generated from the .ecore model. We made some tweaks to the generated .gmfgen model XMI file. We separated the domain model (.ddmdm file) and the diagram file (.ddmdg). We specified a partition for our editor by defining Open Diagram Behaviour Policy. This is illustrated in Figure 32.



```
▼ Gen Child Node Atomic_ModelEditPart
    Metamodel Type Atomic_Model
  ▶ Inner Class Viewmap Atomic_ModelFigure
    Open Diagram Behaviour OpenDiagramEditPolicy
    Type Model Facet
  ▶ Gen Node Label Atomic_ModelNameEditPart
```

Figure 32. Setting Open Diagram Behavior for

The Open Diagram Behaviour Policy references the Atomic Model Plugin. This implies that double clicking the Atomic Diagram Model Figure would open another editor to define the process model. This second editor is described in the sections to follow.

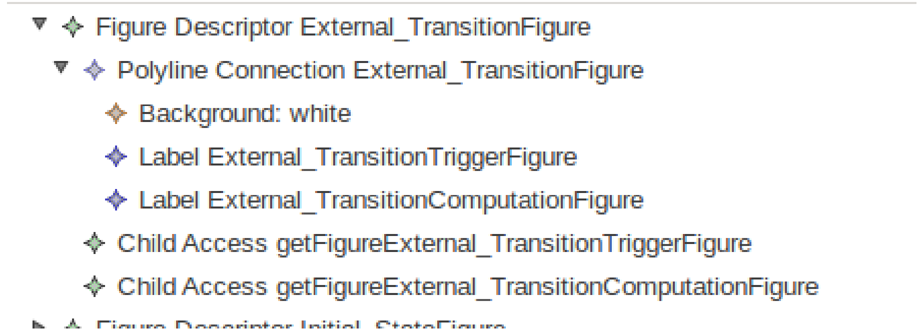
We generated the editor code using the facilities that eclipse provides. We also made modifications to the generated code to meet the functionalities of our graphical editor.

5.2. The DDML Atomic Model Editor

5.2.1. Atomic Model Graphical Definition

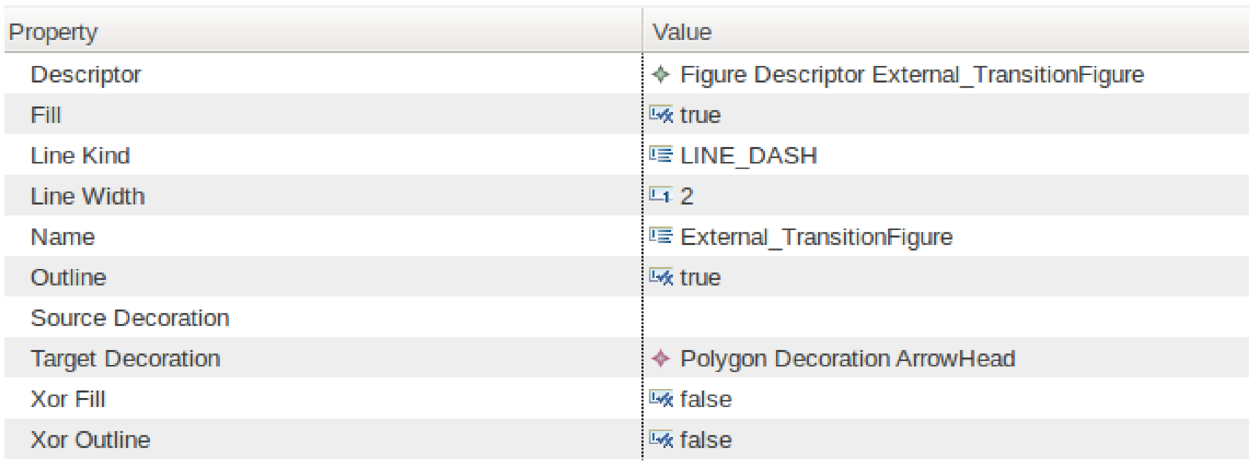
Just like we did with the DDML Model Graphical Definition, we defined the notations and descriptors for the following Process Model Figures: Initial State, Finite State, Passive State, Transient State, External Transition, Internal Transition, Conditional Transition, State Properties, State Activities, Time Advance, and Process Method Definition.

- Figure descriptors for Internal and External Transitions include Polyline Connections (Figure 33). The Properties of the Connections are defined in the Properties View (shown in Figure 34) below for External Transition). For Internal Transition, we used a Solid Line as the Line Kind



```
▼ Figure Descriptor External_TransitionFigure
  ▼ Polyline Connection External_TransitionFigure
      Background: white
      Label External_TransitionTriggerFigure
      Label External_TransitionComputationFigure
    Child Access getFigureExternal_TransitionTriggerFigure
    Child Access getFigureExternal_TransitionComputationFigure
▶ Figure Descriptor Initial_StateFigure
```

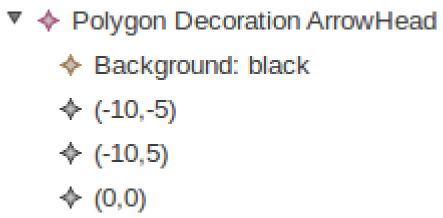
Figure 33. Figure Descriptor for External Transition



Property	Value
Descriptor	Figure Descriptor External_TransitionFigure
Fill	true
Line Kind	LINE_DASH
Line Width	2
Name	External_TransitionFigure
Outline	true
Source Decoration	
Target Decoration	Polygon Decoration ArrowHead
Xor Fill	false
Xor Outline	false

Figure 34. Properties Definition for External

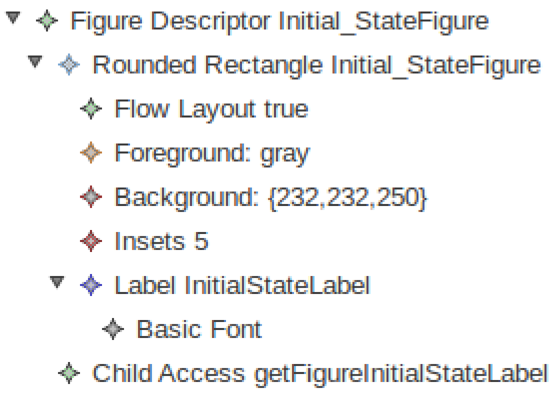
- The figure above shows that we define a target decoration (Polygon Decoration ArrowHead). We show the definition of the target decoration in Figure 35.



```
▼ Polygon Decoration ArrowHead
    Background: black
    (-10,-5)
    (-10,5)
    (0,0)
```

Figure 35. Figure Descriptor for ArrowHead

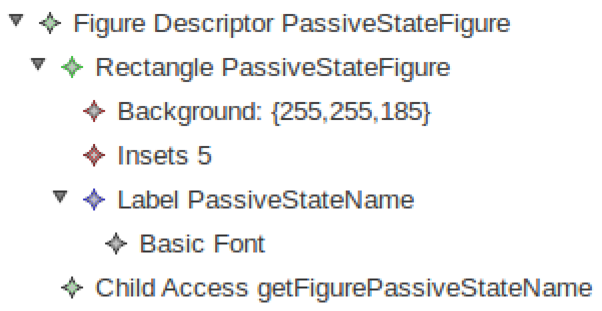
- We defined figure descriptor for Initial State as shown in Figure 36.



```
▼ Figure Descriptor Initial_StateFigure
  ▼ Rounded Rectangle Initial_StateFigure
      Flow Layout true
      Foreground: gray
      Background: {232,232,250}
      Insets 5
    ▼ Label InitialStateLabel
        Basic Font
    Child Access getFigureInitialStateLabel
```

Figure 36. Figure Descriptor for Initial State

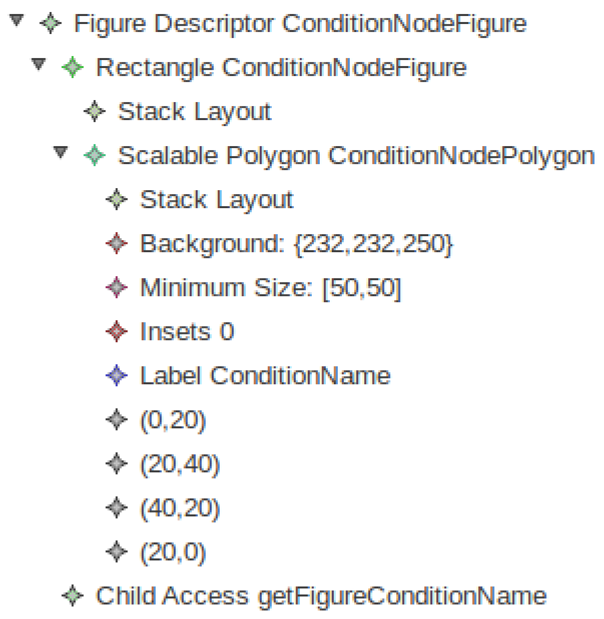
- We defined figure descriptors for definite state, passive state, and transient state. The figure descriptor for passive state is shown in the Figure 37.



```
▼ ✦ Figure Descriptor PassiveStateFigure
  ▼ ✦ Rectangle PassiveStateFigure
      ✦ Background: {255,255,185}
      ✦ Insets 5
    ▼ ✦ Label PassiveStateName
        ✦ Basic Font
    ✦ Child Access getFigurePassiveStateName
```

Figure 37. Figure Descriptor for Passive State

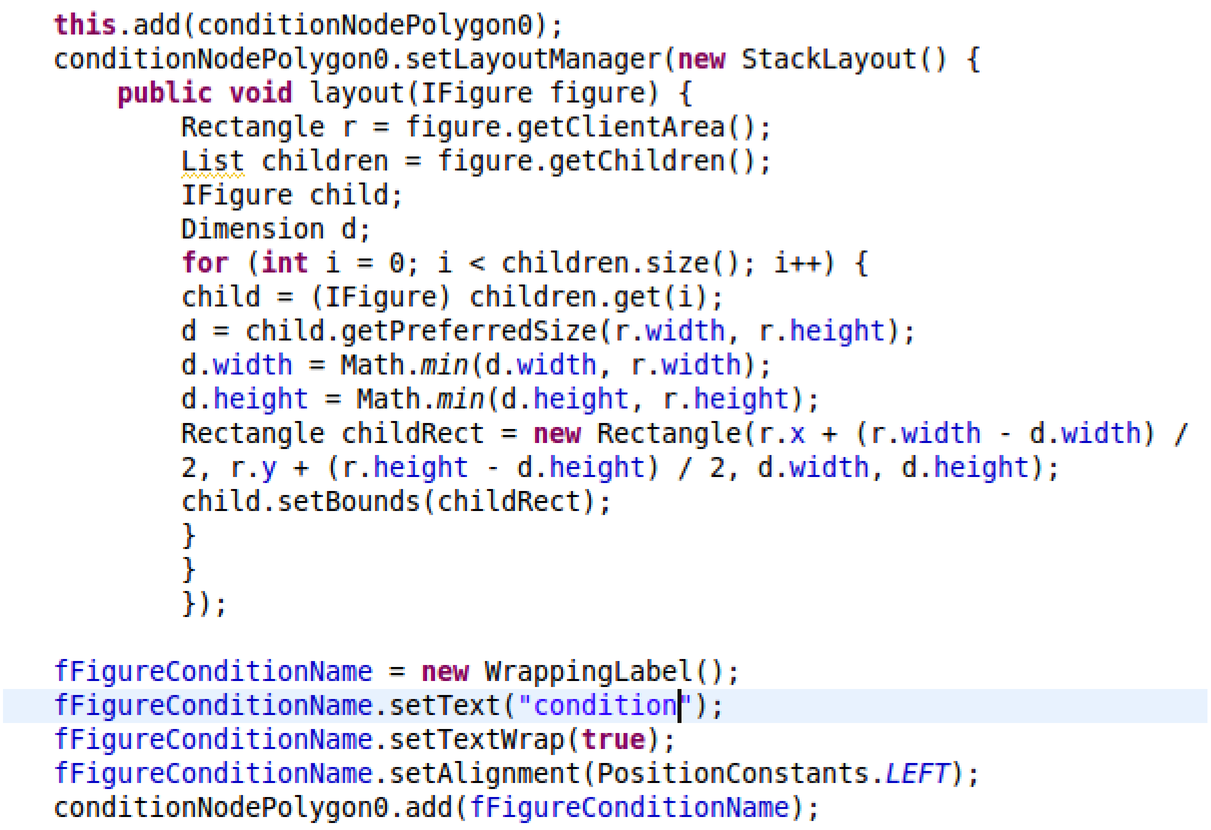
- We defined figure descriptors for conditional transition node as shown in Figure 38.



```
▼ ✦ Figure Descriptor ConditionNodeFigure
  ▼ ✦ Rectangle ConditionNodeFigure
      ✦ Stack Layout
    ▼ ✦ Scalable Polygon ConditionNodePolygon
        ✦ Stack Layout
        ✦ Background: {232,232,250}
        ✦ Minimum Size: [50,50]
        ✦ Insets 0
        ✦ Label ConditionName
        ✦ (0,20)
        ✦ (20,40)
        ✦ (40,20)
        ✦ (20,0)
    ✦ Child Access getFigureConditionName
```

Figure 38. Figure Descriptor for Conditional Transition Node

In the generated code (atomic.diagram.edit.parts.Decision_NodeEditPart class), we make the following tweaks to adjust the Stack Layout and ensure that the Label ConditionName is always wrapped.



```
this.add(conditionNodePolygon0);
conditionNodePolygon0.setLayoutManager(new StackLayout() {
    public void layout(IFigure figure) {
        Rectangle r = figure.getClientArea();
        List children = figure.getChildren();
        IFigure child;
        Dimension d;
        for (int i = 0; i < children.size(); i++) {
        child = (IFigure) children.get(i);
        d = child.getPreferredSize(r.width, r.height);
        d.width = Math.min(d.width, r.width);
        d.height = Math.min(d.height, r.height);
        Rectangle childRect = new Rectangle(r.x + (r.width - d.width) /
        2, r.y + (r.height - d.height) / 2, d.width, d.height);
        child.setBounds(childRect);
        }
        }
        });

fFigureConditionName = new WrappingLabel();
fFigureConditionName.setText("condition");
fFigureConditionName.setTextWrap(true);
fFigureConditionName.setAlignment(PositionConstants.LEFT);
conditionNodePolygon0.add(fFigureConditionName);
```

- We defined Nodes, Compartments, Diagram Labels, and Connections and linked them to their respective figure descriptors. This is shown in Figure 39.

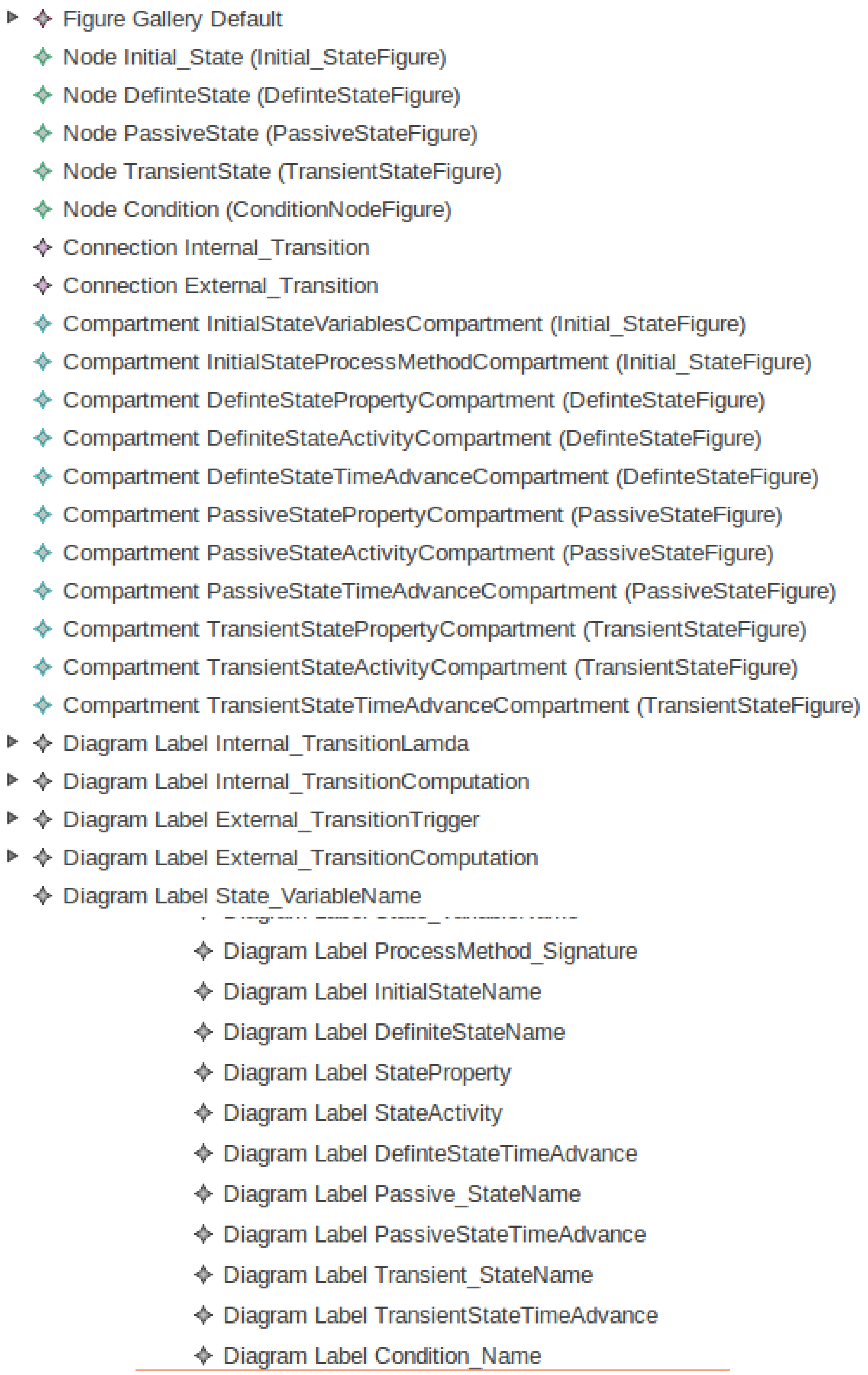
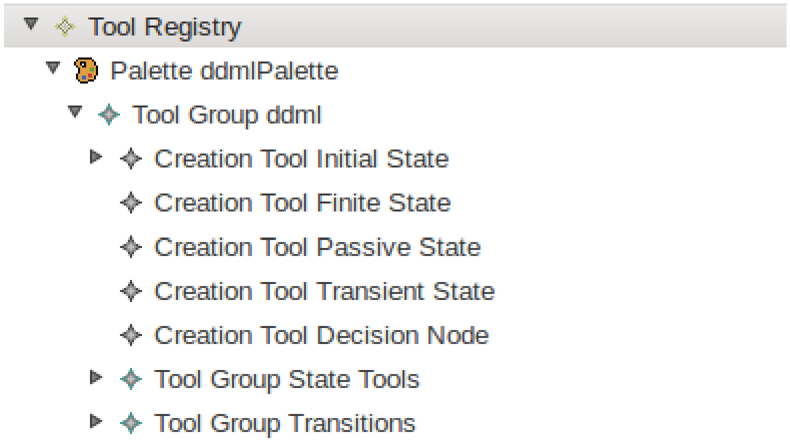


Figure Gallery Default
Node Initial_State (Initial_StateFigure)
Node DefinteState (DefinteStateFigure)
Node PassiveState (PassiveStateFigure)
Node TransientState (TransientStateFigure)
Node Condition (ConditionNodeFigure)
Connection Internal_Transition
Connection External_Transition
Compartment InitialStateVariablesCompartment (Initial_StateFigure)
Compartment InitialStateProcessMethodCompartment (Initial_StateFigure)
Compartment DefinteStatePropertyCompartment (DefinteStateFigure)
Compartment DefiniteStateActivityCompartment (DefinteStateFigure)
Compartment DefinteStateTimeAdvanceCompartment (DefinteStateFigure)
Compartment PassiveStatePropertyCompartment (PassiveStateFigure)
Compartment PassiveStateActivityCompartment (PassiveStateFigure)
Compartment PassiveStateTimeAdvanceCompartment (PassiveStateFigure)
Compartment TransientStatePropertyCompartment (TransientStateFigure)
Compartment TransientStateActivityCompartment (TransientStateFigure)
Compartment TransientStateTimeAdvanceCompartment (TransientStateFigure)
Diagram Label Internal_TransitionLamda
Diagram Label Internal_TransitionComputation
Diagram Label External_TransitionTrigger
Diagram Label External_TransitionComputation
Diagram Label State_VariableName
Diagram Label ProcessMethod_Signature
Diagram Label InitialStateName
Diagram Label DefiniteStateName
Diagram Label StateProperty
Diagram Label StateActivity
Diagram Label DefinteStateTimeAdvance
Diagram Label Passive_StateName
Diagram Label PassiveStateTimeAdvance
Diagram Label Transient_StateName
Diagram Label TransientStateTimeAdvance
Diagram Label Condition_Name

Figure 39. Definitions for Nodes, Compartments, Connections and Labels

5.2.2. Atomic Model Tooling Definition

We defined the Atomic_Model.gmftool which creates the palette and tools for the Aomic Model Editor. In the generated code, we defined the icons for the tools. Figure 40 shows the tools that were created:



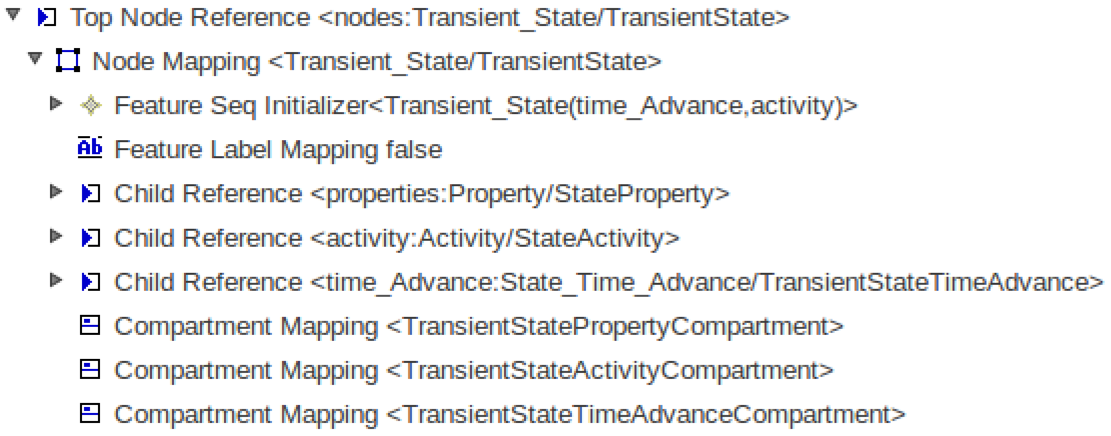
```
▼ Tool Registry
  ▼ Palette ddmlPalette
    ▼ Tool Group ddml
      ▶ Creation Tool Initial State
        Creation Tool Finite State
        Creation Tool Passive State
        Creation Tool Transient State
        Creation Tool Decision Node
      ▶ Tool Group State Tools
      ▶ Tool Group Transitions
```

Figure 40. Tooling Definition for Atomic Model

5.2.3. Atomic Model Mapping Definition

Just as we did when we defined the mapping definition for the DDML Coupled Model (DDML_Model.gmfmap), we defined mappings to define the Atomic_Model.gmfmap. The following were done to achieve this:

- We defined a Top Node Reference for Transient State as shown in Figure 41.

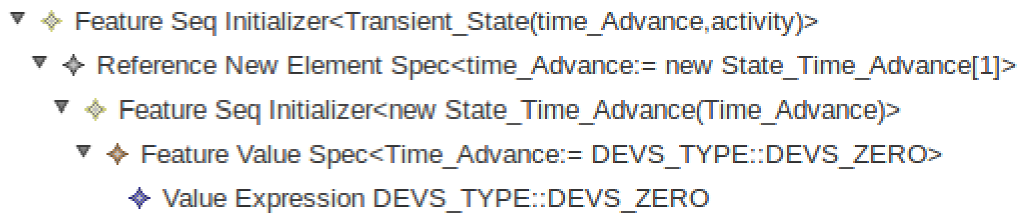


```
▼ Top Node Reference <nodes:Transient_State/TransientState>
  ▼ Node Mapping <Transient_State/TransientState>
    ▶ Feature Seq Initializer<Transient_State(time_Advance,activity)>
      Feature Label Mapping false
    ▶ Child Reference <properties:Property/StateProperty>
    ▶ Child Reference <activity:Activity/StateActivity>
    ▶ Child Reference <time_Advance:State_Time_Advance/TransientStateTimeAdvance>
      Compartment Mapping <TransientStatePropertyCompartment>
      Compartment Mapping <TransientStateActivityCompartment>
      Compartment Mapping <TransientStateTimeAdvanceCompartment>
```

Figure 41. Top Node Reference for Transition State

In the Node Mapping, we defined three Child References (Property, Activity, and Time Advance) and linked them to their corresponding Compartment Mapping. We also defined the corresponding Node Mapping for the Child References. As shown in Figure 42, we also defined a

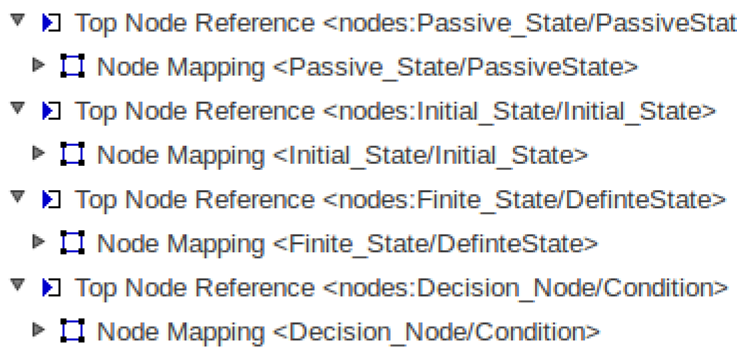
Feature Initializer to Initialize the Time Advance to DEVS_ZERO. This value is fixed for all Transient State. This initialization was done in OCL.



```
▼ Feature Seq Initializer<Transient_State(time_Advance,activity)>
  ▼ Reference New Element Spec<time_Advance:= new State_Time_Advance[1]>
    ▼ Feature Seq Initializer<new State_Time_Advance(Time_Advance)>
      ▼ Feature Value Spec<Time_Advance:= DEVS_TYPE::DEVS_ZERO>
          Value Expression DEVS_TYPE::DEVS_ZERO
```

Figure 42. Feature Initializer for Time Advance for Transient State

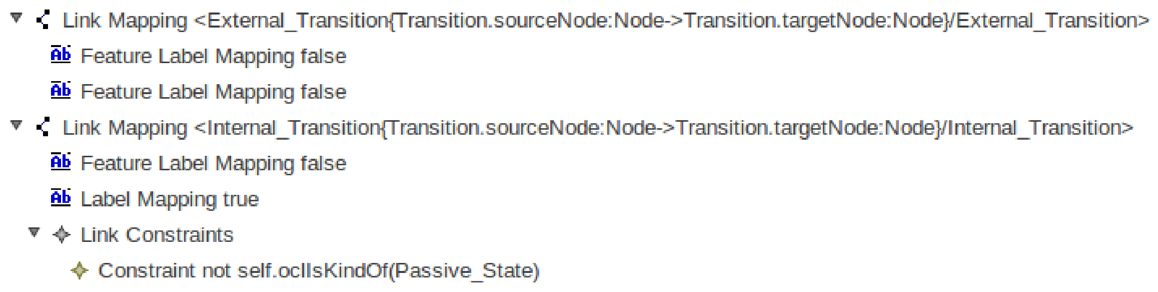
- We defined the Top Node References for Initial State, Passive State, and Finite State. We also defined their compartments, child references, and feature initialization as was necessary. Figure 43 shows the mappings (with Node Mappings suppressed)



```
▼ Top Node Reference <nodes:Passive_State/PassiveState>
  ▶ Node Mapping <Passive_State/PassiveState>
▼ Top Node Reference <nodes:Initial_State/Initial_State>
  ▶ Node Mapping <Initial_State/Initial_State>
▼ Top Node Reference <nodes:Finite_State/DefinteState>
  ▶ Node Mapping <Finite_State/DefinteState>
▼ Top Node Reference <nodes:Decision_Node/Condition>
  ▶ Node Mapping <Decision_Node/Condition>
```

Figure 43. Top Node Reference for Passive State

- We defined link mappings for external transition and internal transition. As shown in Figure 44, we defined a link constraint for the internal transition link. This constraint (Source End Constraint) was defined in OCL (not self.oclIsKindOf(Passive_State). This is necessary to ensure that passive states do not undergo internal transition.



```
▼ Link Mapping <External_Transition{Transition.sourceNode:Node->Transition.targetNode:Node}/External_Transition>
    Feature Label Mapping false
    Feature Label Mapping false
▼ Link Mapping <Internal_Transition{Transition.sourceNode:Node->Transition.targetNode:Node}/Internal_Transition>
    Feature Label Mapping false
    Label Mapping true
  ▼ Link Constraints
      Constraint not self.oclIsKindOf(Passive_State)
```

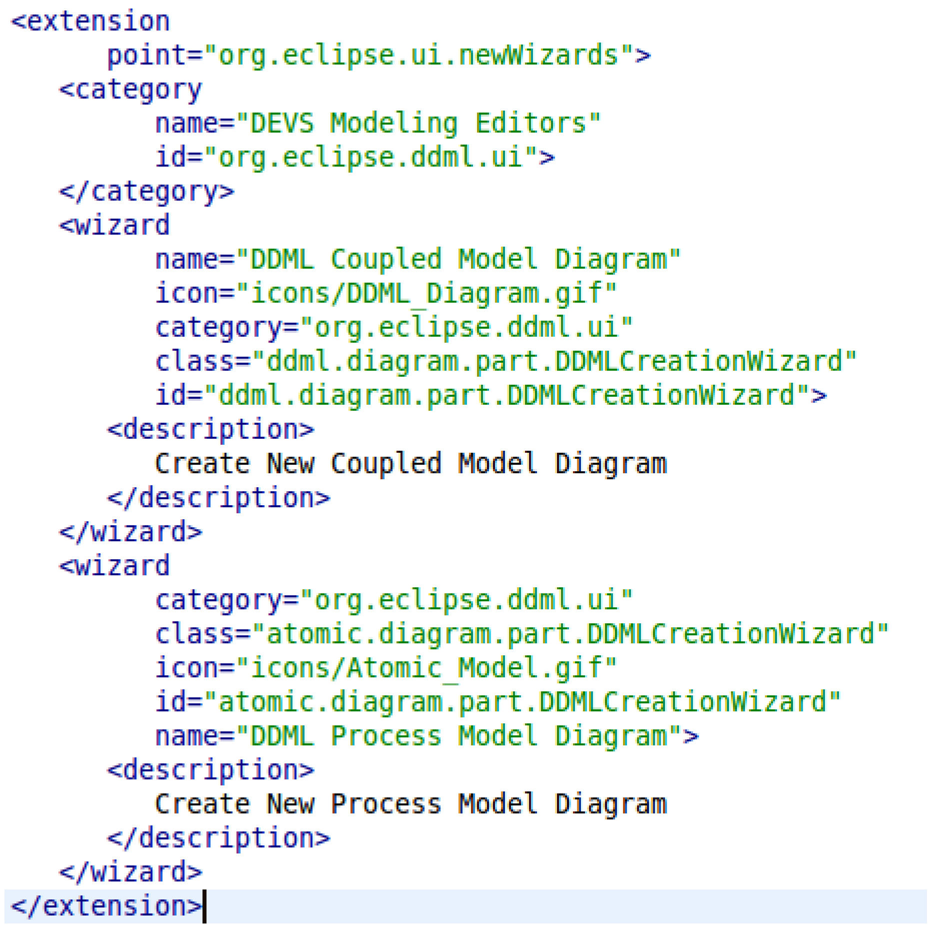
Figure 44. Link Mapping and Constraints for Transitions

5.2.4. Atomic Model Generator Model

Using the tools provided with eclipse, we generated the Atomic_Model.gmfgen file by referencing the .gmfmap and .genmodel files. We made some changes to the generated model. We separated the domain model (.atmdm file) and the diagram file (.atmdg). As noted in section **, this editor was referenced in the Open Diagram Behaviour Policy for Atomic Model defined in DDML_Model.gmfgen. The generator model was used to generate the editor plugin code and tweaks were made to the code to ensure that the editor meets the desired functionalities.

5.3. Creating the DDML UI Plugin

Using the Eclipse Plugin development tools, we defined an eclipse plugin to integrate the DDML Coupled Model plugin and the Atomic Model Plugin. The plugin has extensions from org.eclipse.ui.NewWizards. A snippet from the plugin.xml file is shown below:



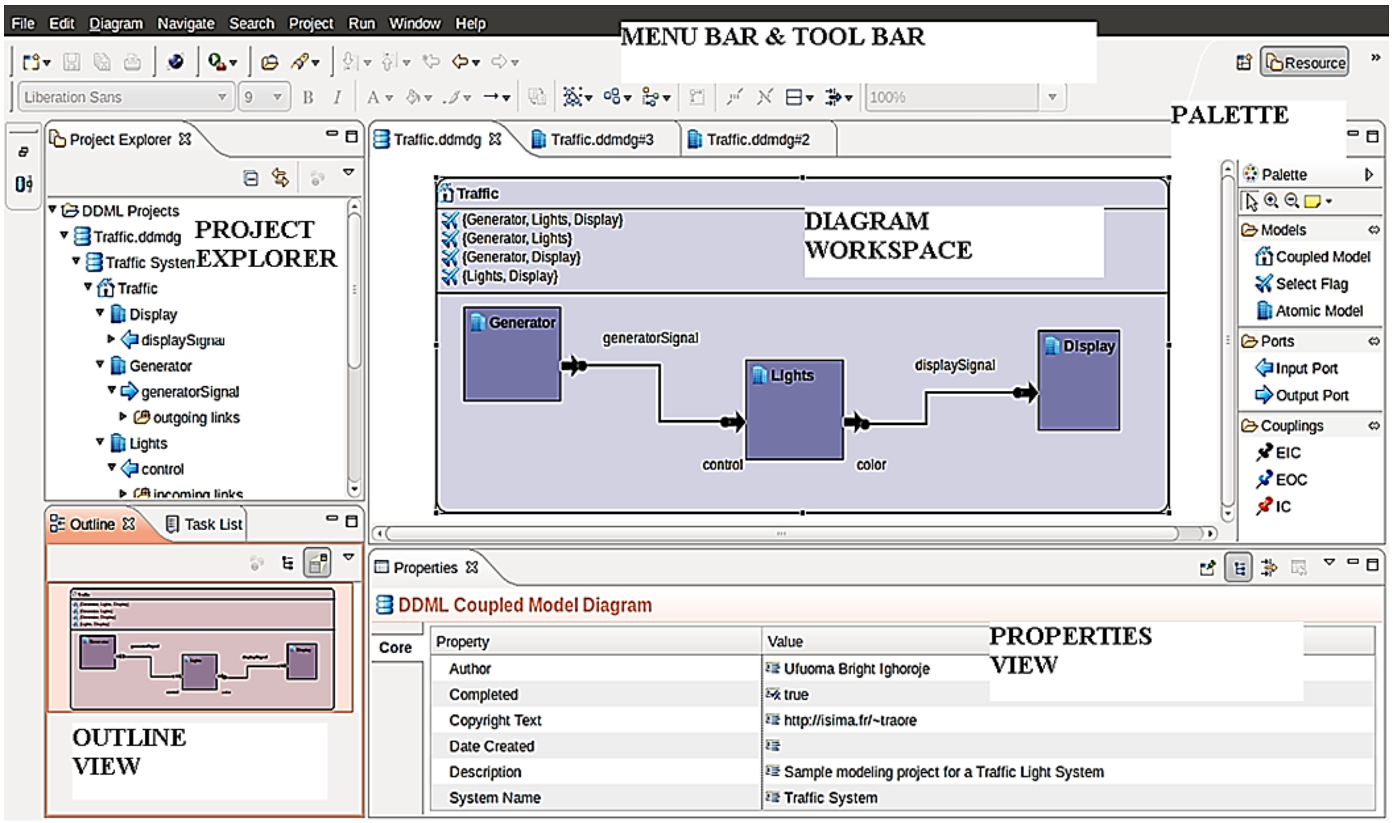
```
<extension
      point="org.eclipse.ui.newWizards">
   <category
         name="DEVS Modeling Editors"
         id="org.eclipse.ddml.ui">
   </category>
   <wizard
         name="DDML Coupled Model Diagram"
         icon="icons/DDML_Diagram.gif"
         category="org.eclipse.ddml.ui"
         class="ddml.diagram.part.DDMLCreationWizard"
         id="ddml.diagram.part.DDMLCreationWizard">
      <description>
         Create New Coupled Model Diagram
      </description>
   </wizard>
   <wizard
         category="org.eclipse.ddml.ui"
         class="atomic.diagram.part.DDMLCreationWizard"
         icon="icons/Atomic_Model.gif"
         id="atomic.diagram.part.DDMLCreationWizard"
         name="DDML Process Model Diagram">
      <description>
         Create New Process Model Diagram
      </description>
   </wizard>
</extension>
```

Chapter 6. Using the Eclipse-DDML Graphical Editor

The DDML Graphical editor is eclipse based and it is as easy to use as eclipse. In this chapter, we present the DDML Coupled Model Editor and the DDML Atomic Model Editor.

6.1. The DDML Coupled Model Editor

Figure 45 below shows a screen shot of the coupled model editor. The editor has a menu bar, a tool bar, project explorer, Outline view, Properties View, Palette, and the Model Editor Workspace. In the following sections, we will take a look at each of these sections.



DDML Coupled Model Diagram

Property	Value
Author	Ufuoma Bright Ighoroje
Completed	true
Copyright Text	http://isima.fr/~traore
Date Created	
Description	Sample modeling project for a Traffic Light System
System Name	Traffic System

Figure 45. DDML Coupled Model Editor

6.1.1. The Menu Bar and Tool Bar

Figure 46 shows the Menu Bar and Tool Bar for the DDML Coupled Model Editor. The menu bar includes the following menus: File, Edit, Diagram, Navigate, Search, Project, Run, Window and Help.

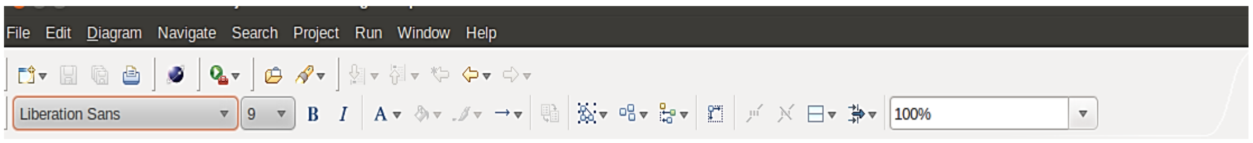
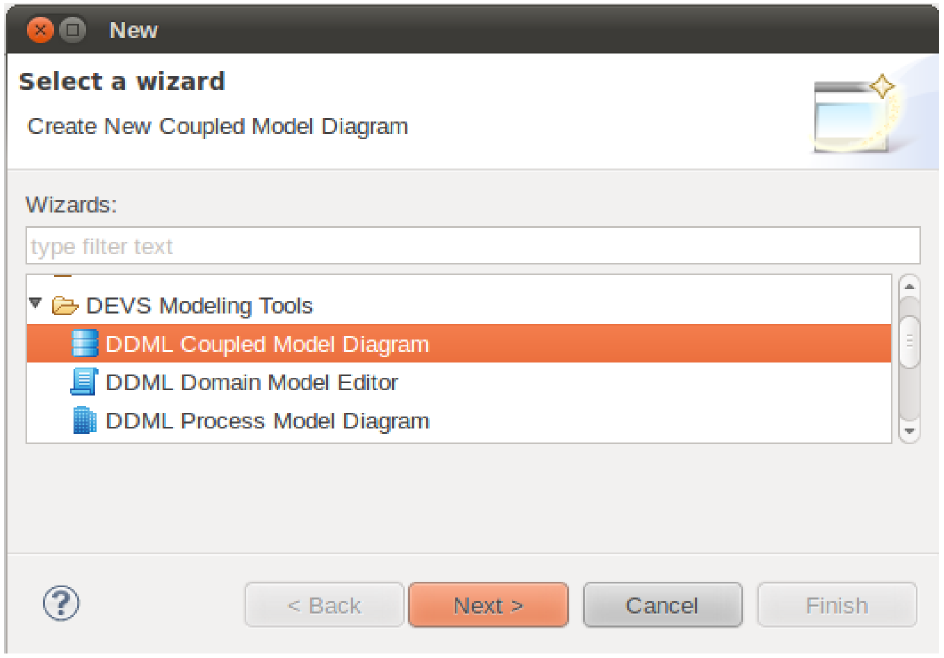


Figure 46. DDML Editor Menu Bar and Tool Bar

The File menu contains menu items for creating a new model, opening an existing model, closing the application window, saving a model, switching/selecting a workspace, exporting and importing models from other locations, printing a model diagram, etcetera. To create a new model, click on the File Menu, click on New, select Other, in the DEVS Modeling Tools category, select DDML Coupled Model Diagram Wizard (or the appropriate wizard). This is shown in Figure 47 below.


New

Select a wizard

Create New Coupled Model Diagram

Wizards:

type filter text

DEVS Modeling Tools
DDML Coupled Model Diagram
DDML Domain Model Editor
DDML Process Model Diagram

< Back | Next > | Cancel | Finish

Figure 47. New DDML Diagram Wizard

The Edit menu contains the following menu items: Undo, Redo, Cut, Copy, Paste, Delete, Select All, Find/Replace, Add Bookmark, Add Task. The keyboard shortcuts (same as the standard shortcuts for editors) for these items are shown beside the menu items.

The Diagram menu contains menu items to change the diagram properties. It includes Font, Color, Line Type, Line Size, Select, Arrange, Order, Align, and other features that facilitate graphical modeling.

The Search, Project, Run, Window, and Help menus are standard eclipse menus with similar functionalities.

The toolbar contains common tools for formatting the model diagram. These tools include tools used for formatting font properties, adjusting model diagram foreground and background properties, zooming, printing a model, saving a model, undoing and redoing modeling activity, selecting a model, and creating a new model. Just like in most editors, the properties of a tool can be discovered by resting the cursor on the tool.

6.1.2. The Project Explorer

The project explorer view (shown in Figure 48) provides a hierarchical view of the project and resources in the Workbench. To add the project explorer to the workbench, click **Window > Show View > Other... > General > Project Explorer**.

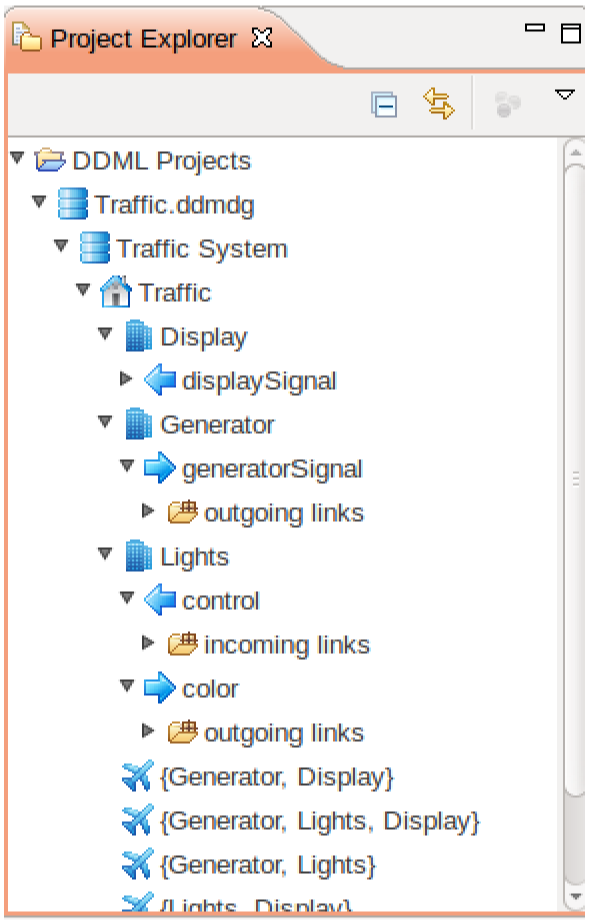


Figure 48. The Project Explorer

6.1.3. The Outline View

The outline view shows a graphical outline of the workspace. Figure 49 shows what the outline view looks like. To add the outline view to the workbench, click Window > Show View > Other... > General > Outline.

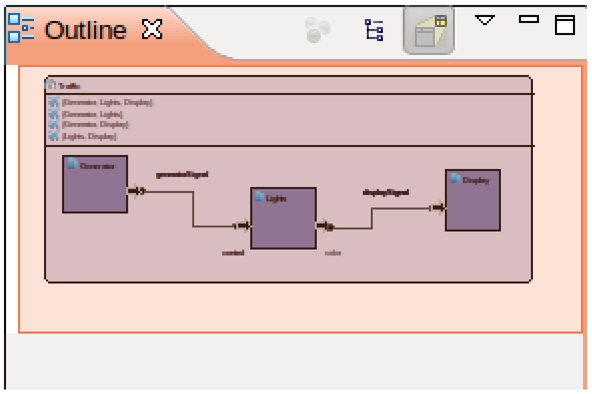


Figure 49. The Outline View

6.1.4. The Palette

The palette contains the tools for defining a model. Figure 50 shows the palette for the coupled model editor.

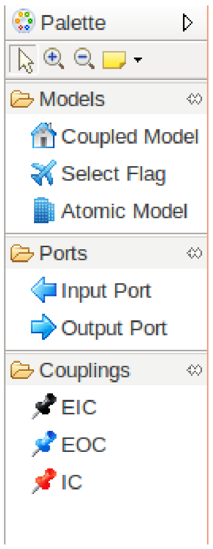


Figure 50. The Palette

The tools in the palette are Coupled Model (for creating a coupled model); Select Flag (for defining a select flag in the select flag compartment of a coupled model); Atomic Model (for creating an atomic model within the model compartment in a coupled model); Input and Output Ports. Ports are added by selecting a port and dropping it on a model (coupled or atomic). It also contains tools for EIC, EOC and IC. The connections are created by selecting the desired tool and drawing a connection between two model ports.

The upper part of the palette also contains tools for zooming (in and out), and adding a note to a model element.

6.1.5. The Drawing Workspace

The drawing workspace is where the model is created. Figure 51 shows the workspace. The modeler defines a model by picking tools from the palette and dropping them on the workspace. The figures in the workspace can be dragged and adjusted just as the modeler wants it. Only a coupled model can be dropped on the top level of the workspace. The coupled model has a compartment for holding select flags. To add a select flag, simply pick the Select Flag tool and drop on the select flag compartment on the coupled model. The coupled model also has a compartment for housing models. Coupled models and atomic models can be placed within a parent couple model by simply picking on the appropriate tool and dropping it on the coupled model's model compartment. Ports can be added to a model (coupled or atomic) by simply picking the appropriate tool (Input Port or Output Port) and dropping it on the model. Connections are made between ports. An External Input Coupling (EIC) can only couple the input port of a coupled model to the input port of its child model (atomic or coupled). An External Output Coupling (EOC) can only couple the output port of a child model and the output port of the parent coupled model. An Internal Coupling (IC) can only couple the output port of a model and the input port of another model within the same hierarchical level.

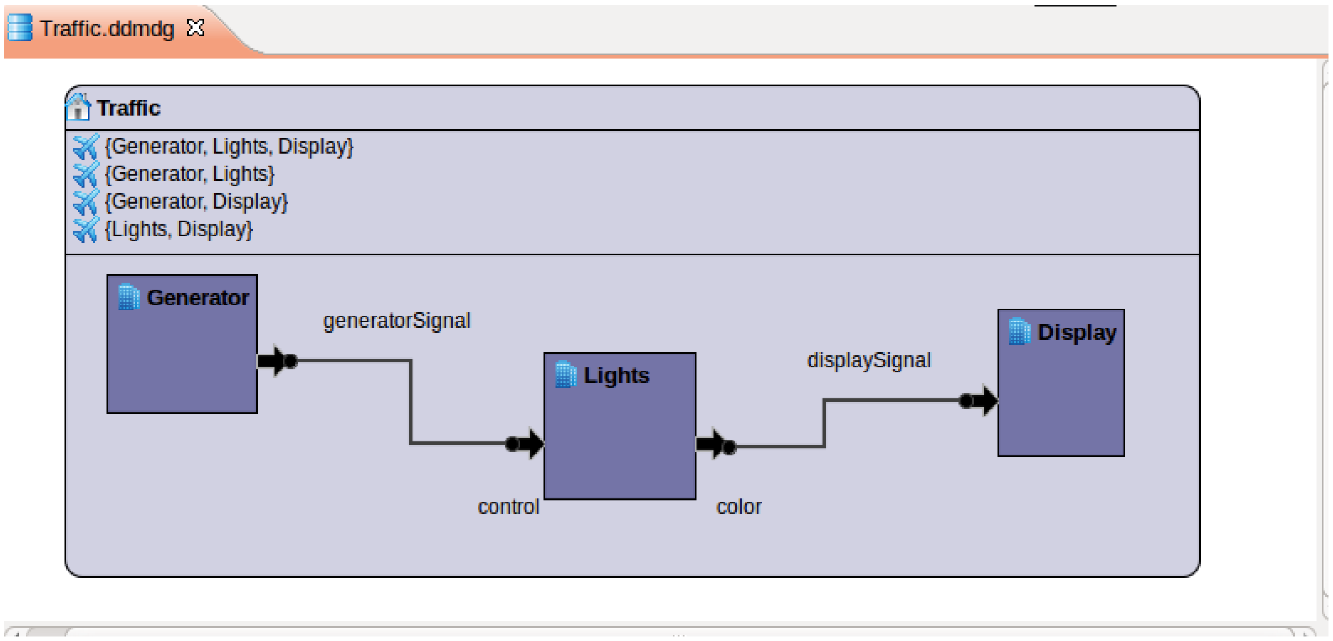


Figure 51. The Diagram Workspace

Right-clicking on the workspace (or a figure element in the workspace) launches a menu with several menu items (shown in Figure 52). From this menu, the modeler can add a note to the workspace or to the model diagram, print a model diagram, export the whole workspace (or a figure if it is selected) to an image file (jpg, png, gif, etc.), display the properties view for the workspace (or the selected figure), etcetera.

To export the workspace to an image file, simply right click on an empty space in the workspace, select File, and select Save As Image File (as shown in Figure 52 below).

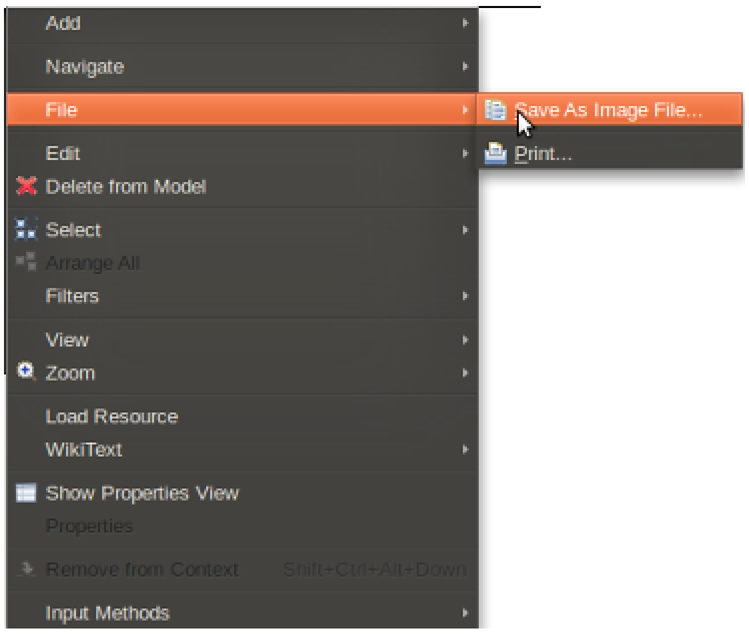
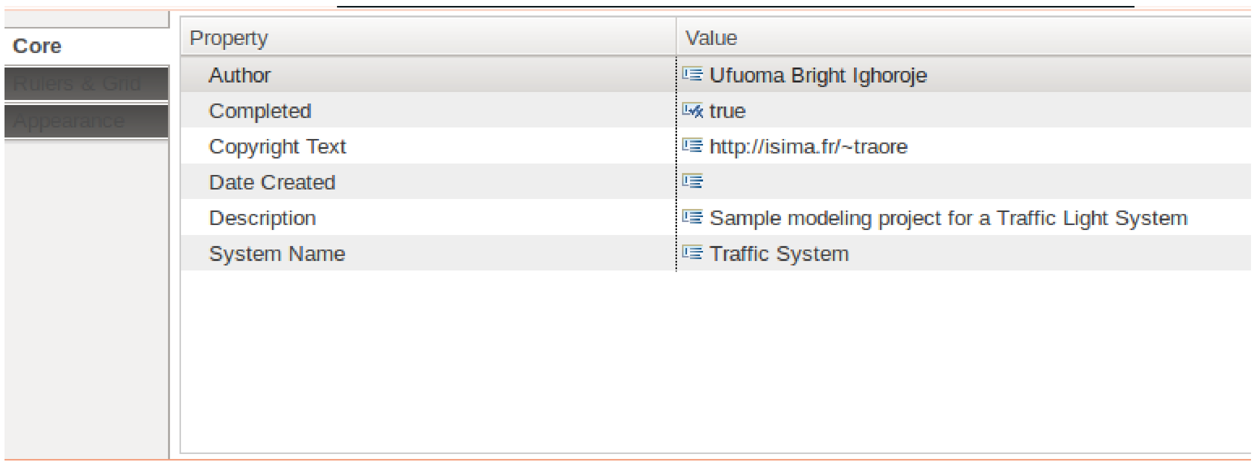


Figure 52. Exporting DDML Diagrams to Image Files

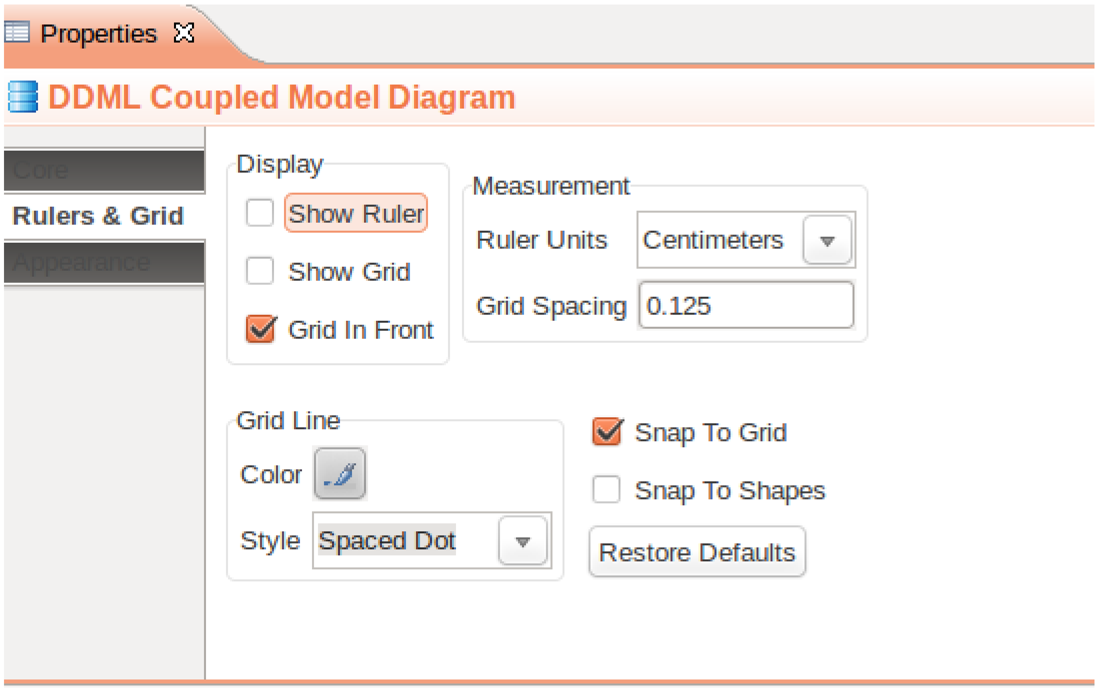
6.1.6. The Properties View

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. This view is very important to the modeler. The default user interface is table with property and value pairs, and the value being modified using a standard dialog cell editor. The properties shown depend on the element (workspace or model element) selected. Figures 53-56 shows the properties view for some selected model elements.



Property	Value
Author	Ufuoma Bright Ighoroje
Completed	true
Copyright Text	http://isima.fr/~traore
Date Created	
Description	Sample modeling project for a Traffic Light System
System Name	Traffic System

Figure 53. Project Property View



Properties

DDML Coupled Model Diagram

Core

Rulers & Grid

Appearance

Display

- [ ] Show Ruler
- [ ] Show Grid
- [✓] Grid In Front

Measurement

Ruler Units: Centimeters

Grid Spacing: 0.125

Grid Line

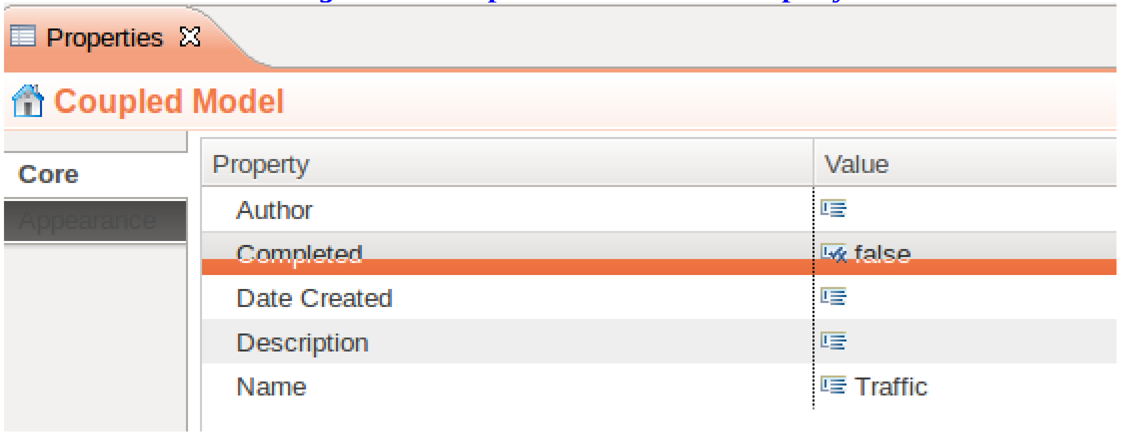
Color

Style: Spaced Dot

- [✓] Snap To Grid
- [ ] Snap To Shapes

Restore Defaults

Figure 54. Coupled Model Editor Property View



Properties

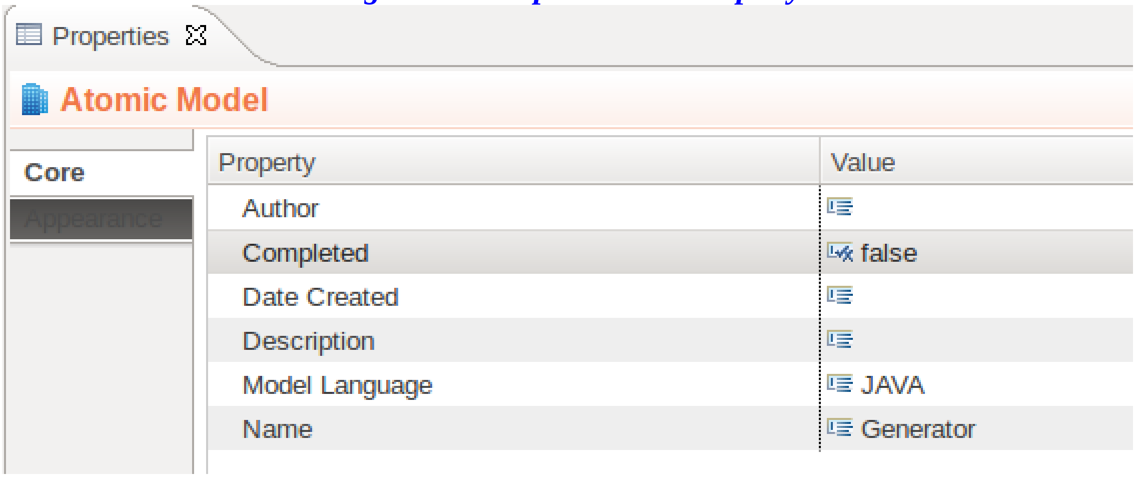
Coupled Model

Core

Appearance

Property	Value
Author	
Completed	false
Date Created	
Description	
Name	Traffic

Figure 55. Coupled Model Property View



Properties

Atomic Model

Core

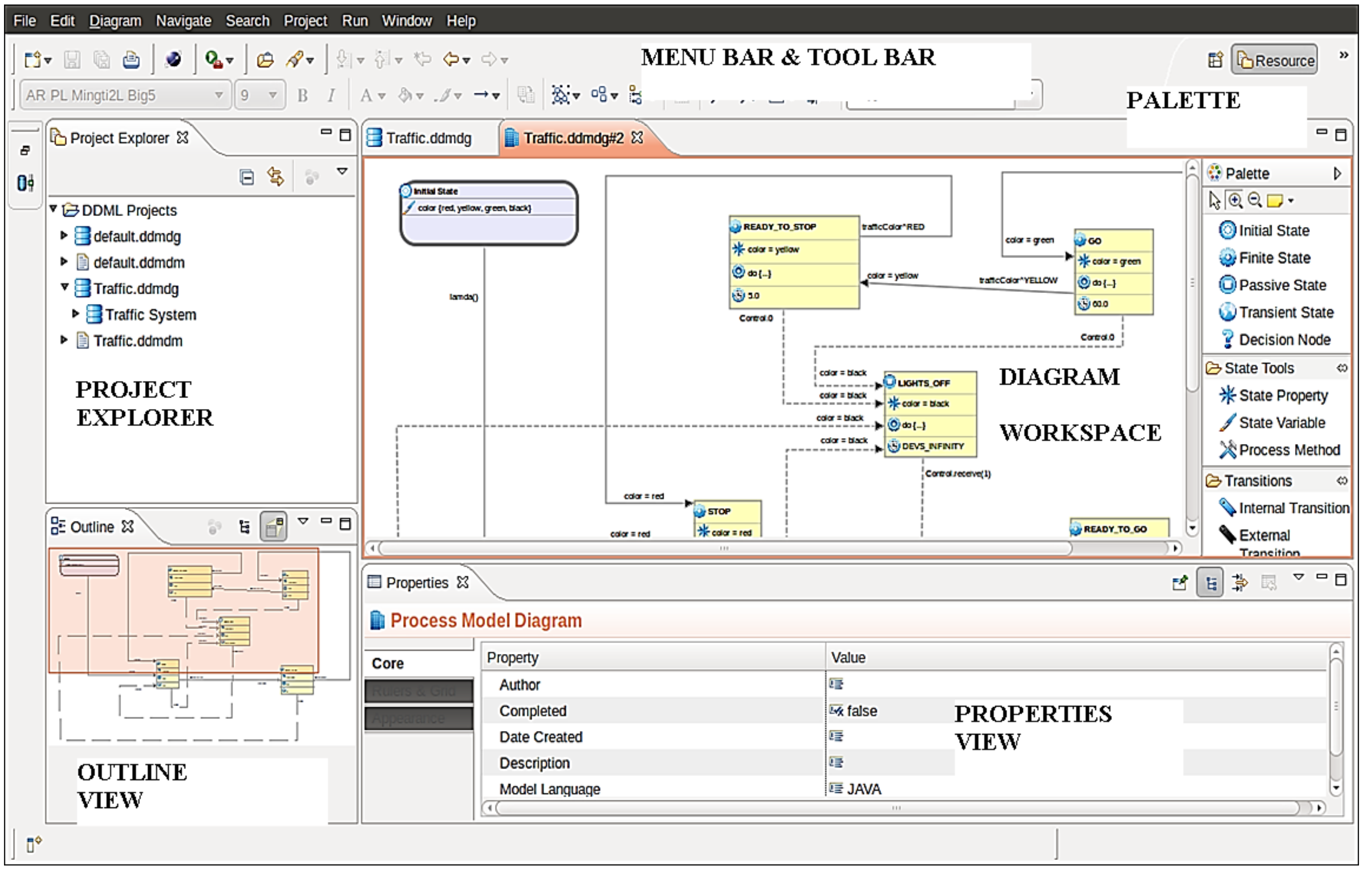
Appearance

Property	Value
Author	
Completed	false
Date Created	
Description	
Model Language	JAVA
Name	Generator

Figure 56. Atomic Model Property View

6.2. The DDML Atomic Model Editor

Figure 57 shows the workbench for the Atomic Model Editor.



Property	Value
Author	
Completed	false
Date Created	
Description	
Model Language	JAVA

Figure 57. DDML Atomic Model Editor Screenshot

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 58 shows the palette. The palette contains tools for States (Initial State, Finite State, Passive State and Transient State); State Tools (State Property, State Variable and Process Method); and Transitions (External Transitions and Internal Transitions).

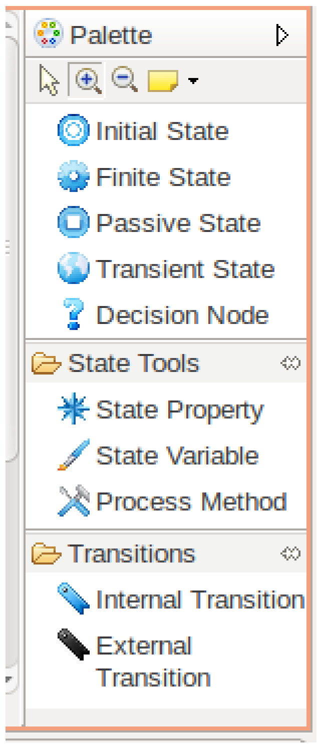


Figure 58. Atomic Model Editor Palette

Defining state transitions for the atomic model involves simply picking the right tools from the palate. The Initial State figure has compartments for State Variables and Process Methods. The State Variables define the states and can be added to the model by simply picking the State Variable tool and dropping it the Initial State (within the State Variables Compartment). A Process Method can be added in a similar way. A process method is a method (the body should be defined in the properties view) that is used within the atomic model. The body of the process method should be written in the language that has been predefined when creating the Atomic Model.

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, while that of the Finite State must be defined by the modeler.

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).

Chapter 7. Conclusion

We presented DDML as 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. We also presented a graphical editing tool with rich editors for defining DEVS coupled models and atomic models. Our tool, with drag and drop features can enable modelers to easily define, edit, share, and store models in a persistent form. DDML is a natural and intuitive approach to modeling. Its notation can easily be understood by both domain experts and modelers. Using a graphical editing tool further increases the simplicity.

DDML is a contribution towards making DEVS available to a wider community. It borrows concepts and ideas from very strong graphical formalisms (like UML and BPMN). 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.

Future work includes the following:

- DDML 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] Adegoke A. 2010. “**Efficient Object Oriented Implementations for the DEVS Formalism**”. M.Sc. Thesis, Computer Science Stream, African University of Science and Technology, Abuja, Nigeria.

[3] Christen, G., A. Dobniewski, and G. Wainer. 2004. “**Modeling state-based DEVS models CD++**”. Proceedings of MGA, Advanced Simulation Technologies Conference 2004, Arlington, VA, USA.

[4] Kidisyuk, K., and G. Wainer. 2007. “**CD++Modeler: A graphical viewer for DEVS models**”. Technical report SCE-017, Ottawa, ON, Canada.

[5] Matias, B.; G. Wainer; R. Castro; 2010. “**Advanced IDE for Modeling and Simulation of Discrete Event Systems**”. Proceedings of 2010 Symposium on Theory of Modeling and Simulation, DEVS’10. Orlando, FL. 2010.

[6] Nutaro, J. ADEVS. URL: http://www.ornl.gov/~1qn/adevs/index.html. Accessed: November 1, 2010.

[7] Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1996. “**Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally**”. Transactions of the SCS 13 (3): 135–154.

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

[9] Zeigler, B. P. 1990. “**Object-oriented simulation with hierarchical, modular models: Intelligent agents and endomorphic systems**”. Boston: Academic Press.

[10] Zeigler, B., and D. Kim. 1995. “**Extending the DEVS-scheme knowledge-based simulation environment for real-time event-based control**”. Technical report, Department of Electrical and Computer Engineering, University of Arizona.

[11] Sarjoughian, H. S., and B. P. Zeigler. 2000. “**DEVS and HLA: Complementary paradigms for M&S?**” Transactions of the SCS 17:187–197.

[12] Zeigler, B. P. 1999. “**Implementation of the DEVS formalism over the HLA/RTI: Problems and solutions**”. Simulation Interoperability Workshop, Orlando, FL.

[13] IEEE Std 1516.1-2000. 2001. IEEE standard for modeling and simulation. ”**High level architecture (HLA)—Federate interface specification**”. IEEE Std 1516.1-2000: 1–467.

[14] Sarjoughian, H. S., and B. P. Zeigler. 1998. “**DEVSJAVA: Basis for a DEVS-based collaborative M&S environment**”. Proceedings of SCS International Conference on Web-Based Modeling and Simulation, San Diego, CA.

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

[16] DÃ¡vila, J., and M. UzcÃ¡gegui. 2000. “**GALATEA: A multi-agent, simulation platform**”. Proceedings of International Conference on Modeling, Simulation and Neural Networks, MÃ©rida, Venezuela.

[17] Himmelspach, J., and A. Uhrmacher. 2004. “**A component-based simulation layer for JAMES**”. Proceedings of 18th Workshop on Parallel and Distributed Simulation (PADS), Kufstein, Austria, 115–122.

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

[19] Bolduc, J. S., and H. Vangheluwe. 2001. “**The modeling and simulation package PythonDEVS for classical hierarchical DEVS**”. Technical report MSDL-TR-2001-01, McGill University.

[20]. de Lara, J., and H. Vangheluwe. 2002. “**AToM3: A tool for multi-formalism and meta-modeling**”. Proceedings of Fundamental Approaches to Software Engineering, 5th International; Lecture Notes in Computer Science, 174–188.

[21] Praehofer, H., and G. Reisinger. 1995. “**Object-oriented realization of a parallel discrete event simulator**”. Technical report, Johannes Kepler University, Department of System Theory and Information Engineering.

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

[23] Gallardo G. et al. 2003. “**Eclipse in Action: A Guide for Java Developers**”. Manning Publications Co.

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

[25] Moore B. et al. 2004. “**Eclipse Development: Using the Graphical Editing Framework and the Eclipse Modeling Framework**”. IBM Redbooks.

[26] Gronback R. C. 2009. “**Eclipse Modeling Project: A Domain-Specific Language Toolkit**”. Addison-Wesley.