Rule Based Operational Semantics Specification in Ptolemy

Yanwar Asrigo

McGill University, School of Computer Science
yanwar.asrigo@mail.mcgill.ca

Abstract. As systems are getting more complex to design and build, people have realized the benefits of building models as part of the development. Many modeling tools, like Ptolemy, supports the model-simulate-synthesize of systems to accelerate the development cycle. As the use of models are getting more popular, people needs to create domain specific notations to lower the learning curve for non-programmers. This drives the demand for modeling tools with meta-modeling capability. Unfortunately, this feature is not provided in Ptolemy. In this project, we investigate the possibility of using other modeling tool that supports meta-modeling to provide such capability. More specifically, we are particularly interested in automatically generating operational semantics of a formalism which is specified using transformation rules and compile these rules into codes that can be integrated into Ptolemy.

1 Introduction

The role of modeling tools is becoming more and more important in designing a system. As systems are getting more complex, it has become quite impossible to implement them based on the specification directly. With the help of proper modeling tools, people can easily model, simulate, and synthesize complex systems. This not only accelerates development cycle, but also reduces development cost and risk as errors can be dealt with earlier. For critical systems, models are usually used for verifying system correctness and safety. Moreover, systems that are code-generated from models tend to be more flexible and have less error as modification of the system specification can be done easily at the model level even by non-programmer person. In this respect, Ptolemy[1] is an excellent tool for modeling, simulation, and design of concurrent, real-time, and embedded systems.

An important factor in modeling a system is selecting the right formalism and the right level of abstraction. This decision is determined by the system designers intention. For example, finite state machines (FSM) have long been used to describe and analyze intricate control sequences. However, not every problem domain has the corresponding appropriate formalism because there is no such formalism yet or the available formalisms are simply too powerful. This has driven the demand for modeling tools that support *meta-modeling*, which allows users to rapidly create their own domain specific notations. Unfortunately,

Ptolemy does not have this capability yet. All the formalisms supported by Ptolemy are hand-coded by writing the Java code. In this project, we investigate the possibility of using other modeling tool which has meta-modeling capability to automatically synthesize codes to create new formalisms in Ptolemy.

Section 2 describes the problem that we investigate in more details. Section 3 introduces the high level design of Ptolemy. Section 4 discusses the finite state machine formalism in Ptolemy. The implementation of the project is outlined in Section 5. Subsequently, Section 6 gives the testing result of the implementation. Finally, Section 7 draws the conclusion and describes some possible future works.

2 Problem Definition

In this project, we investigate the possibility of using the meta-modeling environment of other modeling tool and automatically synthesize the relevant codes to realize the domain specific formalism in Ptolemy. Figure 1 depicts the high level process for creating a formalism from meta-model. We choose to meta-model Finite State Machine as the target formalism because its operational semantics can be easily described using rule-based specification. AToM^3[2] is used for the meta-modeling and rule-specification environment. In addition, we also used the graph grammar compiler of Motif[3] to extract and compile the rules specified in AToM^3. These tools are selected due to our familiarity with the tools and accessibility of the source code.

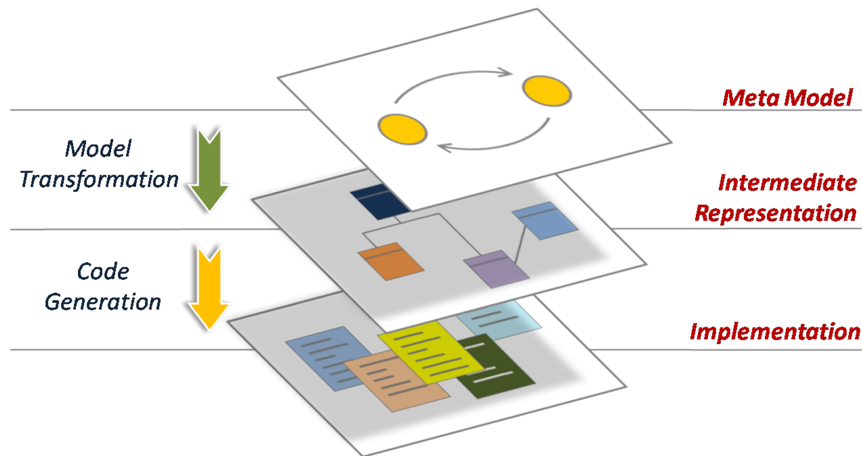


Fig. 1. High level process for realizing a formalism from meta-model level to implementation.

Meta-models in AToM^3 are usually modeled using the *ClassDiagramV3* formalism which has high correlation with the UML class diagram notation that

is commonly used in software system design. We believe transforming meta-model described using *ClassDiagramV3* to UML class diagram can be done quite straight forward. Therefore, the main focus of this project is to create an automatic process for transforming the rule based operational semantics specification to Java code that can be embedded easily in Ptolemys environment.

3 Ptolemy's High Level Design

Ptolemy is Java based modeling tool developed in UC Berkeley. It is designed to address the challenges of heterogeneous composition by introducing a more structured approach to compose different models, i.e. *hierarchically heterogeneous*. This means that complex models are hierarchically decomposed where each level of decomposition contains a network of interacting components. Each level serves as a refinement for a component from the layer above. Each local network is governed by a single interaction mechanism but different interaction mechanisms can be specified at different levels in the hierarchy.

A fundamental concept of hierarchical heterogeneity is the clear separation of flow of data and the flow of control in defining components interaction in a local network. Such a framework is called a *model of computation* (MoC) as it attaches a semantic to a network structure by defining how computation takes places among computational components in the structure. On the other hand, MoC must be compositional as well to support hierarchical composition. MoC must be able to encapsulate the network of components it described into a component that may then be composed again with other components, possibly under a different MoC.

Ptolemy adopts an *actor-oriented* view of a system, where actor forms the basic building block of a system. *Actors* are concurrent components that communicate through interfaces called ports by passing messages. Interconnection between these ports forms the communication structure between actors. Contrast with object orientation whereby control flows through an object in a sequential manner through method calls, actor orientation emphasizes on the flow of stream of data into and out of every actor. Every actor encapsulates an abstract functionality. Actor orientation enables the decoupling of the transmission of data from the transfer of control in Prolemy.

To support the hierarchical composition, an actor can be *atomic*, which is a component at the bottom of the hierarchy, or *composite*, which aggregates other actors. In the hierarchical model illustrated in Fig. 2, the top level composite actor contains actors A1 and A2. Actor A2 is also a composite actor which contains actors A3 and A4. Actors A1, A3, and A4 are atomic actors. In Ptolemy, actors atomic executions (called *iterations*) as depicted in Fig. 3 consist of 3 phases: *prefire*, *fire*, and *postfire*. This is to provide the flexibility to MoC of a composite actor in managing the relation of actors iteration in the same composite. The *prefire* phase checks the preconditions for the actor to execute. The *fire* phase is when an actor typically performs its computation. However, the persistent state of the actor is updated only in the *postfire* phase.

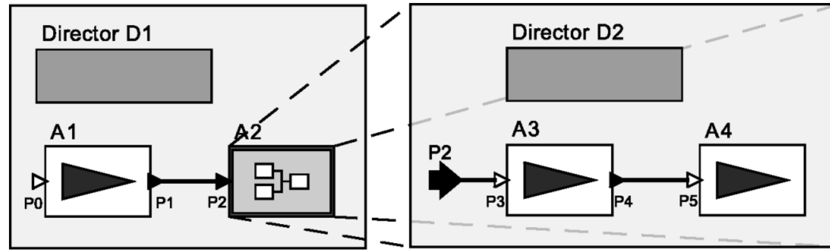


Fig. 2. An example of Hierarchical Model in Ptolemy. Taken from [1].

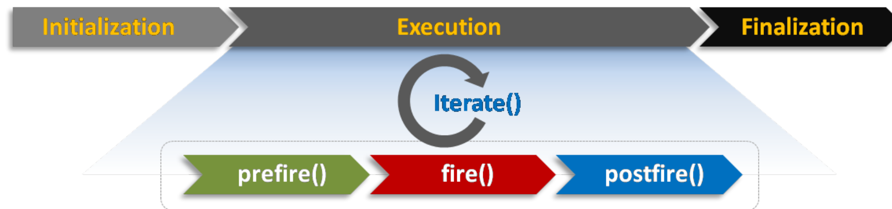


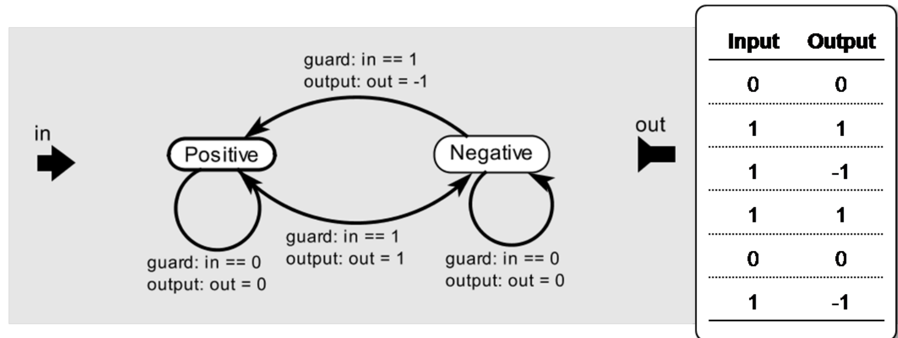
Fig. 3. Ptolemy Actor execution cycle.

The syntactic structure of actors does not have any semantic until it is associated with a MoC. In Ptolemy, an implementation of a MoC associated with a composite actor is called a *domain*. A domain consists of two classes: a *director* and a *receiver* class. The receivers implement the communication mechanisms used in the domain. Note that receivers are contained in input ports and there is only one receiver assigned for each communication channel. When an actor is assigned to a domain, it obtains the domain specific receivers at its input ports. A director is responsible for managing the execution order of the actors contained in a composite. In the Fig. 2 example, it is a hierarchical model using two different domains. The top-level composite actor is controlled by director D1, and A2 is a composite actor with director D2. Director D1 controls the execution order of actors A1 and A2, and director D2 controls the execution of A3 and A4 when A2 is executed. This separation of computation and communication allows actors to be reused in different domains (called *domain-polymorphisms*).

4 Finite State Machine in Ptolemy

Finite State Machine (FSM) in Ptolemy are mealy machines. Figure 4 depicts an example of finite state machine in Ptolemy. Following the definition, the FSM in Ptolemy consists of *states* and *transitions*. A state can be designated as initial state, final state, or both at the same time. There has to be an initial state in an FSM. While transitions are normally triggered by an incoming *event*, Ptolemy uses guard expression instead. This expression can be used to evaluate not only the incoming event but also any variable accessible by that transition. Ptolemy

also supports non-deterministic automata (NDFA). Since Ptolemy supports hierarchically heterogenous decomposition, FSM in Ptolemy has richer semantics than *statecharts*. Although as described in Section 2 that Ptolemy has the notion of actor and domain, FSM formalism inherently does not fit nicely in this mold. The semantics of FSM is actually built in the FSM Actor.



Input	Output
0	0
1	1
1	-1
1	1
0	0
1	-1

Fig. 4. Alternate Mark Inversion (AMI) Coder FSM representation in Ptolemy and a sample of input and output stream for this model.

For this project, the FSM implementation follows closely the standard FSM semantics, i.e. a transition is only enabled by incoming event instead of the more flexible guard expression. To minimize modification, our implementation will reuse the guard field to let user specify the input event. In addition, all transitions are deterministic. Moreover, it does not support hierarchical composition of the states. Figure. 5 shows the AMI coder modeled using the new semantics.

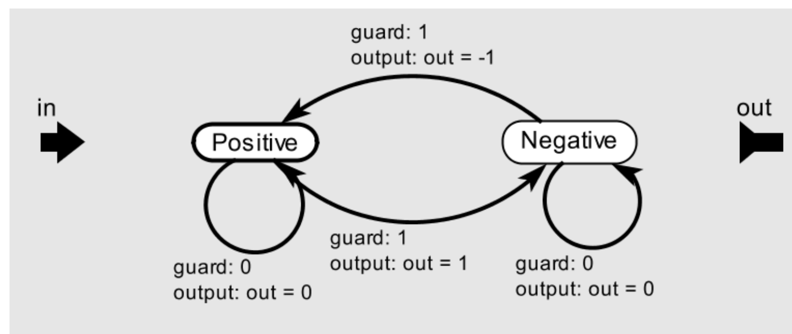


Fig. 5. AMI coder representation using the new FSM implementation.

5 Implementation

Figure. 6 outlines the main steps required in specifying and compiling the operational semantics rule in AToM^3. The process begins with meta-modeling the domain specific notation so that even non-programmers are able to specify the transformation rules that describe the operational semantics. In AToM^3, transformation rule is represented as graph grammar specified by the left-hand-side (LHS) and right-hand-side (RHS) graphs. Users may also specify additional condition in the transformation rule used for matching the LHS graph. The code generated by the rule compiler could be inserted directly in Ptolemy Java project. To support this, we also make some modification in Ptolemy codes. The following sections describe in details our implementations steps.

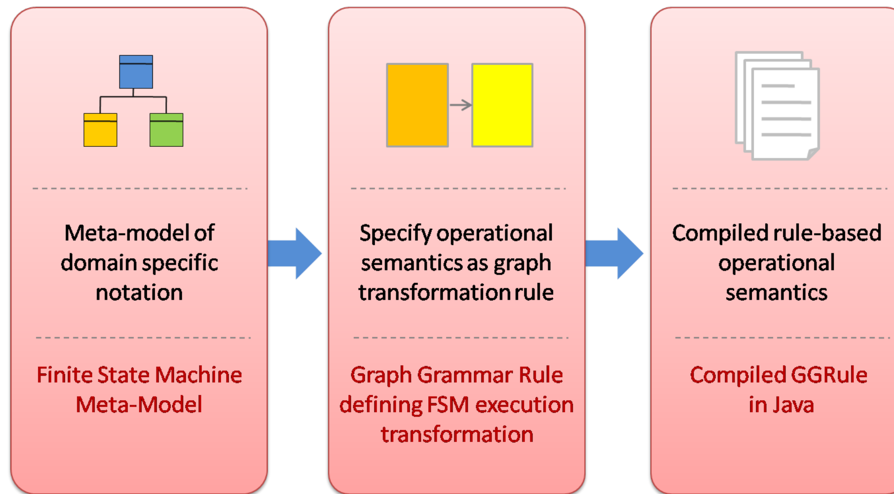


Fig. 6. Specifying and compiling operational semantics rules using AToM^3.

5.1 Meta-Modeling Domain Specific Notation

The meta-model of our FSM is shown in Fig. 7. The meta-model is created using ClassDiagramV3 formalism of AToM^3. *State* is represented as a class and *transition* as a relation that connects 2 states. The *Current* class is introduced for tracking the current active state in the simulation. It is linked to active state by *Points_To* relation.

5.2 Specifying Operational Semantics Using Graph Transformation Rules

We utilize the graph transformation facility of AToM^3 for specifying the transformation rules of FSM operational semantics. As mentioned earlier, each graph

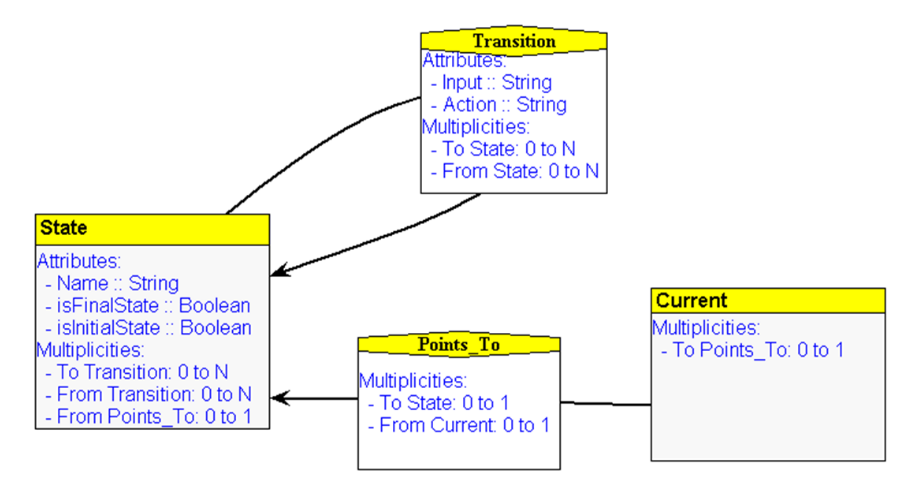


Fig. 7. Finite state machine meta-model in AToM^3 using *ClassDiagramV3*.

grammar in AToM^3 is specified as LHS and RHS. Additional matching condition and final action can also be specified at the rule level. We found that two transformation rules are adequate to describe FSM operational semantics. The first rule (shown in Fig. 8) is used in the beginning of FSM simulation to find the initial state in the model and set it as the current state. The second rule (shown in Fig. 9) is used to update the model when there is an input event in the FSM. The constraint of a transition only activate when its trigger event is received is specified as condition at rule level. Similarly, the code for the emission of transition output is included as the rule final action. With reference to Fig. 9, Motif allows a state matched with the state 1 in LHS to be matched again with state 2. Hence, this update rule can also be used to detect self-loop transition.

5.3 Compiling the Graph Transformation Rules

We reuse the graph grammar compiler from Motif to extract and compile the graph transformation rules defined in AToM^3. The pseudo-code of rules generated by Motif can be found in Fig. 10. In this project, Motif graph grammar compiler is modified to generate Java code instead of Python. The generated Java code also needs to follow the mechanism of getting and setting attributes value in Ptolemy. A Java class is created to encapsulate each transformation rule. In this case, the rule used for FSM initialization is compiled to *InitRule* class and, similarly, the update rule is compiled to *UpdateRule* class.

Motif embeds the code for LHS subgraph matching and the graph rewriting logic of each rule directly in the generated code. In other words, transformation rule execution on a given abstract syntax graph(ASG) input will directly returns the transformed graph if the rule execution is successful. This notion does not exactly correspond to the actor execution cycle in Ptolemy which only updates

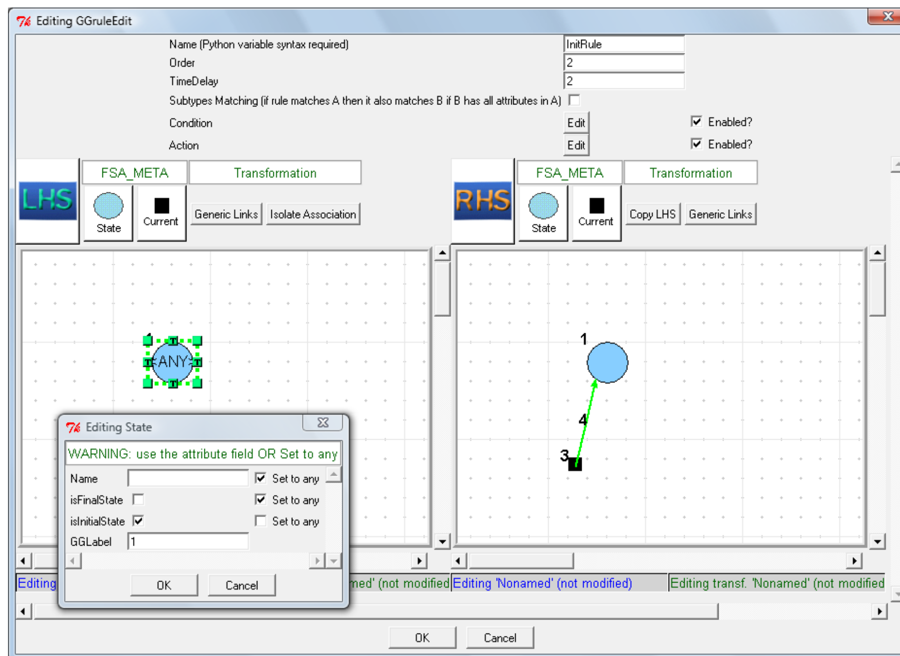


Fig. 8. Transformation rule for finding initial state and set it as current state.

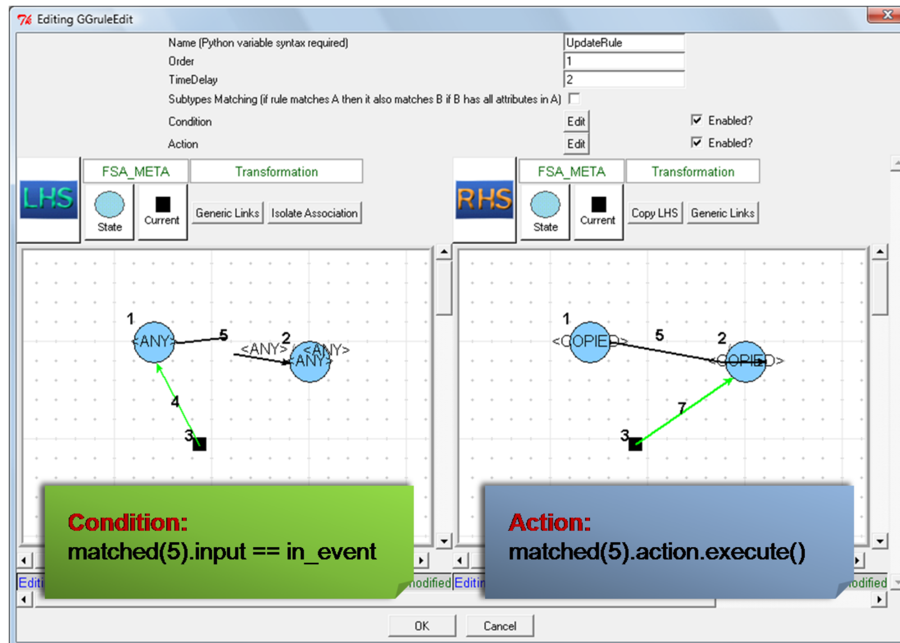


Fig. 9. Transformation rule for updating the FSM given an input event.

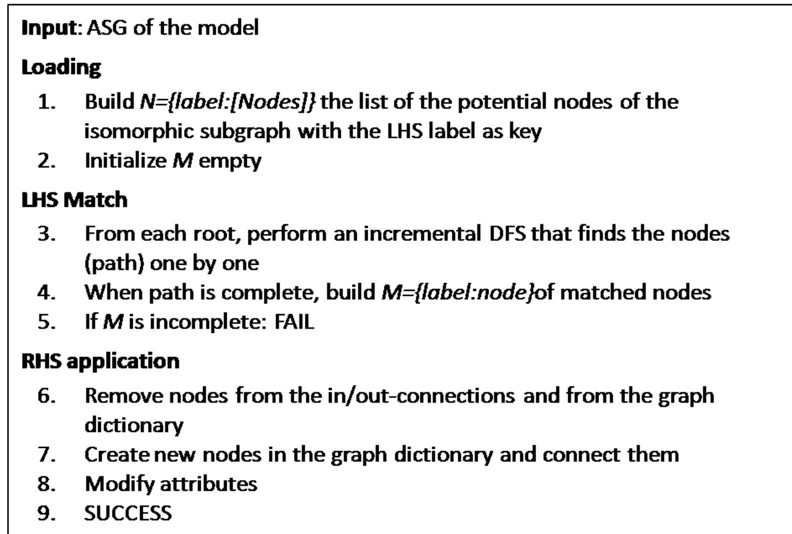


Fig. 10. Transformation rule for updating the FSM given an input event.

state information during *postfire* phase despite the computation occurs during *fire* phase. As a result, the execute() method in the generated graph grammar class is splitted into two methods, namely *executeLHS()*, which contains the logic for the LHS matching, and *executeRHS()*, which contains the RHS application of the transformation rule. The class diagram of rules generated by the graph grammar compiler is shown in Fig. 11.

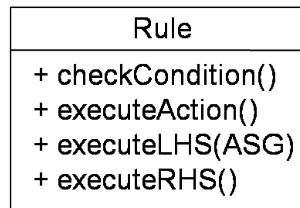


Fig. 11. Class diagram of rules generated by the graph grammar compiler.

The graph grammar compiler is also enhanced to extract the condition and action set at the rule level (refer to Fig. 9) defined in AToM^3 so that they are executed as well in the generated rule. In addition, a bug related to the creation of connection between two new objects specified in RHS is also fixed. For example, with reference to Fig. 8, the original compiler generates a code that cause *Current* node (label 5 in RHS) stored twice in Points_To's (label 3 in RHS) *in_connection* list.

5.4 Modifications in Ptolemy

Rules generated by Motif operate on the ASG representation that is used in AToM^3. This representation is rather different from the representation used in Ptolemy. To bridge this gap, Ptolemy FSM domain is modified to include additional data structures that are used in AToM^3. Figure 12 shows the class diagram of classes that are created to mimic the graph representation seen in AToM^3. The ASG class is used to store the graph representation. The nodes in the ASG are of type ASGNode. ASGNode is defined as an interface which is implemented by all the object classes used in the domain specific notation, i.e. *State*, *Transition*, *Current*, and *Points_To*. Note that, *State* and *Transition* are actually existing Ptolemy classes in the FSM domain. Class Current and Points_To are created to match the class diagram used in the meta-model (refer to Fig. 7).

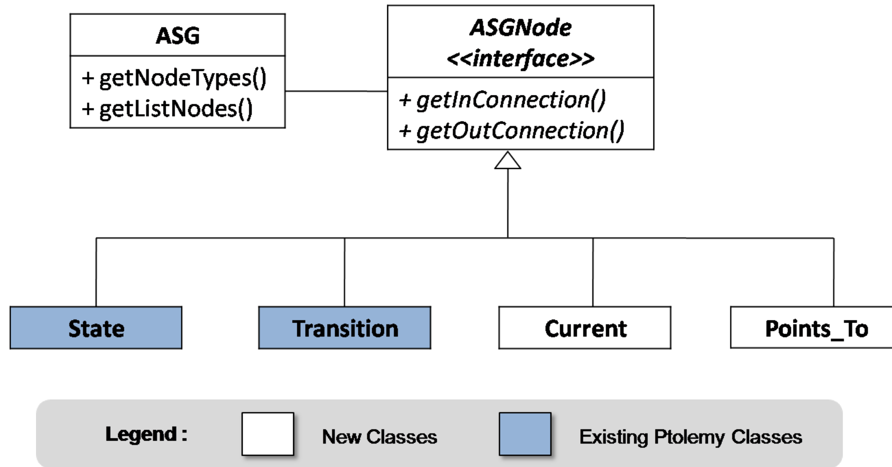


Fig. 12. Class diagram of rules generated by the graph grammar compiler.

The code for FSM Actor is also altered so that it calls the methods of *InitRule* and *UpdateRule* instead of executing original hand-coded logic. *InitRule* is executed in the initialization phase of FSM Actor. As discussed in the previous section, the FSM Actor *fire()* is modified to invoke the *executeLHS()* of *UpdateRule* and the *postfire()* is modified to invoke *executeRHS()*. Since both rules are used in different phases of the execution cycle, we do not need to consider the priority level of the rules. Figure 13 depicts the sequence diagram of the modified FSM Actor.

While the compiled rules operate on the AToM^3-like graph representation, FSM Actor still maintain its own graph structure to ensure consistency with the rest of Ptolemy components such as the user interface. As a result, FSM Actor needs to keep both representations. An adapter is created to reconcile the changes made by the transformation rules on the AToM^3-like graph with the

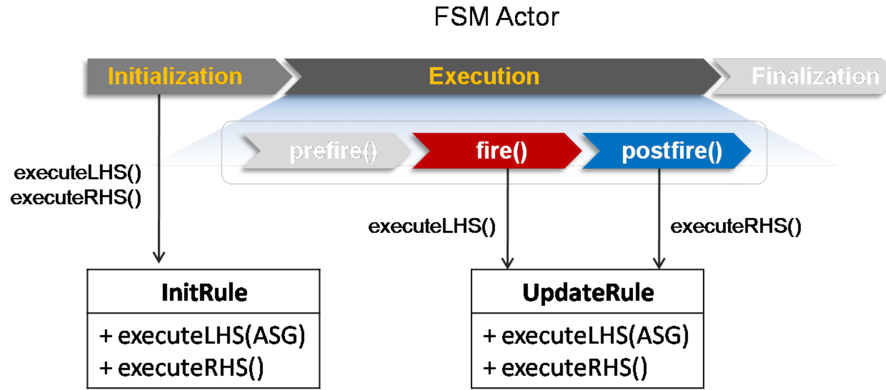


Fig. 13. Sequence diagram of the FSM Actor executing InitRule and UpdateRule.

Ptolemy's internal graph representation. The AToM^3-like graph is created only once during the initialization phase of FSM Actor instead of creating it every time the compiled rules get executed.

6 Testing

We setup the environment depicted in Fig. 14 to conduct our testing. It consists of a pulse generator that generates the input sequence described in Fig. 4. The AMI Coder contains the finite state machine shown in Fig. 5. This example is chosen as test case because it covers both state-to-state transitions and self-loop transitions.

The execution result of the AMI Coder is recorded in Fig. 15.The output obtained from the AMI Coder is as expected (see Fig. 4). It is also verified that the original Ptolemy code produces identical result.

7 Conclusion

In this project, we have successfully demonstrated that other modeling tool can be used to provide meta-modeling capability for Ptolemy. Since we use AToM^3 for the meta-modeling environment, generating UML class diagram from the meta-model can be considered straight forward. This meta-model can then be used to specify the formalism's operational semantics using transformation rules. The Motif graph grammar compiler has been modified for this project to extract the transformation rules and compiled it into Java code which can be run in Ptolemy. A minor modification has been done in Ptolemy so that the generated code can be easily integrated in it. To illustrate our approach, we meta-modeled finite state machine formalism, compiled its operational semantics rule into Java classes, and successfully integrated them into Ptolemy. The main difficulty we

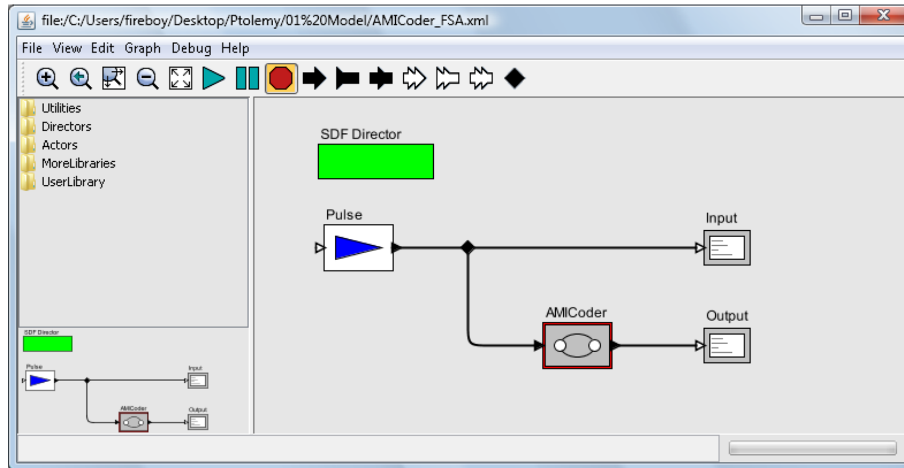


Fig. 14. AMI Coder testing environment.

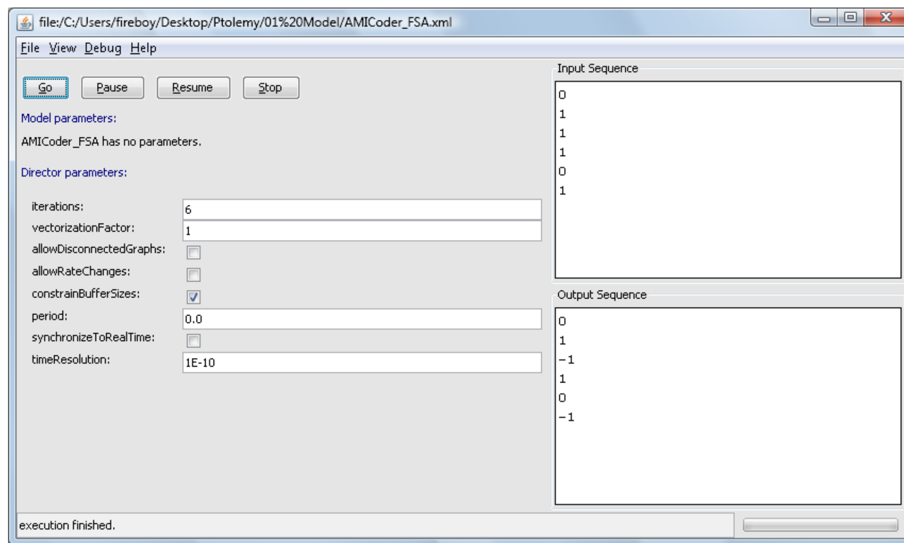


Fig. 15. AMI Coder simulation result.

encountered is bridging the mismatch of Ptolemy's and AToM^3 infrastructure. Our approach is to keep two representation of the same model. However, this might caused consistency issue when the for more complex formalism and graph gets bigger. Hence, as a possible future work, the graph grammar compiler can be tweaked further to produce code that can work direcly on Ptolemy's data structure. Another possible future work would be investigating the possibility of meta-modeling formalism which operational semanctics can not be specified using transformation rule.

References

1. Eker, J., Janneck, J., Lee, E., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the ptolemy approach. Proceedings of the IEEE **91**(1) (Jan 2003) 127–144
2. de Lara, J., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering, London, UK, Springer-Verlag (2002) 174–188
3. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with devs. In Nagl, M., Schr, A., eds.: Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007). Spriger-Verlag, Kassel, Germany. (October 2007)