Transforming Graph Grammar Specification of FSM to Kermeta

Jun Li

School of Computer Science, McGill University
jun.li3@mail.mcgill.ca

Abstract. Kermeta [1] is an executable meta-modeling language, which is designed to define both structures and behaviors of meta-models. However, it lacks of a graphical modeling environment to model the operational semantics. Interactive tool AToM^3 [3, 4] uses graph grammar model to describe the behavior of meta-model. In this project, we explore a process of using AToM^3's graph grammar model to define the operational semantics of Kermeta meta-models, then translate them into Kermeta programs, which contain the same functionalities of hand coded programs. We demonstrate this process by a Finite State Automata example.

1 Introduction

Kermeta [1] is an executable meta-modeling language, which is designed to define both structures and behaviors of meta-models. It is a model-oriented language, compliant with the OMG Essential Meta-Object Facility (EMOF) [9] and Ecore of Eclipse Modeling Framework (EMF) [8]. Kermeta also provides tools for Transforming its meta-models from and to EMOF and Ecore meta-models. Kermeta can be used to precisely specify the operational semantics of meta-mode, including static semantic, similar to Object Constraint Language (OCL), and dynamic semantic defined as operations of the meta-model.

As a programming language, Kermeta is an imperative, object-oriented language. It also integrates aspect-oriented features, and supports some design-by-contract features, such as pre and post conditions, and invariants [2].

As an action language, Kermeta is powerful for specifying behaviors of meta-model. However, one of the drawbacks of Kermeta is that it lacks of a graphical modeling environment to model the operational semantics. The implementations for specifying the operational semantic need to be coded by hand.

AToM^3 [3, 4], "A Tool for Multi-formalism and Meta-Modelling", provides a graphic way to define the operational semantics of meta-models. In AToM^3, the operational semantics can be defined as graph grammar models, each of which consists of a set of rules. Those graph grammar models are widely used for graph transformations by graph rewriting systems. Those models can be compiled into classes files used by programmed graph transformation systems [7].

In this project, we explore a process of using AToM^3's graph grammar model to define the operational semantics of Kermeta meta-models, and translate them

into Kermeta programs, which contain the same functionalities of hand coded programs. We demonstrate this process by a Finite State Automata example.

2 From Kermeta to AToM^3

2.1 Overview of Kermeta Meta-modelling

Kermeta meta-model consists of two parts, the structure of the meta-model defined by a class diagram and the operational semantic specified by an executable model-program. The class diagram models the meta-model, which defines the elements of modeling formalisms and their relationship constraints. It is similar to the Entity-Relationship formalism used for modeling meta-model in AToM^3.

The class diagram is compliant with EMOF and Ecore; stored in .km (kermeta program in xmi format), .kmt (Kermeta program in textual syntax), or .kmdi (used for graphic editing) files. The operational semantics of the meta-model is store in a .kmt file, which is an executable program written in Kermeta language [1].

Fig. 1 shows the structure of the Kermeta meta-modeling system. The meta-model (bottom-left block) is prototyped by class diagram. The class diagram can be translate into Kermeta class program (middle-middle), which contains definition of classes, their attributes and operations (similar to methods). Then users can construct model (top-left block) based on the meta-model. When doing simulation, user constructed model will be loaded by using Eclipse Modelling Framework (EMF) functions into system as instance of the meta-model, which composed by the instances of classes defined in the class diagram.

The relations between the classes and their attributes are also defined in the class diagram; however, they are not translated into the class program. They implicitly exist and are required when using the data of model instance. Here we call them as relation program, shown in center-bottom of the figure. The operation semantic program (right-bottom block) is a hand-coded program, which consists of implementations of operations, pre and post conditions, and invariants. It has all the functionalities to traverse the graph, update the graph data, and ask user input, and doing the simulation. A Kermeta program usual consists of those three parts, class program, relation program, and semantic program.

Simulating the user created model is starting from an extra launching program, written in Kermeta. The launching program first loads the user model, then passes the model instance to Kermeta program and activates, the simulation function of that program.

2.2 Differences Between AToM^3 and Kermeta

A Kermeta meta-model is described by a UML class diagram, which defines the elements of the meta-modeling formalism and their relationship. The internal data structure, including relationship between classes and their attributes, is also defined in the same class diagram. Since in the class diagram, users can

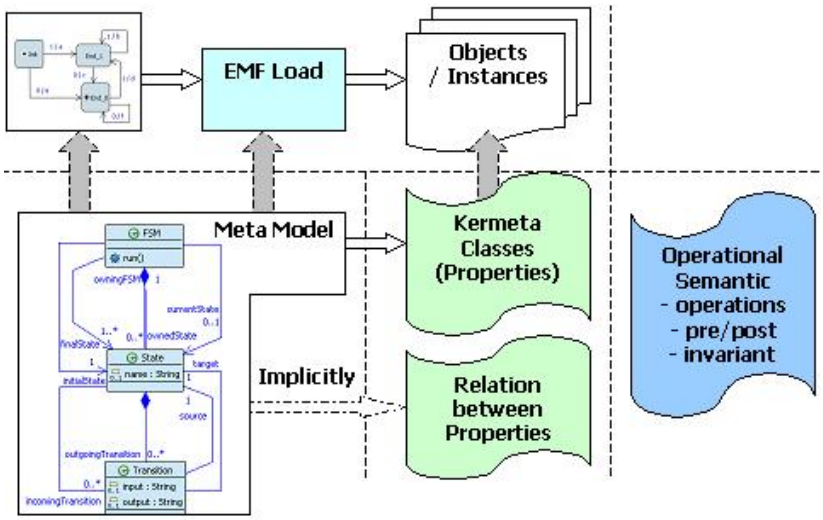

Fig. 1. FSA Model Internal Data

define classes and attributes the way they want, the Kermeta model system loses the control of unified data structure. Different meta-models, if they have different classes and attributes relations, then will have different data structures.

AtoM^3 uses Entity-Relationship formalism extended with constraints to model the meta-model [4], and all the models and graph grammars are internally represented as same data structure, Abstract Syntax Graphs (ASGs). This unified data structure becomes major advantage for AtoM^3 when creating graph grammar compiler and doing model simulation.

In this project, we uses Finite State Automata (FSA) model defined by Vangheluwe & de Lara [5] as example. For comparison purpose, we lists the meta-model and model in both systems. Fig. 2 shows the FSA meta-model defined in AToM^3. Fig. 3 shows the even binary number recognizer FSA model constructed by in AToM^3. The corresponding meta-model and FSA model in Kermeta are shown in Fig. 4 and Fig. 5.

The differences between two systems:

- 1. In AToM^3 meta-model, there is extra object 'current' and relation 'points_to', which are designed for graphic simulation purpose. Corresponding Kermeta model uses attribute 'currentState' of class 'FSM' to store that information. Because Kermeta only provide text model simulation, putting it as internal data with no graphic notation in the target model still meets its simulation requirement.
- 2. In the user constructed the even binary number recognizer FSA model, there is not big difference in the graphs. But the data structures behind

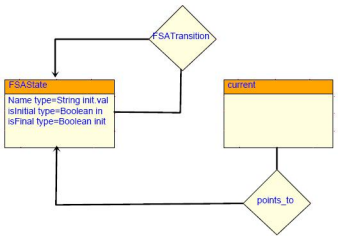

Fig. 2. FSA Meta-Model (ER+Constrants) [5]

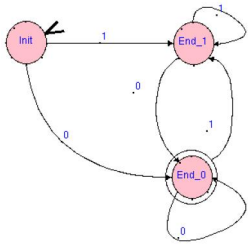

Fig. 3. Even Binary Number Recognizer FSA Model [5]

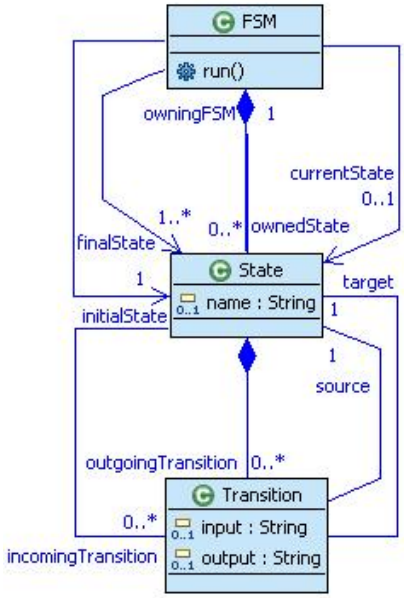

Fig. 4. Kermeta FSA Meta-Model

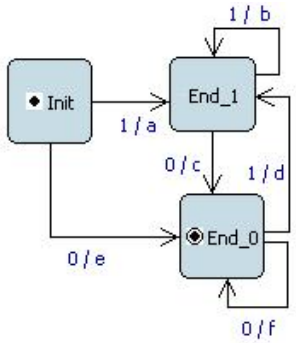

Fig. 5. Kermeta FSA Model, Even Binary Number Recognizer Model

the scene are totally different. We will detail compare them in the following subsection.

2.3 Internal Data Structure of Kermeta Meta-Model

The class diagram defines the internal data structure of meta-model. Fig. 6 shows the class program translated from the FSA class diagram (Fig. 4). Fig. 7 shows the internal data of the user created FSA model (Fig. 5), which lists all the States, Transitions objects and their relations. To get that information. Program needs to use the implicit data relationship defined in the class diagram.

The implicit data relationship of this FSA meta-model:

- 1. The instance of user created FSA model is a *FSM* object.
- 2. All the states in the FSA model are stored in the *FSM*'s array attribute *ownedState*.
- 3. Each state has two sets of transitions. According to the direction of the transition, they are stored in two array attributes, *outgoingTransition* and *incomingTransition*.
- 4. Each transition has attributes *source* and *target* to store source State and target State object it links to.

According to these relation information, Fig. 8 is an example Kermeta program fragment used to traverse the FSA model and produce previous output.

3 Graph Grammar Compiler for Kermeta

3.1 Graph Grammar Compiler for Kermeta

Graph Grammar model is used to describe model transformations, thus defines the operational semantic of a meta-model. FSA simulator model grammar rules,

```
package fsm;
class FSM
{
    attribute ownedState : State[0..*]#owningFSM
    reference initialState : State[1..1]
    reference finalState : State[1..*]
    reference currentState : State
    @overloadable "true"
    operation run() : kermeta::standard::~Void is
        raise kermeta::exceptions::NotImplementedException.new
}
class State
{
    reference owningFSM : FSM[1..1]#ownedState
    attribute name : String
    attribute outgoingTransition : Transition[0..*]#source
    reference incomingTransition : Transition[0..*]#target
}
class Transition
{
    reference source : State[1..1]#outgoingTransition
    reference target : State[1..1]#incomingTransition
    attribute input : String
    attribute output : String
}
```

Fig. 6. FSA Model Class Program

```
State : Init
  Transition : Init-(0/e)->End_0
  Transition : Init-(1/a)->End_1
State : End_0
  Transition : End_0-(0/f)->End_0
  Transition : End_0-(1/d)->End_1
State : End_1
  Transition : End_1-(1/ b)->End_1
  Transition : End_1-(0/c)->End_0
```

Fig. 7. FSA Model Internal Data

```
fsm.ownedState.each
{ s |
    stdio.writeln("State : " + s.name)
    s.outgoingTransition.each { t |
       stdio.writeln("  Transition : " + t.source.name
       + "-(" + t.input + "/" +  t.output  + ")->"
       + t.target.name)
}   }
```

Fig. 8. FSA Model Traverse Program

defined by Vangheluwe & de Lara [5], is shown in Fig. 9, (with adjustments for this project). A graph grammar compiler for Kermeta is used to translate those graph grammar rules into Kermeta programs, composing by operations, pre/post conditions and invariant. Each graph grammar rule will be translated into one Kermeta operation. Fig. 10 shows the structure of the GG compiler proposed in the project. The compilation process is described as following.

The compiler first traverses the left-hand-side of the graph grammar rule, to collect the objects, their relations, and specified attributes. Then compiler uses the knowledge from the 'Relation Database' to translate them into conditional expressions used in Kermeta matching code, first part of operation. The condition of each rule will be converted into pre and post condition of each operation, which must be satisfied before the operation being executed.

After matching the LHS graph, the compiler traverses the right-hand-side of the graph grammar rule, gathers information of the differences between the RHS and LHS, which include deleted objects, new created objects, updated attributes, and so on. Then the compiler translates them into assignment statements based on the 'Relation Database' into Kermeta updating code to update the model instance. The action script of the rule will be directly used as part of updating code.

The compiler also generates framework code, which includes user interactive code, outputting results, trace code, and extra helper methods.

After translating all the GG rules, the compiler merges them with extra framework code into one executable Kermeta program.

3.2 Relation Database

The graph grammar uses Abstract Syntax Graphs (ASG) representation. Every object in ASG has the same set of attributes, so the code used by the compiler for traversing LHS/RHS graphs can be used in the target programs. Thus, the compiler can perform one-to-one translation. However, Kermeta meta-model uses internal data structure defined by the class diagram. Each meta-model has different class diagram, thus, has different data structure and classes/attributes relations. Also, there is no one-to-one relationship between Kermeta data structure and

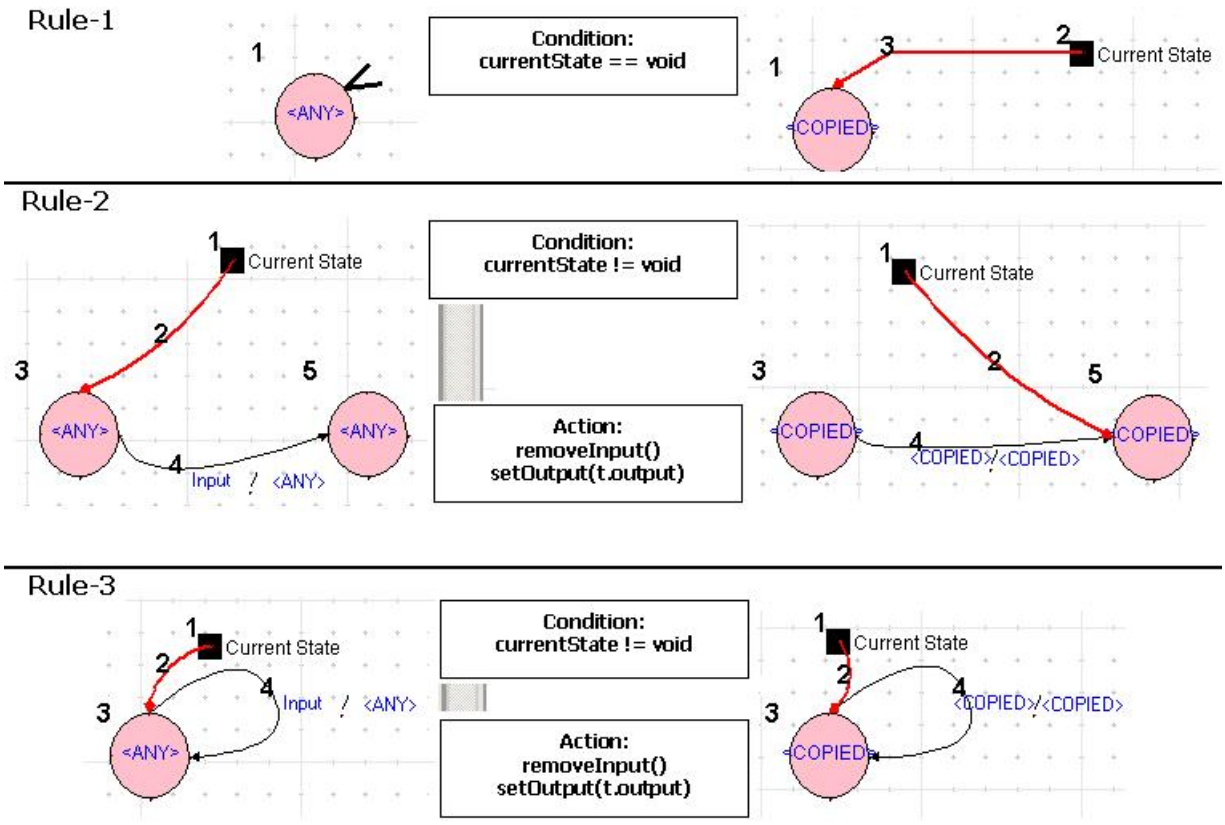

Fig. 9. FSA Model Graph Grammar Rules

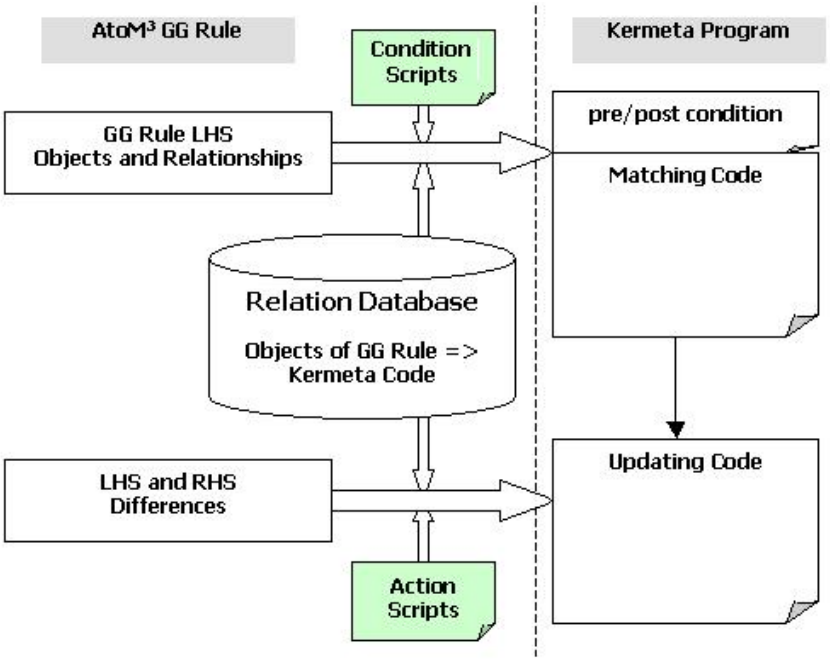

Fig. 10. FSA Model Graph Grammar Rules

ASG. The same property in the different ASG classes needs to be translated into different Kermeta code. Fig. 11 shows the differences of classes and attributes between AToM^3 and Kermeta. For example, in ASG the code represents the outgoing transition connected to a State is 'FSAState.out_connections_', but in Kermeta, it is 'State.outgoingTransition'. For the target State of a Transition in ASG that can be denoted as 'Transition.out_connections_', but in Kermeta it is 'Transition.target'.

Class in GG	Class in Kermeta FSM
FSAState	State
FSATransition	Transition
current	FSM.currentState
points_to	--

Properties	
FSAState.	***State.***
out_connections_	outgoingTransaction[0..*]
in_connections_	incomingTransaction[0..*]
FSATransition	***Transition***
out_connections_	target
in_connections_	source

Fig. 11. Naming Difference between AToM^3 and Kermeta

Therefore, we build a 'Relation Database' to store the knowledge from translating ASG graph objects into corresponding Kermeta object for FSA formalism. Fig. 12 shows the translation knowledge stored in the 'Relation Database'.

The program of graph grammar to Kermeta compiler is based on the GGCompiler.py of [6]. It uses the original program logic for interpreting LHS and RHS graphs, and adds the code for translating ASG graph fragment into corresponding Kermeta code Kermeta, and generate program output.

4 Constructing the Compiler

4.1 Changes on Conditions

Several changes are made on the conditions of graph grammar rules. In the Kermeta, the pre and post conditions can only use local variables of the operation it belongs to, and member variables it can access. In this project, the compiler translates graph grammar rules into operations that are belonged to FSM class, and the pre and post condition are belonged to those operations. Because the

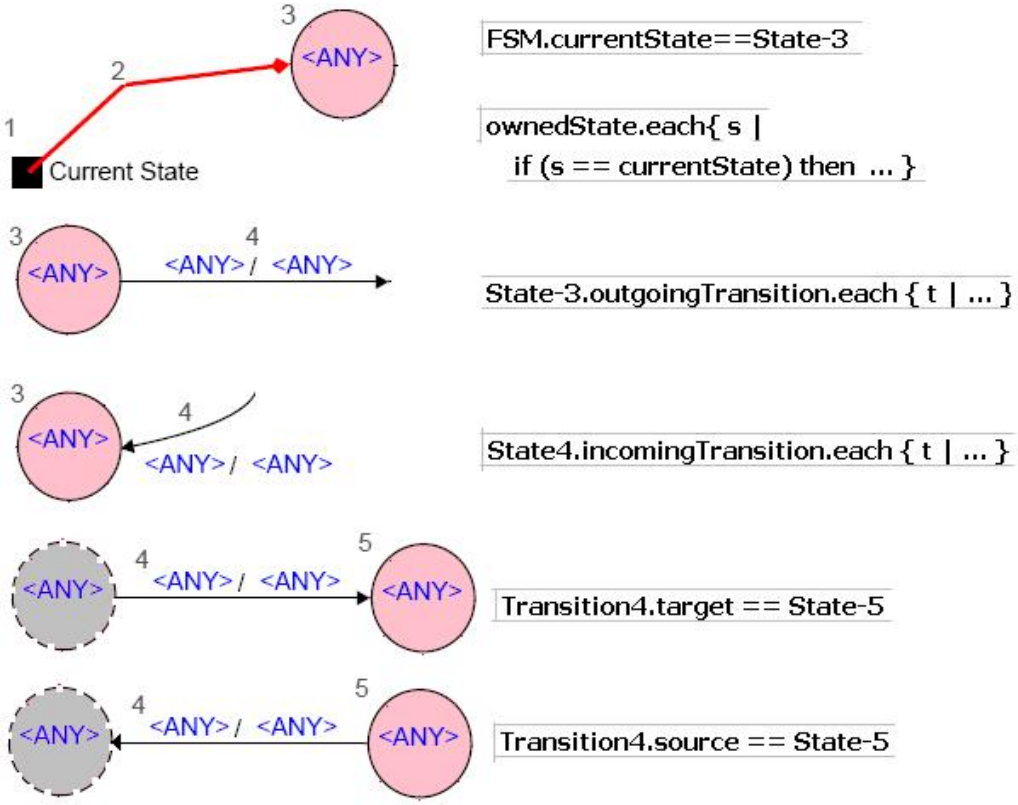

Fig. 12. FSA Model Graph Grammar Rules

local variables are generated by compiler whereas user cannot know; therefor, per and post conditions should only use attributes of the FSM class. So, we use new conditions, 'currentState==void', to substituted the original ones in FSA graph grammar (Fig. 9). In Rule-2 and Rule-3, the original ones are converted into specified value of input attribute of transition-4. Action codes of those rules are changed to calling the helper methods. We assume users know them.

4.2 Compilation Results

The new graph grammar rule-1 and translated Kermeta operation are shown in Fig. 13).

```
operation Rule01(str : String) : String raises FSMException
    pre preRule01 is
        currentState == void
is do
    ownedState.each
    { s |   // Matching LHS
        if(s == initialState) then
            do  // LHS->RHS changes
                currentState := s
                // Generel error handling
                rescue (err : Exception)
                    stdio.writeln(err.toString)
                    stdio.writeln(err.message)
            end
        end }
end
```

Fig. 13. Kermeta Code for Rule-1

The new graph grammar rule-2 and translated Kermeta operation are shown in Fig. 14).

The new graph grammar rule-3 and translated Kermeta operation are shown in Fig. 15).

4.3 Simulation Examples

After translating the graph grammar into Kermeta program, we have conducted two test cases. In order to show more valuable results, the priority order of three rules is Rule-1, Rule-3, and Rule-2; and the simulations are set to automatic trying all rules until no rule can be applied.

Fig. 16 shows simulation of the even binary number recognizer FSA model (Fig. 5). The trace shows that after user inputs '0', the Kermeta program successfully applies the Rule01 and Rule02, and move to state 'End_0', then report

```
operation Rule02(str : String) : String raises FSMException
    pre preRule02 is currentState != void
is do
    ownedState.each
    { s | // Matching Code
        if (s == currentState) then
            s.outgoingTransition.each { t |
                if (t.input.equals(Input)) then
                    do  // Updating Code
                        setOutput(t.output) // 1. Produce output
                        removeInput()       // 2. consumed the input
                        currentState := t.target // 3. updating current-state
                        // Generel error handling
                        rescue (err : Exception)
                            stdio.writeln(err.toString)
                            stdio.writeln(err.message)
end     end     }   end    }     end
```

Fig. 14. Kermeta Code for Rule-2

```
operation Rule03(str : String) : String raises FSMException
    pre preRule03 is currentState != void
is do
    ownedState.each
    { s | // Matching Code
        if (s == currentState) then
            s.outgoingTransition.each { t |
                if (t.input.equals(Input) and t.target==s) then
                    do  // Updating Code
                        setOutput(t.output) // 1. Produce output
                        removeInput()       // 2. consumed the input
                        // Generel error handling
                        rescue (err : Exception)
                            stdio.writeln(err.toString)
                            stdio.writeln(err.message)
end     end     }   end    }     end
```

Fig. 15. Kermeta Code for Rule-3

pre-condition violation when trying Rule-1 again. Then it stops at state 'End_0'. After user inputs another '0', Rule-3 is applied. And so on ...

```
Current state : <None>
>> give me a letter : 0
-- Rule01 is applied.
Current state : Init
-- Rule02 is applied.
string produced : e
Current state : End_0
[kermeta::exceptions::ConstraintViolatedPre:15497]
pre preRule01 of operation Rule01 of class FSM violated
Current state : End_0
>> give me a letter : 0
[kermeta::exceptions::ConstraintViolatedPre:15662]
pre preRule01 of operation Rule01 of class FSM violated
-- Rule03 is applied.
string produced : f
Current state : End_0
[kermeta::exceptions::ConstraintViolatedPre:15718]
pre preRule01 of operation Rule01 of class FSM violated
Current state : End_0
>> give me a letter : 1
[kermeta::exceptions::ConstraintViolatedPre:17439]
pre preRule01 of operation Rule01 of class FSM violated
-- Rule02 is applied.
string produced : d
Current state : End_1
[kermeta::exceptions::ConstraintViolatedPre:17500]
pre preRule01 of operation Rule01 of class FSM violated
Current state : End_1
```

Fig. 16. Simulation Of The Even Binary Number Recognizer FSA Model

Fig. 17 shows simulation of a FSA model, which contains non-transitions between S1, S2, and S3. The trace shows that after user inputs 'd', the program successfully applies the Rule01 and Rule02, and move to state 'S1'. Then the program automatically retries all the rules, and successfully applies Rule02 twice on non-transition and stop at state 'S3'.

5 Conclusions and Future Work

5.1 Conclusions

Coding Kermeta program for specifying operational semantic of meta-model should be replace by modeling approach such as AToM^3's graph grammar model.

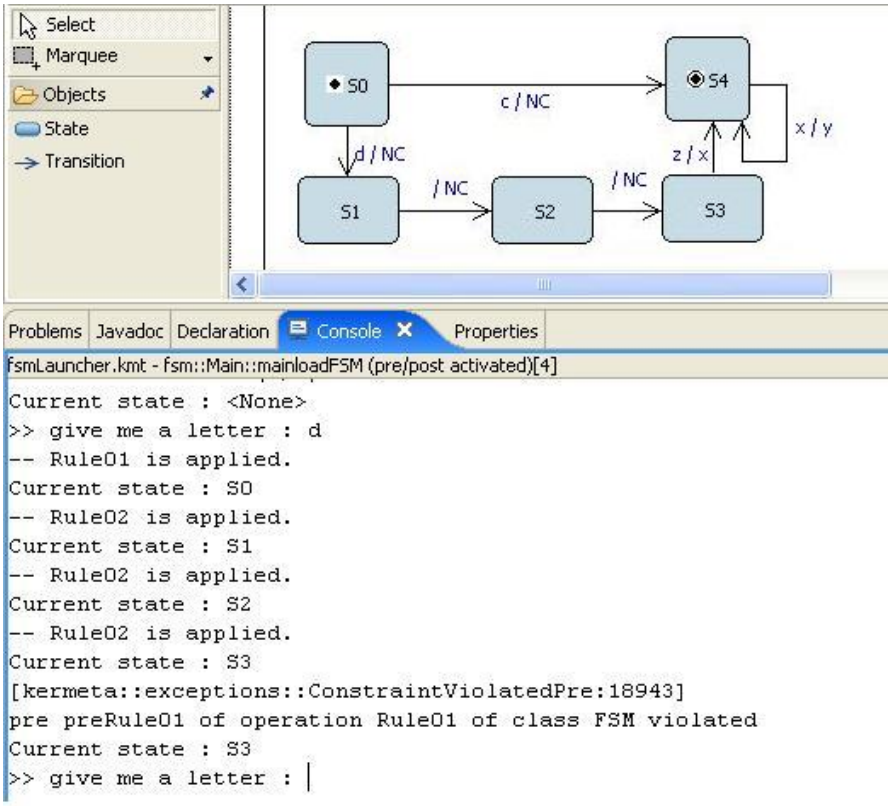

Fig. 17. Simulation Of FSA Model with Non-Transition

In this project, we demonstrate the process of translating the AToM^3's graph grammar model, which contains operational semantic of Kermate meta-model, into Kermeta program. However, the class diagram used to define Kermate meta-model contains too much implicit relations between classes and attributes. It makes the internal data structure of Kermeta meta-model really complicated and inconsistent between meta-models. So the graph grammar compiler needs the knowledge of those relations to be able to do the translation for other meta-models.

On the other hand, in AToM^3, graph grammar model and all other models are represented by the same data structure, Abstract Syntax Graphs (ASG). ASG has unified classes/attributes definition, thus it is easy for compiler to traverse the graph and find relationship between objects. Therefore, ASG is very efficient for graph rewriting and model simulation. The graph grammar compiler for AToM^3 can have one-to-one relationship when translate graph grammar into python classes for standalone simulation.

The Kermeta meta-model has its own advantage. It is very easy to generate Object-Orient programs from class diagrams. The internal data structure is efficient when doing non-graph calculations. Also the Kermeta programs are more readable, easy to debug in the comprehensive IDE, and meet software engineer aspect, which is a major objective for model translation.

5.2 Future Work

The implicit relation between classes and attributes are defines in the class diagram. The Kermeta compiler has used it to generate Kermeta class program. If the GG compiler can get those relations from the class diagram, then the compiler will not need the 'Relation Database', and be able to work for all the meta-models.

Acknowledgments. I would like to thank Professor Hans Vangheluwe, who proposed this topic, lead me to the right direction, and provided useful references.

References

1. F. Fleurey, Z. Drey, D. Vojtisek, C. Faucher.: Kermeta Language, Reference Manual. Internet: http://www.kermeta.org/docs/KerMeta-Manual.pdf, 2006.
2. Muller, PA., Fleurey, F., Jézéquel, JM., Weaving executability into object-oriented meta-languages, in Proc. of MoDELS'05, Montego Bay, Jamaica, October 2005, LNCS 3713, pp. 264–278
3. de Lara, J., Vangheluwe, H.: AToM^3: A tool for multi-formalism and meta-modelling. In European Joint Conference on Theory And Practice of Software (ETAPS), Fundamental Approaches to Software Engineering (FASE), Lecture Notes in Computer Science 2306, pages 174 - 188. Springer-Verlag (2002).
4. de Lara, J., Vangheluwe, H., Alfonseca, M.: Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. Software and Systems Modeling (SoSyM), 3(3):194-209, August 2004.

5. Vangheluwe, H., de Lara, J.: Meta-models are models too. In E. Yücesan, C.-H. Chen, J.L. Snowdon, and J.M. Charnes, editors, Winter Simulation Conference, pages 597 - 605. IEEE Computer Society Press, December 2002. San Diego, CA.
6. Eugene Syriani and Hans Vangheluwe. Programmed Graph Rewriting with DEVS. In Applications of Graph Transformations with Industrial Relevance (AGTIVE 2007), LNCS, Springer-Verlag, October 2007
7. Eugene Syriani and Hans Vangheluwe. Programmed Graph Rewriting with Time for Simulation-Based Design. In: International Conference on Model Transformation (ICMT 2008). LNCS, Springer-Verlag, 2008.
8. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R. and Grose, T. Eclipse Modeling Framework. Addison Wesley Professional, 2003.
9. OMG. MOF 2.0 Core Final Adopted Specification, Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2004.