Mutation-based Model Synthesis in Model Driven Engineering

Sagar Sen
McGill University (also at IRISA)
School of Computer Science
McConnell Engg. Bldg. 202
Montreal Canada

Benoit Baudry
IRISA
Campus de Beaulieu
35042 Rennes Cedex
France

Abstract

With the increasing use of models for software development and the emergence of model-driven engineering, it has become important to build accurate and precise models that present certain characteristics. Model transformation testing is a domain that requires generating a large number of models that satisfy coverage properties (cover the code of the transformation or the structure of the metamodel). However, manually building a set of models to test a transformation is a tedious task and having an automatic technique to generate models from a metamodel would be very helpful. We investigate the synthesis of models based on plans*. Each plan comprises of a sequence of model synthesis rules (or mutation operators) specified as* Graph Grammar *(GG) rules. These mutation operators are primitive* GG *rules , automatically obtained from any meta-model. Such plans can be evolved by various artificial intelligence techniques to generate useful models for different tasks including model transformation testing.*

1. Introduction

The ever growing complexity of software systems has driven research in academia and industry towards development of methodologies that enable automatic synthesis of software from high-level system descriptions, such as *models*. Modelling a software system is based on principles from an emerging field known as *Model-driven Engineering* (MDE) [9]. A *meta-model* in the MDE framework specifies a modelling language that is used to create models that conform to the modelling language. A modelling language is associated with its *abstract syntax, concrete syntax,* and *semantics* [7] and this set is called the *modelling formalism*. Any form of manipulation on a model is done via a *model transformation*. Model transformations have many uses such as to specify formalism transformations, to define operational/denotational semantics, to obtain the concrete syntax of the model, and for evolving the model for various purposes [4]. These applications of model transformations make it a very important part of the MDE framework since in practice several models undergo such transformations. Therefore given a model transformation we ask, is the model transformation error-free, is it efficient, and does it cover all the important representative model types?

The focus of this paper is on applying large scale testing for the validation of model transformations. To test a model transformation it is first necessary to give it input models that conform to a modelling language. It is tedious to construct these models by hand and hence there is a need for automatic model synthesis. Typically, in model transformation testing research the effectiveness of a test model generation technique is measured by its ability to detect faults injected into the tested model transformation. This artificial injection of faults in the model transformation to test the validity of a test model is known as a *mutation analysis*. Mutation analysis in model transformation testing [8] is employed to validate effective test model synthesis methods. However, synthesizing models for testing requires the mutation of models itself. Models are mutated (modified) with the hope that they cover different parts of the model transformation or the meta-model. Our work is concerned with specifying *mutation operators* to synthesize input models.

Mutation operators for models have been proposed for specific domains such as UML class diagrams [13], temporal logic formulae, labelled transition systems [3], and component models [6]. However, the domain specificity of the mutation operators in the existing approaches makes it difficult for us to easily extend their use to other domains. We extend these mutation operators to

make them powerful enough to make both small changes to a model and also to completely generate any model conforming to its meta-model specification. We propose mutation operators that are automatically synthesized from any meta-model in a domain of knowledge. A sequence of these primitive mutation operators can be used to both synthesize a complete model and to enhance an exisitng model.

We describe how a meta-model can be specified as an instance of the Essential Meta-object Facility (EMOF) [2] meta-meta-model. EMOF is an Object Management Group (OMG) [1] standard for specifying meta-models to develop domain specific modelling languages. The meta-models specified using the EMOF standard are further constrained by constraints specified using the Object Constraint Language (OCL) [10]. A *valid model* conforms to a modelling language by conforming to its EMOF meta-model and by satisfying all the associated OCL constraints. The abstact syntax of a model is specified using an *abstract syntax graph*. We use the *Himesis* graph kernel [11] to represent the abstract syntax of a model using *hierarchical labelled graphs*. A set of primitive mutation operators from the meta-model are synthesized as Graph Grammar (GG) rules [12]. These GG rules are applied in a sequence to synthesize a model in its abstract syntax graph representation. The sub-graph matching algorithm in the *Himesis* kernel is used for pattern matching in the GG rules for model mutation and synthesis. Once, the mutation operators are generated we introduce the notion of a *plan* in our framework, to combine synthesized mutation operators. A plan is a sequence of lists. Each list consists of a mutation operator and its parameters. This sequence or plan is used to synthesize a complete model just from mutation operators. The ideas in this paper are illustrated using a running example of the Hierarchical Finite State Machine (HFSM) formalism.

The structure of the paper is as follows. In Section 2 we introduce meta-modelling with constraints to specify models of a modelling language. The process of generating mutation operators from a meta-model specifiation is given in Section 3. In Section 4 we describe how mutation operators can be combined to form a plan which, when executed, results in a model. We conclude in Section 5.

2. Meta-modelling with Constraints

2.1 Meta-models

sec:metamodels A *meta-model* specifies the domain of a set of conforming *instances* or *models* in a modelling language. Taking this statement a step further we can say that a *meta-meta-model* specifies the domain of a set of conforming meta-models. In the MDE framework EMOF is a meta-meta-model. EMOF can model itself. This property of EMOF is known as bootstrapping, making EMOF expressive enough to be a starting point for modelling itself and meta-models for other modelling languages. For instance, in Figure 1 we show the meta-model for the Hierarchical Finite State Machine (HFSM) modelling language in EMOF.

The meta-model of HFSM as a model of EMOF is specified using the notion of *classes* and their *properties*. A class represents an entity in the modelling domain. For instance, in the HFSM modelling language the State class represents the set of all possible State objects. Each class contains zero or more properties. A property is an *attribute* or a *reference*. An attribute of a class is a constant or a variable of a primitive datatype. The primitive datatypes are Float, Integer, Boolean and String. The domain of the values taken by these primitive datatypes is divided into default *partitions* as shown in Table 1.

A reference associates one class to another as a relationship. A reference has a name and is associated with a *multiplicity*. The multiplicity constrains the number of relationships of a particular type between two associated classes. For instance, an AbstractState class has a reference incoming-Transition with multiplicty 0..*, implying that an AbstractState can have zero or more incoming Transition objects. The multiplicities for the relationship with references incomingTransition and source between the classes AbstractState and Transition respecitively is also shown in Figure 1. Finally, *inheritance* in the meta-model allows reuse and automatic copying of common properties in a super class to its sub classes. For instance the properties of the class AbstractState are inherited by classes State and Composite.

2.2. Constraints on Meta-models

Classes and their properties constrain the way models are synthesized. However, some constraints that cannot be expressed directly within the meta-model have to be specified along with the meta-model. For example, it is not possible to use multiplicities to express that a HFSM must contain only one initial state. This has to be defined in an external constraint that is attached to the meta-model. The OCL can be used to define such properties. The OCL expression that constrains the number of initial states is :

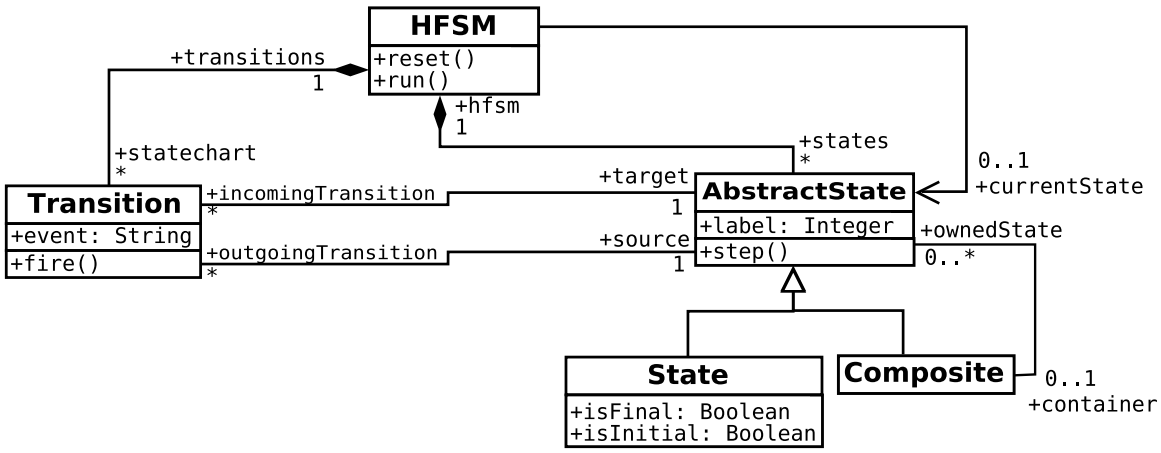


Figure 1. The Hierarchical Finite State Machine Meta-model

Table 1. Default Partitions for Primitive Datatypes

Type	Structural Partition
Boolean	$\{true\}$, $\{false\}$
Natural	$\{0\}$,$\{1\}$,$\{2, ..MaxInt\}$
Integer	$\{MinInt, .., -2\}$,$\{-1\}$,$\{0\}$,$\{2, ..MaxInt\}$
String	$\{null\}$,$\{""\}$,$\{". + "\}$
Enumeration	each literal

$context\ State\ inv$:
$State.allInstances() \rightarrow select(s|s.isInitial = 1) \rightarrow size() = 1$

2.3. Model Representation

The domain of values of each attribute and the multiplicities of the references in a meta-model defines a set of valid models that conform to the meta-model. This set can also be called the *model design space*. A point in the model design space is a model. A sequence of mutations on a model can be used to explore the model design space. If a sequence of mutation operators (see Section 3) results in a model that does not conform to the meta-model and its constraints then the model is outside the model design space.

In the MDE community it is common practice to visually represent a model as an *object diagram*. The object diagram for the HFSM model in Figure 2 is shown in Figure 3. The object diagram has the structure of a *labelled graph* with *nodes* and *edges*. We transform objects in memory to a graph in a graph modelling formalism. Our chosen graph formalism is that of *hierarchical labelled graphs* as designed and implemented in the Himesis graph kernel [11]. There are two node types in Himesis. First, a *node* is just a labelled node with two properties which are *label* (which usually represents the type of a node), and a *Name* (which is a unique identifier). Second, a *primitive node* stores a primitive variable such as a String, Float, Boolean, or an Integer. The variable *value* is an additional property of primitive nodes.

We represent an object of a class in the model graph by first creating a node to represent the object. The node is labelled with the name of the concrete class of which the object is an instance. The name property of the node takes the name of the object. The primitive attributes of an object are transformed to primitive nodes that connected with the node representing the object via a *parent-child* edge. A parent-child edge is represented using a directed dotted blue arrow. A reference node of an object is also connected using a parent-child edge. However, the reference node is not a primitive node but just a node since it could be used to connect to reference nodes in other related objects. This is determined by the relationships obtained from the meta-model.

The attributes and the nodes for the example

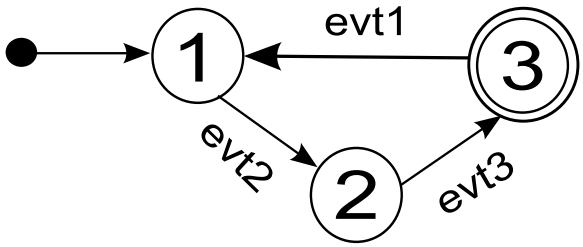


Figure 2. A Hierarchical Finite State Machine model in Concrete Syntax

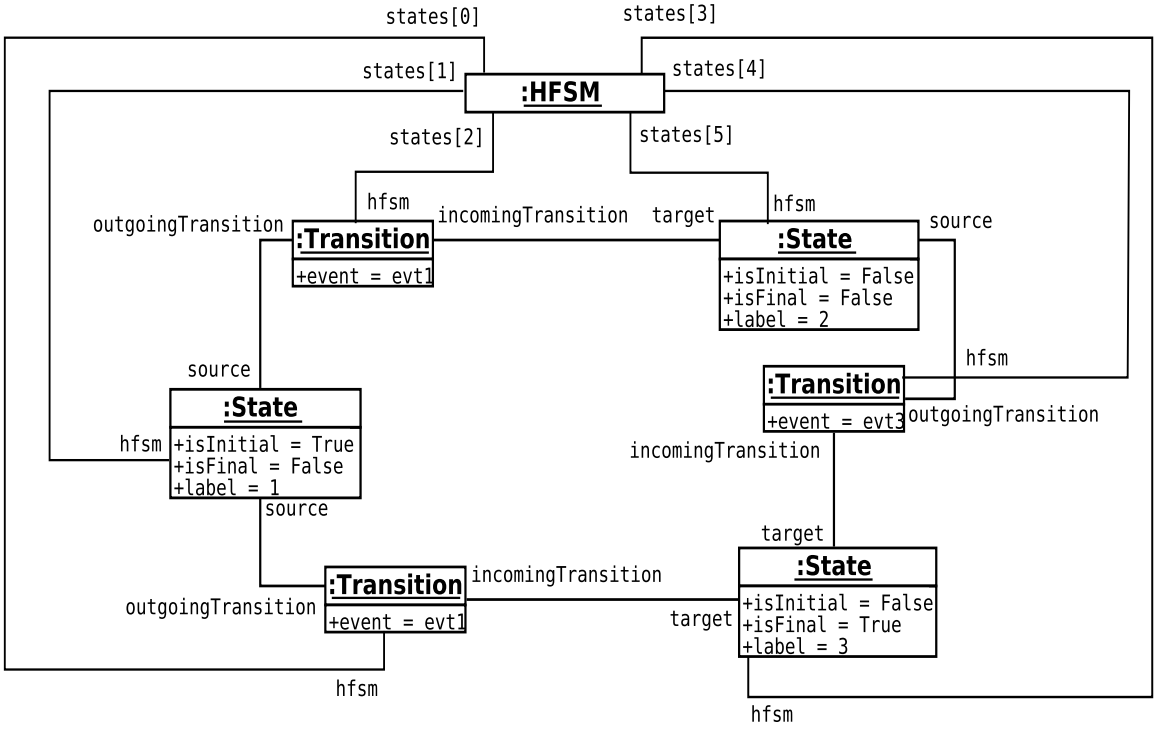


Figure 3. The Object Diagram of a Hierarchical Finite State Machine Model

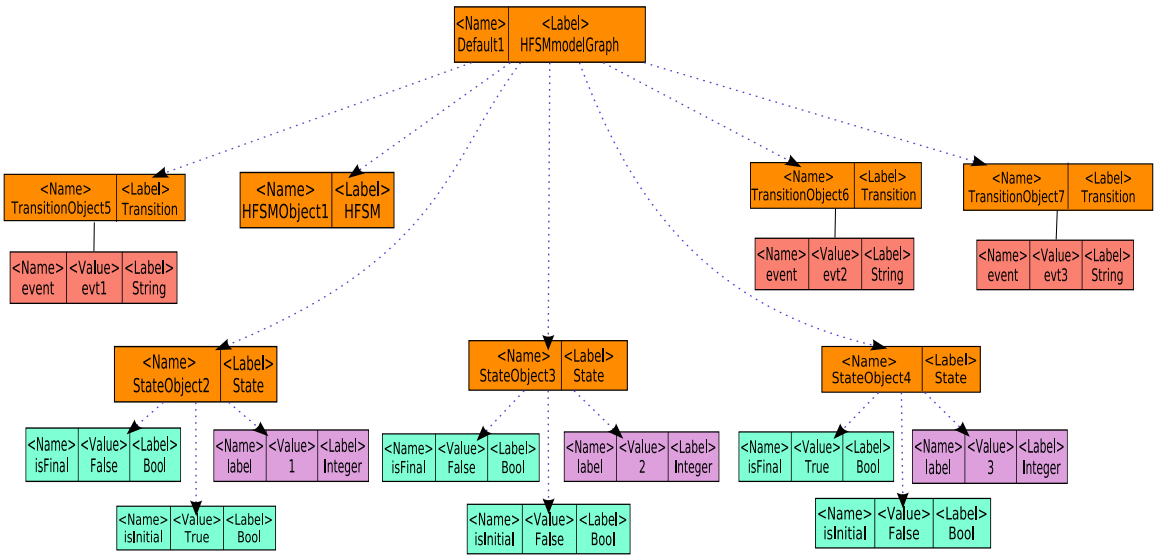


Figure 4. Attributes View of Abstract Syntax Graph Representation of an HFSM Model

shown in Figure 2 is illustrated in the attributes view of the graph representation in Figure 4. The attributes for each object node in the graph are shown in this view. For instance, the event attribute of TransitionObject5 is a primitive node of type String. This ofcourse is not the complete graph representation of our model. We now see how references are represented in the graph formalism.

To represent a relationship between associated objects we connect references which are child nodes of the related objects. We show the references view of the example model (in Figure 2) in Figure 5. A *connection* edge is created between related reference nodes. This is represented using an undirected black line between reference nodes. For instance, the reference outgoingTransition which is a child node of StateObject2 is connected to source which is a child node of TransitionObject5. The container reference for State object is not shown (although it exists) since our example does not contain a Composite object. Therefore, the attributes view in Figure 4 and the references view in Figure 5 together represent the graph of the example model in Figure 2. They have been shown separetely for ease of explanation and due to space constraints.

We choose a graph representation for our models to make them viable for the application of Graph Grammar (GG) rules for model synthesis and mutation. In the next section we see how given a meta-model a model transformation is able to synthesize a set mutation operators in the form of GG rules.

3. Mutation Operators for Model Synthesis

We deal with the model synthesis problem by exploring the space of abstract syntax graphs representing the models. To explore the space of graphs we introduce mutation operators based on Graph Grammar (GG) rules [5]. Graph Grammars are a generalization, for graphs, of Chomsky grammars. Graph Grammar are composed of a ordered collection of rules. Each rule consists of Left Hand Side (LHS) and Right Hand Side (RHS) graphs. Rules are evaluated against an input graph, called the host graph. If a matching is found between the LHS of a rule and a sub-graph of the host graph, then the rule is applied. When we apply a rule, the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions (pre-conditions and post-conditions), as well as actions (pre-actions, post-actions, specifications) to be performed when the rule is applied. We use the Himesis sub-graph matching routine and the Python programming language to specify GG rules. The sub-graph matching algorithm in Himesis returns a list of matches called Matches. Each match in Matches is a list of tuples containing two elements. The first element is the node in the sub-graph and the second one is the matching node in the host graph. The action performed by the rule utilizes the reference to the matching host graph node for modifying the

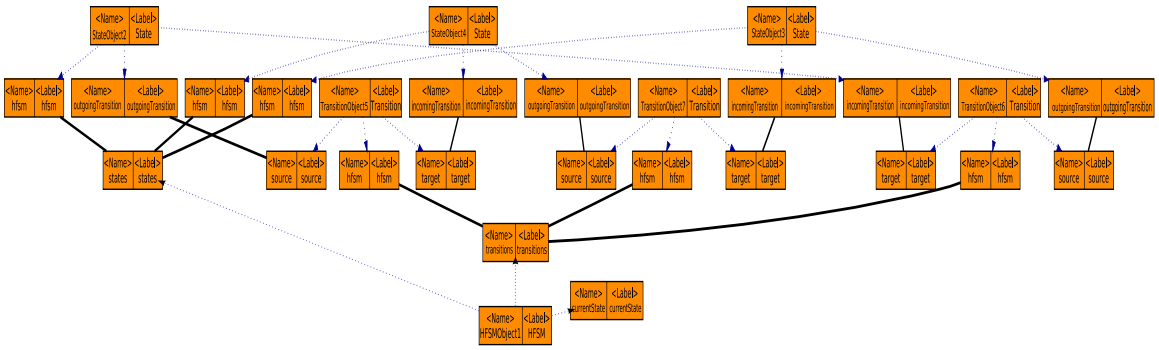


Figure 5. References View of Abstract Syntax Graph Representation of an HFSM Model

graph. Graph Grammars can transform models between formalisms (specifying denotational semantics), structurally optimize models, and also can be used to specify operational semantics such as simulations. We use it to develop the structure and properties of a model.

The mutation operators are automatically obtained from any meta-model specification using the model transformation MM_2_GG_RuleGenerator. MM_2_GG_RuleGenerator takes a meta-model as input and generates domain-specific GG rules. MM_2_GG_RuleGenerator is based on EMOF and synthesizes rules using the concepts of classes, their properties, and inheritance.

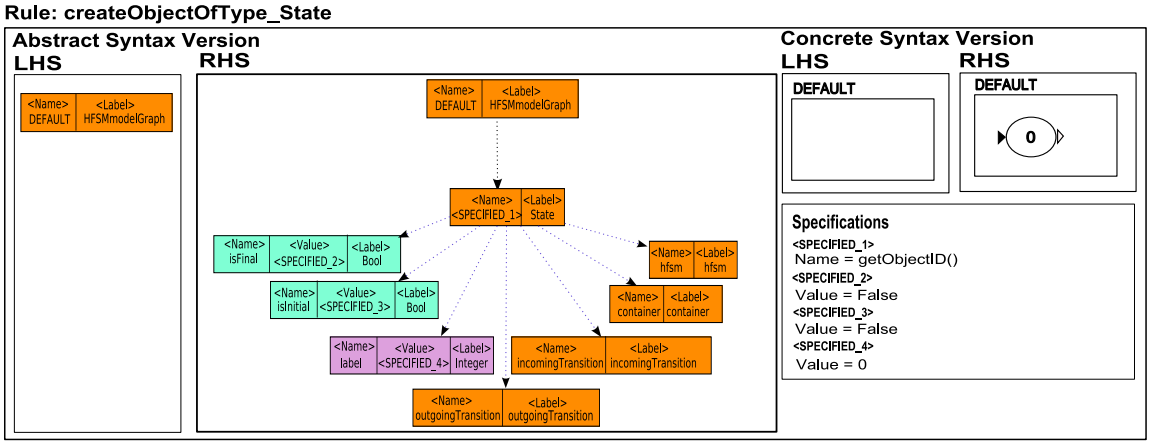
We have identified three kinds of mutation operators. The operators/rules can be used for:

1. Creation of an object of a particular concrete class in the model graph
2. Creation of a relationship between two existing objects
3. Specification of an attribute for an existing object

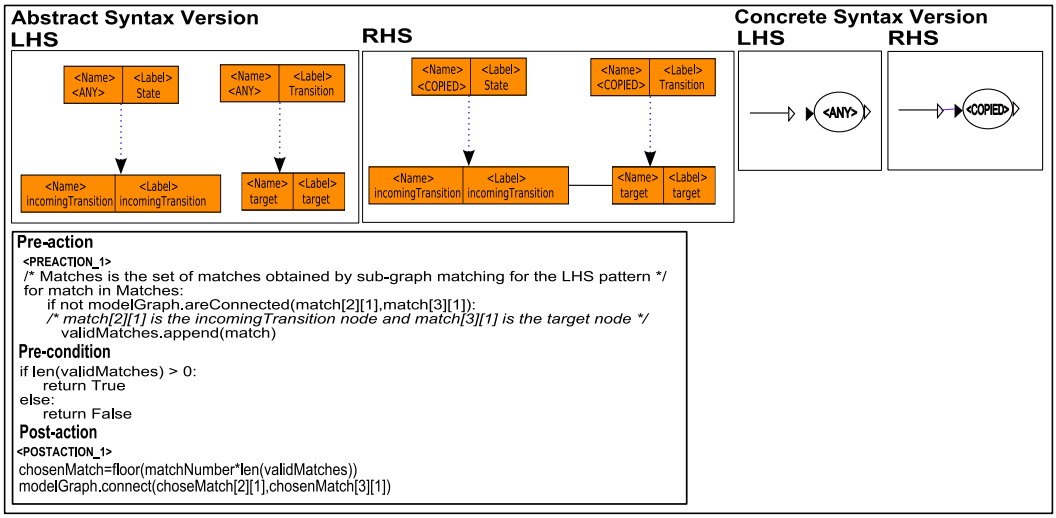
MM_2_GG_RuleGenerator synthesizes code for GG rules to implement the operators presented. All rules that are used to create objects of a particular concrete class are named using the template createObjectOfType_*concreteClass*. The placeholder *concreteClass* takes the name of a class in the meta-model. There are no paramters for object creation rules. The objects are created with default values. For instance, we create an object of type State by the application of the rule in Figure 6 (a). The LHS finds the HFSMmodelGraph top-level container node. The RHS of the rule then adds a node with label State and its child nodes (which are the class properties) with default values. The *specifications* on attribute values are also shown. For instance, the Name attribute gets a unique objectID using the getObjectID routine. The concrete syntax version of the operation is also shown in Figure 6 (a). The rectangle represents the HFSMmodelGraph top-level container node and the State node is represented by a circle. The incomingTransition and outgoingTransition references shown as triangular ports.

The rules used to create a relationship between related references are named using the template createRelationshipOfType_*relationshipName*. The placeholder *relationshipName* contains the name given to a relationship in the meta-model. First, we execute sub-graph matching to identify the patterns in the graph where a relationship is applicable. The rules for creating a relationship require a parameter called the *matchNumber*. The matchNumber parameter selects a match from the list of valid matches in the model. The value of matchNumber is a real number between 0 and 1. The list validMatches is first created from the list of Matches. This list contains the nodes where we can apply a rule. For instance, in Figure 6 (b) we present the rule where the incomingTransition reference of a State object is connected to the target reference of a Transition object. The list of valid matches is obtained from the list of Matches for the LHS in the rule. A match is valid if the reference nodes are not connected to each other. From the set of all valid matches a valid match is chosen using the function: $chosenMatch = floor(matchNumber * len(validMatches))$. The rule is applied on the model graph (or host graph) using the reference to the nodes in the chosen match.

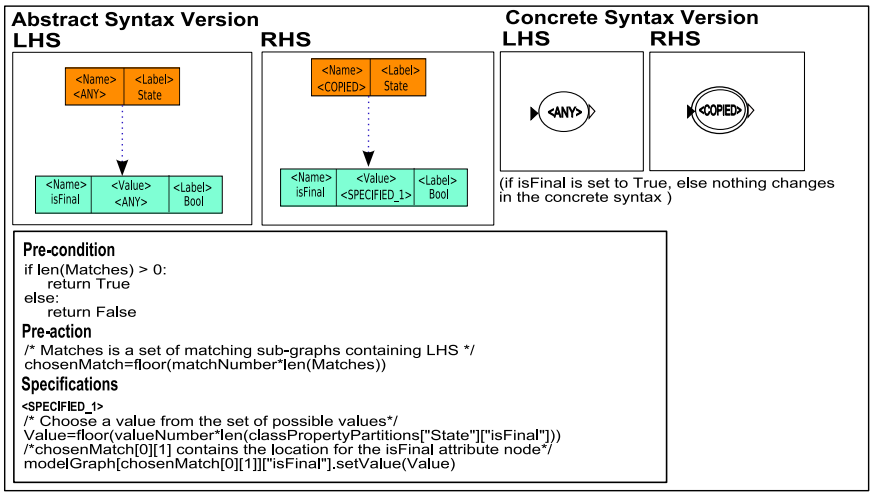
The third kind of rules are used to specify the value of a primitive attribute in



(a)



(b)



(c)

Figure 6. Three Types of Mutation Operators

the model. These rules are synthesized by MM_2_GG_RuleGenerator using the template specify*concreteClass*Attribute_*attributeName*. The *concreteClass* placeholder contains the name of the concrete class whose attribute *attributeName* the rule specifies. For instance, in Figure 6 (c) we show the generated rule specifyStateAttribute_isFinal for setting the value of the isFinal attribute of a State object. These rules take two parameters. One is the *matchNumber*. It has the same meaning as before, i.e. it chooses a match from the set of Matches where the rule is applicable. The second parameter is *valueNumber* which is a real number between 0 and 1. The valueNumber attribute is used to choose an element from the domain of possible values for the attribute. The list of possible values that can be taken for an attribute is given in Table 2. The table also contains the list of multiplicities for references (preceded by a #) but we do not use the information in this paper as it does not deal with checking of structural constraints. The value is chosen in a fashion similar to choosing a match in a list of Matches except for it is chosen from the domain of values (see Table 2).

The mutation operators generated from the HFSM meta-model using MM_2_GG_RuleGenerator are shown in Table 3. These rules are genereted based on the templates described in this section. In the next section we describe how these mutation operators are combined to given rise to a *plan* which when executed results in a model or a mutation of an existing model.

4. Model Synthesis Plans

We define a *plan* as a sequence of parameterized mutation operators. Applying a plan on a model graph is used to either completely synthesize a model or mutate an exisiting one. A plan is comprised of *atomic* operations. Each atomic operation consists of a mutation operator and two parameters. The first parameter is *matchNumber* and the second parameter is *valueNumber*.

The three types of mutation operators as discussed in Section 3 use zero, one or two parameters depending on the nature of their operation. An operator of type *createObjectofType*_concreteClass does not require a parameter as it just creates an object of type *concreteClass* with default property values and adds it to the model graph. The operator of type *createRelationshipOfType*_relationshipName requries one parameter which is the *matchNumber*. The operator of type *specify*concreteClass*Attribute*_attributeName requries two parameter which is the *matchNumber* and the *valueNumber*.

A plan in a list of 3 tuples (flattened in the list) and has a length which is a mutiple of 3. Every tuple of a plan first contains the opcode for the mutation operator, a float between 0 and 1 for the mathcNumber and a float between 0 and 1 for valueNumber. A sequence of these tuples comprise the complete plan which operates on a graph. For instance, the opcodes for the mutation operators for the HFSM formalism is given in Table 3. The domain of values for attributes is given in Table 2. The execution of a plan to create our example model is given in Figure 7. The executed plan is given at the bottom of the figure. The rules executed (not all in the same order as in the plan) are annotated over the arrows in the figure.

The mutation operators encoded in the plan do not always need paramters as discussed earlier, hence the since of the plan list can be be further reduced. The reason we have a uniform distribution of mutation operators and two parameters is for the potential application of genetic algorithms (which require a structure genome) or for hardware implementation. Many artificial intelligence planning technqiues evovle plans incrementally. In such cases the extraneuous parameters can be omitted.

5. Conclusion

Meta-modelling with constraints is a general approach to specify the domain of a modelling language. In particular, meta-modelling is very useful when there is a need to develop visual modelling languages. Developing testing methods for modelling languages is very essential to guarantee the quality of a modelling language and formalism. We have shown that a set of primitive mutation operators can be automatically generated from a model to mutate and to completely evolve a model. We have shown that a plan is a combination of these mutation operators and its structure is a lot like a genome making it an ideal candidate for application of genetic algorithms. We however, have not delved into the application of a genetic algorithm to evolve a plan in this article. A plan can also be incrementally developed using *artificial intelligence* (AI) planning methods such as reinforcement learning. The application of AI combined with symbolic constraint analysis to evolve plans that satisfy constraints and perform effective model transformation tests is planned future work.

Table 2. Partitions for the Hierarchical Finite State Machine Meta-model

Type	Partitions
Transition::**event**	$\{$"", $\{$"$evt1$"$\}$,$\{$". + "$\}$
Transition::**#source**	$\{0\}$,$\{1\}$,$\{2, ..MaxInt\}$
Transition::**#target**	$\{1\}$
AbstractState::**label**	$\{0\}$,$\{1\}$,$\{2, ..MaxInt\}$
AbstractState::**#container**	$\{0\}$,$\{1\}$
AbstractState::**#incomingTransition**	$\{0\}$,$\{1\}$,$\{2, ..MaxInt\}$
AbstractState::**#outgoingTransition**	$\{0\}$,$\{1\}$,$\{2, ..MaxInt\}$
State::**isInitial**	$\{true\}$,$\{false\}$
State::**isFinal**	$\{true\}$,$\{false\}$
Composite::**#ownedState**	$\{0\}$,$\{1\}$,$\{2, ..MaxInt\}$

Table 3. Synthesized mutation operators from the HFSM meta-model

Opcode	Mutation Operator	Description
	Object Creation Rules	**Parameters** : None
0	createObjectOfType_State	State object creation rule
1	createObjectOfType_Composite	Composite object creation rule
2	createObjectOfType_Transition	Transition object creation rule
3	createObjectOfType_HFSM	HFSM object creation rule
	Relationship Creation Rules	**Parameters** : (matchNumber)
4	createRelationshipOfType_relationship3	target::Transition→incomingTransition::State
5	createRelationshipOfType_relationship2	hfsm::Composite→states::HFSM
6	createRelationshipOfType_relationship1	states::HFSM→hfsm::State
7	createRelationshipOfType_relationship7	transitions::HFSM→hfsm::Transition
8	createRelationshipOfType_relationship6	outgoingTransition::Composite→source::Transition
9	createRelationshipOfType_relationship5	target::Transition→incomingTransition::Composite
10	createRelationshipOfType_relationship4	outgoingTransition::State→source::Transition
11	createRelationshipOfType_relationship9	container::Composite→ownedState::Composite
12	createRelationshipOfType_relationship8	container::Composite→ownedState::State
13	createRelationshipOfType_relationship11	hfsm::HFSM→currentState::Composite
14	createRelationshipOfType_relationship10	hfsm::HFSM→currentState::State
	Attribute Specification Rules	**Parameters** : (matchNumber, valueNumber)
15	specifyCompositeAttribute_label	Specifies *label* attribute for a Composite object
16	specifyStateAttribute_isFinal	Specifies *isFinal* attribute for a State object
17	specifyStateAttribute_isInitial	Specifies *isInitial* attribute for a State object
18	specifyStateAttribute_label	Specifies *label* attribute for a State object
19	specifyTransitionAttribute_event	Specifies *event* attribute for a Transition object

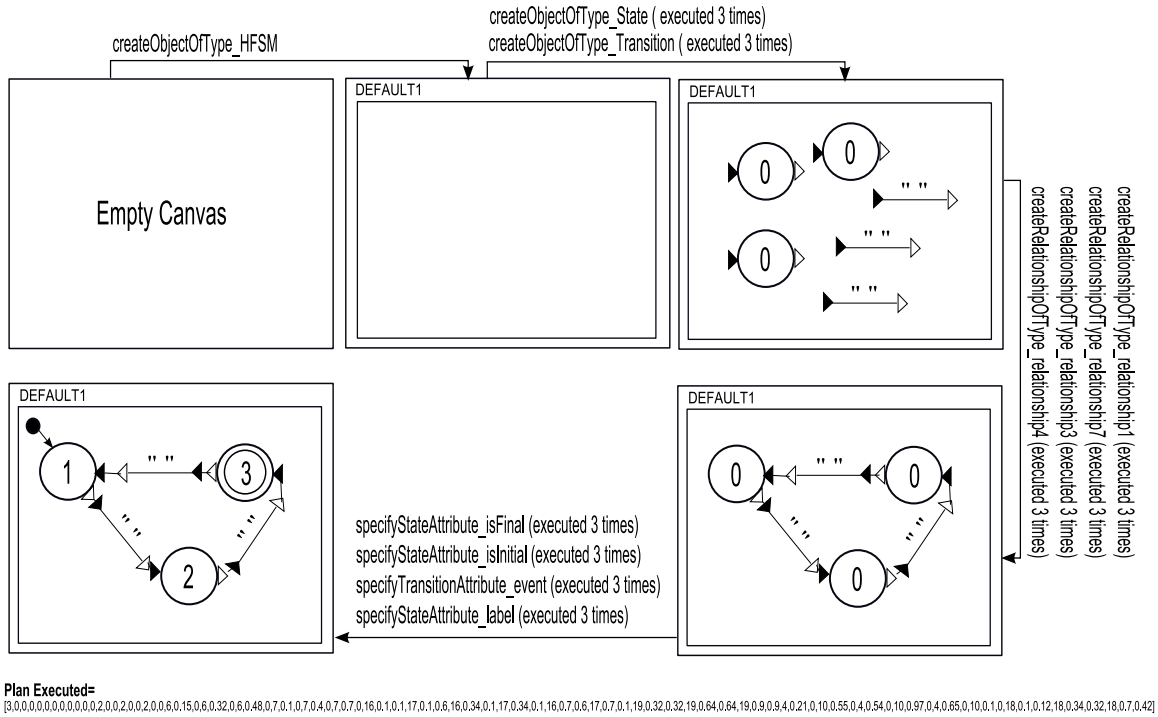


Figure 7. Execution of a Plan to Synthesize a Model

References

[1] OMG Home page. http://www.omg.org.

[2] OMG. MOF 2.0 Core Final Adopted Specification. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2005.

[3] Black P.E., Okun V. , and Yesha Y. . Mutation operators for specifications. In *Proceedings of ASE'00 (Automated Software Engineering)*, Grenoble, France, September 2000.

[4] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[5] Ehrig, H. and Engels, G. and Kreowski, H.-J. and Rozenberg, G., editor. *Handbook of Graph Grammars and Computing by Graph Transformation.*, volume Vol. 1-3. World Scientific, 1999.

[6] S. Ghosh and A. Mathur. Interface mutation. *Software Testing, Verification and Reliability*, 11(4):227–247, 2001.

[7] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel, 2000.

[8] J.-M. Mottu, B. Baudry, and Y. Le Traon. Mutation analysis testing for model transformations. In *Proceedings of ECMDA'06 (European Conference on Model Driven Architecture)*, Bilbao, Spain, July 2006.

[9] S. Kent. Model driven engineering. *LNCS , Integrated Formal Methods: Third International Conference,Turku, Finland,*, 2335:286, May 2002.

[10] OMG. The Object Constraint Language Specification 2.0, OMG Document: ad/03- 01-07.

[11] M. Provost. Himesis: A hierarchical subgraph matching kernel for model driven development. Master's thesis, McGill University, 2005.

[12] R. Bardohl, G. Taentzer, M. Minas, A. Schurr. *Handbook of Graph Grammars and Computing by Graph transformation, volume II: Applications, Languages and Tools*. World Scientific, 1999.

[13] T. Dinh-Trong, S. Ghosh, Robert France, Benoit Baudry, and Frank Fleurey. A taxonomy of faults for uml designs., october 2005. In *Proceedings of MoDeVa'05 (Model Design and Validation Workshop associated to MoDELS'05)*, Montego Bay, Jamaica, 2005.