1

Towards Domain-specific Model Editors with Automatic Model Completion

Sagar Sen , Benoit Baudry, Hans Vangheluwe

Abstract

Integrated development environments such as Eclipse allow users to write programs quickly by presenting a set of recommendations for code completion. Similarly, word processing tools such as Microsoft Word present corrections for grammatical errors in sentences. Both of these existing structure editors use a set of constraints expressed in the form of a natural language grammar to restrict/correct the user (*syntax-directed editing*) or formal grammar (*language-directed editing*) to aid document completion. Taking this idea further, in this paper we present an integrated software system capable of generating recommendations for model completion of partial models built in editors for *domain-specific modelling languages*. We present a methodology to synthesize model editors equipped with automatic completion from a modelling language's declarative specification consisting of a meta-model with a visual syntax. This *meta-model directed* completion feature is powered by a first-order relational logic engine implemented in ALLOY. We incorporate automatic completion in the generative tool AToM^3. We use the Finite State Machines modelling language as a concise running example. Our approach leverages a *correct by construction* philosophy that renders subsequent simulation of models considerably less error-prone.

Index Terms

Meta-model directed editing, Domain-specific modelling language, Structure editor, Partial model, Model completion, AToM^3, Alloy

I. INTRODUCTION

Documents in the form of computer programs, diagrams, chemical formulas, and markup text can currently be edited in document editors called *structure editors*. These structure editors are cognizant of the document's underlying structure such as the grammatical syntax or a formal grammar of the language. Functionally, these structure editors are syntax or language-directed to aid the user by presenting recommendations for completion of code, text, or a diagram based on correct possibilities prescribed by the underlying structure. This enables faster document development with fewer errors. However, structure editors are separately constructed for each domain-specific language and are built mainly for grammar-based textual languages. We are interested in the subject of extending structure editors from high-level models built using the principles of *Model Driven Engineering* (MDE) [13] where domain-specific model editors are automatically synthesized for a variety of modelling languages.

In MDE, given a meta-model specification of a domain-specific modelling language, software tools can automatically generate *domain-specific model editors*. For example, generative modelling tools such as AToM^3 (A Tool for Multi-formalism Meta-modelling) [11] [17],GME (Generic Modelling Environment) [3], GMF (Eclipse Graphical Modelling Framework) [21] can synthesize a domain-specific visual model editor from a declarative specification of a domain-specific modelling language. A declarative specification consists of a meta-model and a visual/textual syntax that describes how language elements (objects and relationships) manifest in the model editor. The designer of a model uses this model editor to construct a model on a drawing canvas. This is analogous to using an integrated development environment (IDE) to enter a program or a word processor to enter sentences. However, IDEs such as Eclipse present recommendations for completing a program statement when possible based on its grammar and existing libraries [5]. Similarly, Microsoft Word presents grammatical correction recommendations if a sentence does not conform to a natural language grammar. Therefore, we ask: Can we extrapolate similar technology or develop new technology for partial models constructed in a model editor for a domain-specific modelling language (DSML)?

Extrapolating code completion techniques for model completion is not feasible in the general case. The first reason is the difference between the underlying structure of code and models. Code completion techniques use the Backus-Naur Form (BNF) grammar of a programming language while models are specified by a meta-model and constraints on it. Second, model completion must consider completing the entire model as constraints can span entire models unlike code completion which presents solutions at a program statement level. Third, in terms of reduction in effort model completion must help reduce the effort of a modeller by automatically satisfying all relevant language constraints since in general they may be too hard for a modeller to resolve manually. The output of model completion must be one or many valid models that conform to their language. This notion of reduction in effort is different from that in code completion. Code completion presents local

Received 4 May, 2008, Revised 18 Sept, 2008, 22 Mar 2009, Accepted 19 May, 2009

Sagar Sen is the *corresponding author*. Email: ssen@irisa.fr, Postal Address: INRIA Rennes Bretagne-Atlantique, Campus universitaire de beaulieu, Rennes 35042, France, tel : + 33 2 99 84 72 98 fax : + 33 2 99 84 71 71

suggestions to complete navigational expressions or concept names but it does not perform constraint satisfaction to output a valid program. In the general case, model completion may take more time than code or sentence completion which are almost instantaneous. Therefore, there is a need to develop new techniques for model completion with different goals such as relaxing the exigence towards time to complete.

The major difficulty for providing completion capabilities in model editors is to integrate heterogeneous sources of knowledge in the computation of the possible solutions for completion. The completion algorithm must take into account the concepts defined in the meta-model, constraints on the concepts and the partial model built by a domain expert/user. The difficulty is that these three sources of knowledge are obviously related (they refer to the same concepts) but are expressed in different languages, sometimes in different files, and in most cases by different people and at different moments in the development cycle as they are separable concerns.

In this paper, we propose an automatic transformation from heterogenous sources of knowledge to an ALLOY [10] [20] constraint model. ALLOY is a declarative constraint language based on *quantified first-order logic* with support for operators in *relational calculus*. The generated ALLOY model is then used to synthesize a set of Boolean CNF formulae which are solved using an appropriate satisfiability (SAT) solver. The solutions are parsed and transformed from low-level logic and returned as high-level models to the model editor as recommendations for completing the partial model. These solutions are shown in the concrete visual syntax of the modelling language. Our transformation from the different sources to ALLOY is integrated in the software tool AToM^3. We select AToM^3 since its has a light-weight python implementation for rapid-prototyping and proof of concept. Also, AToM^3 supports an easy to use icon editor and DSML generator for rapidly synthesizing DSML visual modelling environments from meta-model and visual syntax specifications. A transformation module in AToM^3 can also also serialize standard XMI versions of models for use in industry standard tools such as Eclipse GMF. The simple and easy to use Python-API of AToM^3 allows fast integration of new features and transformations in modelling languages. A similar implementation in widely used tools such as Eclipse GMF is certainly feasible but will need more programming effort as it is Java-based where one needs experience in using the several interdependent libraries available in the Eclipse platform. Nevertheless, any model in the AToM^3 environment can be used in the Eclipse GMF environment using the standard XMI interchange format. The meta-model for a DSML is built directly in AToM^3's model editor using its class diagram formalism. The constraints on the concepts of this meta-model are defined using ALLOY facts. Using this information and a description of the concrete visual syntax (specified in an icon editor) for a modelling language, AToM^3 synthesizes a visual model editor for the DSML. The partial model can be built and edited in the generated model editor and the designer can ask for recommendations for possible automatic completions. The partial model is automatically transformed to a constraint called an ALLOY predicate. This ALLOY predicate is then augmented to the ALLOY model to give the whole set of constraints. We invoke a solver on this this set of constraints to obtain zero/one/more complete model(s).

In Section II we present related work. An overview of our methodology is presented in Section III. One of the key parts of our methodology is the automatic synthesis of domain-specific model editors from their specification comprising of the meta-model and visual syntax. This process is described in Section IV. The component that will add model completion ability to the synthesized model editor is a transformation from the meta-model and partial model to an ALLOY model. We present this transformation in Section V. Once we include this transformation into the synthesis of a domain-specific model editor we are able to synthesize domain-specific model editors with automatic model completion. We describe the model completion process in Section VI. We present examples of model completion recommendations generated for partial models in Section VII. We conclude in Section VIII.

II. RELATED WORK

Language-directed editors have been around for since the early 1980s. Some of the well-cited research on language-directed editors are Mentor [12], Interlisp [34], Program Synthesizer [33], Rational [6], PECAN [28], and Gandalf [16]. Most of the existing language-based editors such as in Eclipse are based on *attribute grammars* [15]. These systems have been widely adopted and integrated in many editors for tasks such as syntax highlighting and syntax-directed editing. The *openArchitectureWare* [1] framework , based on the Ecore [14] meta-modelling framework, supports automatic sentence completion already implemented in Eclipse to help make recommendations to sentences in textual domain-specific modelling languages. These suggestions for sentence completion are based on the textual syntax of the modelling language and do not consider the complete consistency of the model with respect to the meta-model and constraints of the language.

In *Model Driven Engineering* (MDE), models built in domain-specific model editors pose a new challenge. The challenge is to complete a partial model specified in the model editor. This involves the editor to use domain-specific modelling language constraints to direct the completion of the partial model. Simply put, this involves constraint solving using knowledge described in the partial model to synthesize a model that conforms to the domain-specific modelling language. Constraint solving for model synthesis has been well-studied in the literature such as model design space exploration [31], partial model completion using Prolog [29] and constraint logic programming [22]. In [30] we present a model completion system in a domain-specific editor by combining knowledge from the meta-model and the partial model specified in the model editor to SWI-Prolog. The Prolog program is solved using a backtracking based solver to return results to the domain-specific environment which was

originally synthesized by AToM^3 using the meta-model. Our methodology is valid for any domain-specific modelling language in the limits of first-order Horn clause logic of SWI-Prolog. However, their primary limitation is that the number of objects in the complete model is equal to the number of objects in the partial model. No new objects are suggested by the model completion system and the user is limited to specifying only the correct number of objects in the partial model. This is primarily due to the fact that constraints are specified at the object property level in SWI-Prolog and not at the meta-level such as on sets of objects.

We identify the need to develop a model completion system that can automatically suggest complete models especially for DSML meta-models containing constraints both on sets of objects and their properties. This typically involves mapping of a meta-model and constraints based DSML specification to a mathematical formalism with tool support that solves constraints to give correct instances of the DSML. Notably, such instances should contain the network of objects (original object identities need not be preserved) specified in the partial model and additional objects (if required) with appropriate property values such that the complete model conforms to its DSML. We would also like to control the maximum size or scope of the complete model for practical time considerations. Transformation of meta-models expressed in UML/OCL [27] to various formal systems is not new [9] [4] [19] [24] [2] [8] [23]. In [9] the authors present a transformation from UML Class Diagrams to Description Logics. Their approach is theoretically rigourous where a knowledge base in description logic on its variants is obtained for a UML Class diagram and theorem provers such as FACT [18] and RACER [35] are used to obtain instances by inferring from the knowledge base. They prove that the time for inference using a description logic representation of an UML Class diagram in EXPTIME-complete. However, their approach does not support transformation of meta-level constraints such as those expressed in Object Constraint Language (OCL) [27] to description logic. An extension of this work for obtaining instances in finite domain is presented in [24]. The transformation of meta-level constraints such as OCL along with UML class diagrams to formal higher-order logic language called Isabelle has been explored in tools such as HOL-OCL [2]. Similarly, we have seen the transformation to constraint programming language ECLiPSE in [19]. Both, these approaches are used primarily for verification of a UML Class Diagram instance against the UMLCD meta-model specification. A constraint in OCL can be verified against an instance of UMLCD but we need the instance itself. In our pursuit to find complete models we need to automatically synthesize instances of a meta-model rather than verifying an arbitrary constraint against an existing instance.

In our approach, we transform the specification of a domain-specific modelling language and the partial model to the common language of first-order relational logic. The target common language is implemented in ALLOY. Transformation of a meta-model specification from UML to ALLOY has previously been explored in the tool UML2Alloy [8] [23] [7]. UML2Alloy supports transformation from meta-model concepts to ALLOY model concepts such as class to signature, property to signature field, operation to function, enumeration/enumeration literal to extends signature, and constraints to predicates. In our approach to transforming a meta-model to an ALLOY model we keep the same transformation format such we transform classes to signatures and properties to class fields. In UML2Alloy composition and aggregation are transformed first to OCL constraints and then to ALLOY. In our tool we transform composition and aggregation in a meta-model directly to ALLOY facts. Our, approach to transforming inheritance is the same as in UML2Alloy. Inheritance is transformed to an ALLOY signature that extends an other ALLOY signature. There is no clear specification in UML2Alloy related articles [8] [23] [7] about transforming multiplicities to ALLOY. In our case we transform multiplicity constraints to ALLOY signature fields in case of occurrence of 0, $0..1$, or $0..*$ multiplicities. If the multiplicity is variable such as $a..b$ we synthesize an ALLOY fact constraining the size of a set of relations. The constraints in meta-model is restricted to a small subset of OCL as UML2Alloy transforms only this subset of OCL to ALLOY. However, in our tool we propose the user to directly enter ALLOY predicates and facts in the ALLOY model giving the user the flexibility of expressing a wider range of constraints (those that have not been implemented in UML2Alloy) such constraints with transitive closure which cannot be expressed directly in OCL. We also present a method to synthesize ALLOY predicates from a partial model. This use of partial knowledge to synthesize complete models greatly reduces model development time. The tool UML2Alloy, does not support the use of partial model knowledge to help generate models.

III. METHODOLOGY OVERVIEW

The development and use of a domain-specific model editor with automatic model completion can be divided into the following phases and sub-phases:

1) Specification of a domain-specific modelling language (in AToM^3)
   a) Specification of a meta-model
      i) Specification of a class diagram (AToM^3 class diagram in our case)
      ii) Specification of facts on the concepts in the class diagram (ALLOY facts in our case)
   b) Specification of a visual syntax in an icon editor (available in AToM^3) for concepts in the meta-model
2) Transformation of meta-model and visual syntax to a model editor
   a) Synthesis of an editor with buttons, menus and icons
   b) Synthesis of a drawing canvas with features such as automatic layout
   c) Synthesis of a clickable widget for model completion

   d) Synthesis of a dialog box for specifying model completion parameters
3) User interaction
   a) Drawing a partial model on the canvas
   b) Editing model completion parameters
   c) Click on a button to generate complete model(s)
4) Model Completion (hidden from user)
   a) Synthesis of base ALLOY model from the AToM^3 class diagram.
   b) Augmenting meta-model facts with base ALLOY model
   c) Synthesis of an ALLOY predicate from partial model and augmentation to base ALLOY model
   d) Synthesis of run commands from the model completion parameters and augmentation to current ALLOY model
   e) Solving final ALLOY model and returning complete models as recommendations to the model editor

The specification of a domain-specific language is usually done by a *language designer* who interacts with domain experts to identify the concepts, their properties and relationships in a domain of knowledge, science or engineering. The language designer also develops a repository of constraints among the concepts and its properties. The assembly of the concepts and relationships is expressed as a Class Diagram (CD) by the language designer. In our methodology we use the AToM^3 CD formalism for this purpose. The constraints on the CD are expressed in a formal constraint language. Preferably, a constraint language that has a finite number of solutions and is decidable. In our methodology we use *facts* expressed in the language ALLOY to represent such constraints. The CD and the set of constraints on it results in the *meta-model* of a Domain-specific Modelling Language (DSML)

A *visual syntax designer* specifies a concrete visual syntax for the concepts and relationships in the modelling language. In our methodology we use the AToM^3 icon editor to specify a visual syntax. In Section IV we discuss in detail the specification of the modelling language for Finite State Machines (FSM) along with a visual syntax.

Once we have all the elements (meta-model and visual syntax) necessary for a domain-specific modelling language a *model transformation engineer* develops a transformation to synthesize a visual domain-specific model editor from these elements. The model editor consists of buttons, menus, and a canvas. A user can select and drop objects on a drawing canvas and connect them using relationships. The objects are manifested as icons as specified in the icon editor for the concrete visual syntax by the visual syntax designer. The relationships are links between these icons. In the model editor by clicking on the icon the user can edit or specify the values of properties.

In our work, we extend this model transformation by transforming the meta-model to an ALLOY model. The transformation also synthesizes a button widget in the domain-specific model editor. A *domain expert* or *user* can click on this button resulting in the solving of the ALLOY model augmented with ALLOY predicates synthesized from the partial model drawn on the canvas. Recommendations as one or more complete models (if found) are returned to the model drawing canvas. In Section V we present the transformation from the meta-model and the partial model to ALLOY. An illustrative outline of the model completion methodology is shown in Figure 1.

One of the key benefits in using ALLOY is its ability to handle constraints specified on a set of objects using quantifiers such as *some*, *all*, *one*, and *lone*. The new system allows us to synthesize models with new objects (if required) to ensure that the complete model is consistent with its meta-model which normally contains quantified constraints on sets of objects of different classes. The solver finds the closest complete model in a given scope and returns the result as a recommendation for model completion.

The methodology described in the paper supports any meta-model with the following characteristics:

- With classes containing attributes of the basic types Integer, String, and Boolean
- With classes containing references with variable multiplicities such as $a..b$ to other classes in the meta-model
- Constraints on the concepts in the meta-model that are expressible as ALLOY facts and predicates. These facts and predicates are constraints in first-order relational logic with quantifiers.

We currently do not support floating-point attributes in meta-models. Although such properties can be approximated as integers. We also do not support the entire range of characters in unicode for string attributes. We either model strings as a sequence of characters or as an integer value. Higher-order constraints where a constraint is specified on another constraint is not supported. In a constraint language such as ALLOY this transforms to the expression of the unsupported higher-order relations (a tuple of relations) instead of supported first-order relations (a tuple of atoms).

IV. SPECIFYING A DOMAIN-SPECIFIC MODELLING LANGUAGE

In this section we explain the steps taken to declaratively specify a domain-specific modelling language. We use Finite State Machines (FSM) as a running example for a modelling language. The FSM modelling language is a visual language with circles representing states and directed arrows representing transitions between states. To define a modelling language and to generate a visual model editor from it requires three inputs:

1) A Meta-model as an AToM^3 Class Diagram (CD)

TABLE I
DOMAINS FOR PRIMITIVE DATATYPES

Type	Domain
Boolean	$\{0, 1\}$
Integer	$\{MinInt, .., MaxInt\}$
String	$\{"a", "b", "c", "event1", .., \}$

2) A Set of Alloy Facts on the concepts in the CD
3) A Visual Syntax

We briefly describe the specifications of these aspects of the FSM language in the following sub-sections.

A. Meta-model

A model consists of objects and relationships between them. The meta-model of the modelling language specifies the types of all the objects and their possible inter-relationships in a specific domain. The type of an object is referred to as a *class*. The meta-model for the FSM modelling language is presented in Figure 2. The classes in the meta-model are FSM, State and Transition.

In this paper we use the Class Diagram formalism in AToM^3 for specifying a meta-model. The Class Diagram formalism can specify itself and hence exhibits the property of *bootstrapping*. We use the visual language notation of class diagrams to specify the meta-model for the FSM modelling language in Figure 2.

Each class in the meta-model has *properties*. A property is either an *attribute* or a *reference*. An attribute is of a primitive type which is either Integer, String, or Boolean. For instance, the attributes of the class State are isInitial and isFinal both of which are of primitive type Boolean. An example domain of values for the primitive attributes is given in Table I. The String variable can be a finite set consisting of a null string, and finite length strings that specify a set of strings. In this paper, we consider a finite domain for each attribute.

Describing the state of a class of objects with only primitive attributes is not sufficient in many cases. Modelling many real-world systems elicits the need to model complex relationships such as modelling that an object contains another set of objects or an object is related to another finite set of objects. This set of related objects is constrained by a *multiplicity*. When a class is related to another class, the related classes refer to each other via *references*. For instance, in Figure 2 the classes State and Transition refer to each other via references annotated with unidirectional relationships. The multiplicity constraints are also annotated with the relationship.

Containment relationships exist between the FSM class and the State and Transition classes. The black-fill ended arrow at the FSM Class signifies that all objects of type State and Transition are always inside an FSM object. There is exactly one FSM object that contains all State and Transition objects.

Apart from attributes and references, objects can inherit properties from other classes. The attributes and references of a class called a *superclass* are inherited by *subclasses*. Similarly, a subclass inherits the references in the superclass. There is no inheritance in our FSM meta-model, nevertheless we consider transformation of inheritance relationships in the transformation presented in Section V.

B. Constraints on Meta-model

Constraints on a meta-model are not always conveniently specified using diagrams such as CDs. They are better expressed in a textual constraint language who's semantics have no side-effect (does not change the state of an object or structure of the model) on the meta-model or its instances (models). The OMG standard for constraint specification is the Object Constraint Language (OCL); however, in our current work we use first-order relational logic statements in the form of ALLOY facts. In this paper, we transform OCL constraints to ALLOY manually. In future, we intend to transform a subset of OCL to ALLOY using an automatic model transformation. This is a challenging task as OCL and ALLOY are very different in terms of what they express. OCL is a query language with no side-effects while ALLOY enforces certain values on objects and properties such that facts are always satisfied (if consistent). OCL is a language in higher-order logic without a thoroughly well-defined semantics. ALLOY on the other hand is a first-order relational logic language with quantifiers that has well-defined semantics for analysis. Bridging this gap in expressiveness is the hurdle in completely automating the OCL to ALLOY transformation. Therefore, we directly present ALLOY facts specified on a meta-model's objects and properties. A brief introduction to the ALLOY language is given in Section V-A.

In Table II we present the constraints on the FSM meta-model in natural language and as ALLOY facts.

The ALLOY signatures [10] used to specify the ALLOY facts are extracted from the FSM meta-model CD. This transformation is discussed in Section V. In the appendix we present the complete ALLOY model for the FSM modelling language. This ALLOY model can be loaded into the ALLOY Analyzer [20] for directly obtaining valid FSM models.

TABLE II
CONSTRAINTS IN NATURAL LANGUAGE AND AS ALLOY FACTS

Constraint Name and Definition	Alloy Fact
exactlyOneFSM: There must be exactly one FSM object in a FSM model	`fact exactlyOneFSM { one FSM }`
atleastOneFinalState : There must be at least one final state in a FSM model	`fact at leastOneFinalState { some s:State|s.isFinal==True }`
exactlyOneInitialState : There must be exactly one initial state in the FSM model	`fact exactlyOneInitialState { one s:State|s.isInitial==True }`
sameSourceDiffTarget : All transitions with the same source must have different target	`fact sameSourceDiffTarget { all t1:Transition,t2:Transition| (t1!=t2 and t1.source==t2.source) => t1.target!=t2.target }`
setTargetAndSource : The target of an incoming transition to a State itself and the source of all its outgoing transitions is the same State	`fact setTargetAndSource { all s:State | s.incomingTransition.target = s and s.outgoingTransition.source=s }`
noUnreachableStates: There can be no unreachable states in the FSM from an initial state. Since, its a ternary constraint we approximate it by stating that a non-initial state can be reached from an initial state up to a maximum depth of N (N=3 is the given example).	`fact noUnreachableStates { all s:State| (s.isInitial==False) => #s.incomingTransition >=1 and (s.isInitial==True and #State > 1) => #s.outgoingTransition >=1 and s.outgoingTransition.target!=s }`
uniqueStateLabels : All State objects have unique labels	`fact uniqueStateLabels { #State>1 => all s1:State,s2:State | s1!=s2=>s1.label != s2.label }`

C. Visual Syntax

The final step (in specifying a DSML for synthesizing a model editor) we take is to specify the concrete visual syntax of the class of objects in the meta-model. The visual syntax specifies what an object looks like on a 2D canvas. An icon editor in AToM^3 is used to specify the visual syntax of the classes in the meta-model.

An icon editor is used to specify the visual syntax of meta-model concepts such as classes and relationships. The icon for State is a circle annotated with three of its attributes (isFinal, isInitial, and label). The connectors in the diagram are points of connection between State objects and Transition objects.

The visual syntax can also by dynamically changed based on the properties of the model. In an iconic visual modelling language such as FSM, the first step taken in specifying a visual syntax is drawing an icon that represents a class of objects. If needed it is annotated with text and its properties. Connectors are added to the visual object so that it can be connected to other objects if they are related.

V. Transformation from Meta-model and a Partial Model to Alloy

The transformation of a meta-model consists of transforming the Class Diagram of a domain-specific modelling language to ALLOY signatures and specification of other constraints as ALLOY facts. We first describe ALLOY and then explain our transformations using relevant examples in the following sections.

A. The ALLOY Language

We present ALLOY [20] [10] as a common constraint language equipped with a solver to help complete partial models. It is founded on quantified first-order logic with support for operators in relational calculus. The domain of a constraint model in ALLOY is given by a set of types called signatures. These signatures may contain a number of fields. Constraints may be added as facts or predicates to the ALLOY model to express additional invariants. ALLOY can be used to perform two kinds of analysis. First, we can solve a predicate to generate model instances that satisfy the facts in an ALLOY model. Second, we can use ALLOY to generation *counter-examples* for assertions that we assume to be true. These, analyses rely on the small scope hypothesis ([10]): only a finite subspace is searched based on the assumption that if there is an instance or a counterexample there is one of small size. The rest of the article contains several commented or described ALLOY listings to give the reader a better intuition of the language.

B. Transformation of a Class Diagram

The transformation of a class diagram consists of the transformation of a class and its properties, transformation of the multiplicity constraints in the class diagram to ALLOY facts, transformation of containment constraints to ALLOY facts and finally the transformation of inheritance to an ALLOY extends relationship between ALLOY signatures. This results in a base ALLOY model. We discuss these transformations in detail below.

1) Class and its Properties: A class is transformed to an ALLOY signature. Consider the State class in the FSM meta-model in Figure 2. This class is transformed to the following ALLOY signature:

```
sig State
{
  label: Int,
  outgoingTransition: set Transition,
  incomingTransition: set Transition,
  fsmCurrentState: one FSM,
  fsmStates: one FSM,
  isFinal:one Bool,
  isInitial:one Bool
}
```

The class contains *properties* which are either *attributes* or *references* to other objects. We see how these properties are transformed to ALLOY using our State class example. The *attributes* of the class are transformed to *fields* in the ALLOY signature. For instance, the *label* attribute of the State class is transformed to an ALLOY field of type *Int*. The ALLOY implementation contains a built-in implementation for the definition of integers (*Int*) for which a domain size can be set using a bit length. The attributes *isFinal* and *isInitial* are set as fields of type *Bool* (boolean values). ALLOY also contains a built-in implementation of *Bool*. There are several operations defined on the *Bool* and *Int* types in ALLOY which allow us to specify several complex constraints on such properties. An attribute of type String is transformed to a String type we define as follows in ALLOY:

```
abstract sig Character{}
one sig A extends Character {}
```

```
one sig B extends Character {}
...
sig String
    {
      content : seq Character
    }
```

First, we describe an abstract signature Character and characters such as A,B,C,D,... that inherit from Character. However, the language in ALLOY prevents the use of digits. A substitute will be to use D1, D2,... for the digit 1,2,... respectively. A String signature is now defined as a sequence of characters. In this paper, we consider integers as simplified abstractions for strings and replace all possible string types to integers. After all, characters have a numerical encoding in a computer. To specify constraints on String objects also requires the specification of ALLOY functions on the String class.

The *references* of a class are also transformed to fields in the ALLOY signature of its containing class. For instance, in the State object the references *outgoingTransition* and *incomingTransition* are transformed to an ALLOY field expressing a set of Transition instances. The *set* keyword indicates that there can be $0..*$ Transition objects related to the State object. The other quantifiers like *set* are *one* meaning exactly one and *lone* meaning $0..1$ objects. For instance, the reference *fsmStates* to *one FSM* object indicates that there must be exactly one container FSM for all State objects.

2) Multiplicity: If we want to specify a reference to a set of objects with multiplicity between *a* and *b* number of objects we synthesize an ALLOY fact. For instance, if we want to specify that a State object has at least 3 and at most 7 Transition object references as incoming transitions, we synthesize the following fact:

```
fact multiplicity_incomingTransition
{
 #State.incomingTransition >=3 and #State.incomingTransition<=7
}
```

3) Containment: We synthesize an ALLOY fact from a containment relation between two classes. For example, the fact that all Transition objects are contained in the FSM object is expressed as the following ALLOY fact:

```
fact containmentTransition
{
 Transition in FSM.transitions
}
```

All Transition objects in a complete model of the FSM modelling language now have a containment relation with the top-level FSM object.

4) Inheritance: Inheritance between a superclass and subclass in a meta-model is transformed to an *extends* relation. The current FSM meta-model used as a running example in the paper does not contain an inheritance relationship. However, imagine a meta-model with an abstract class AbstractState which is inherited by two subclasses State and Composite. We first synthesize an *abstract* signature for the AbstractState class such as:

```
abstract sig AbstractState
{
 label: Int,
 outgoingTransition: set Transition,
 incomingTransition: set Transition,
 container: lone Composite,
 fsmCurrentState: one FSM,
 fsmStates: one FSM
}
```

The State class which inherits from this AbstractState class looks like:

```
sig State extends AbstractState
{
 isFinal: one Bool,
 isInitial: one Bool
}
```

The attributes isInitial and isFinal attributes are transformed to fields in ALLOY. The field isFinal : one Bool implies that all State objects have an isFinal property that can be either True or False. Similarly, the isInitial field in all State objects can be either True or False. These fields model an intial state, final states, and regular states in an FSM.

C. Specifying ALLOY facts on the Meta-model

We specify ALLOY facts using the names of the classes and properties on the AToM^3 class diagram in a text-box called *Textual Meta-information* in the meta-modelling language canvas. The specification of facts in ALLOY is possible because the ALLOY model we generate from the AToM^3 class diagram preserves the names of classes and properties enabling the use of the ALLOY constraint language for specification of constraints on the meta-model. All the facts in Table II are specified on the FSM class diagram. During the model completion process these facts are augmented to the base ALLOY model. This gives us a complete description of the meta-model.

D. Transformation of a Partial Model

We define a partial model as a graph of objects such that: (1) The objects are instances of classes in the modelling language metamodel (2) The partial model either does not conform to the language metamodel or its invariants expressed in a textual constraint language. A complete model on the other hand contains all the objects of the partial model and additional objects or property value assignments in new/existing objects such that it conforms both to the metamodel and its invariants.

A partial model, such as in Figure 3 (a), is *automatically* transformed to a set of ALLOY predicates by navigating it object by object in the canvas. We navigate all objects of a certain type and put them together as an ALLOY predicate. We want to keep the already specified properties for each object in the partial model but also allow for extensibility. For instance, for all the State objects in the partial model of Figure 3 (a) we create an ALLOY predicate as shown in the first predicate of Figure 3 (b). The ALLOY predicate states that there exists at least one State object s1, at least one State object s2, at least one State object s3, at least one State object s4 (representing the four State objects in the partial model), at least one Transition object t1, and at least one Transition object t2 such that s1,s2,s3,s4 are not equal and t1,t2 are not equal. The predicate also states that the Transition objects t1 and t2 are in the set of outgoing transitions for State object s1. Transition object t1 is in the set of incoming transitions of s1. The Transition object t2 is in the set of incoming transitions of s2. These sets are open for inclusion of new Transition objects. These predicates preserve all knowledge coming from the partial model while allowing the extension to relations to more objects.

We present a procedure to describe the transformation from the partial model to a set of ALLOY predicates below:

The following represents the procedure to synthesize an ALLOY predicate from a partial model

1) We start by synthesizing the header of a partial model:
   `pred partialModel {`
2) For all objects of o_{ij} of type $Class_j$ in a partial model we synthesize an ALLOY expression:
   some $o_{ij} : Class_j, ... \mid$
3) For all objects of o_{ij} of type $Class_j$ and all objects o_{kj} of type $Class_j$ in a partial model we synthesize an ALLOY expression:
   $o_{ij} != o_{kj}$, each expression separated by and
4) For all defined attributes a_{ijk} of o_{ij} we synthesize the expression:
   $o_{ij}.a_{ijk} = v$, where v is the specified value separated by commas
5) For all defined references r_{ijk} of o_{ij} we synthesize the expression:
   v in $o_{ij}.r_{ijk}$, where v is the object in the set of referred objects separated by commas
6) We finish the predicate by closing the brace.

E. Transforming ALLOY Model Completion Parameters

The user is provided with a dialog box to insert *model completion parameters*. Model completion parameters include finite scopes such as the upper bound on the number of objects of any class, or the upper-bound on the number of objects for each class, or the exact number of objects for each class, or a mixture of upper bounds and exact number of objects for different classes. The default scope is number of objects in the partial model. An other parameter is the number of solutions required S. This information is used to synthesize an ALLOY *run command* that is finally inserted in the ALLOY model. For example, if the partial model predicate is called *partialModel1* and the user states that he wants exactly one object of class A, up to 10 objects of class B, and a scope of 5 for integers then the following run statements is synthesized:

```
run partialModel1 for exactly 1 A, 10 B, 5 Int
```

If the number of objects in the partial model is N, then the default run command the editor generates is:

```
run partialModel1 for N
```

VI. MODEL COMPLETION PROCESS

The model completion process integrated in the domain-specific model editor takes as input the AToM^3 class diagram, augmented ALLOY facts, and a partial model drawn in the model editor synthesized from the class diagram of a modelling

language, and set of parameters to define the scope of the complete models to be synthesized. The process is invoked when a user draws a partial model in the modelling canvas and clicks on the *Generate Completion Recommendations* button.

The following steps are executed during the completion process:

1) An ALLOY model (ALS) file is synthesized containing the signature definitions of the classes in the AToM^3 class diagram and the facts corresponding to the multiplicity and containment constraints as described in Section V-B
2) The modelling language facts are augmented to the ALLOY model. These facts are specified as described in Section V-C
3) The partial model drawn in the model editor canvas is transformed to a predicate as described in Section V-D and augmented to the current ALLOY model
4) The model completion parameters are transformed to a run command (See Section V-E) and augmented to the ALLOY model giving us an adequate description for model completion.
5) An ALLOY API call is made to load the ALLOY model, parse it and store signatures, facts, predicates, run command as objects in memory.
6) An ALLOY API call is made to transform the first-order relational logic signatures, facts, predicates and model completion parameters to a finite Boolean CNF.
7) We then use the ALLOY API to invoke a third-party SAT solver such as ZChaff [36] or Minisat [25] [26] to solve the Boolean CNF.
8) If a solution exists, then it is stored as an XML file. If a number of solutions (given by model completion parameter S) are required then we attempt to produce S solutions and store them in XML Files.
9) The editor then loads and parses these XML files. The complete model recommendations are presented in the domain-specific model editor in their concrete visual syntax. The user can click on *Next* to scan through all possible completions. The results of model completion are also stored as model files representing graph structures in a temporary directory. They can be loaded into the model editor when required or used by another program for transformation.

It is important to note that the partial model is specified as a source of knowledge about what objects and properties that the user wants to absolutely see in the complete model. In the complete model we can see the intact contents of the partial model. However, the object identifiers of the partial model are not preserved in the complete model. We also do not perform pattern matching to identify the original partial model in the complete model, although such a mechanism can be incorporated if needed. In the default case we find the nearest-consistent complete model(s) to a given partial model.

If a solution is not found the ALLOY API returns a *no solution found exception* to AToM^3 (the invoker). We show this result in a dialog box in the AToM^3 environment. In our work we do not debug a partial model to find the exact source of inconsistency. This incurs a computational cost and time as we need to check every partial model predicate expression against the meta-model constraints to see which characteristics of the partial model leads to an inconsistency. We leave it to the user and depend on his/her expertise of the DSML to identify the inconsistent part of the partial model and correct it.

VII. EXAMPLES

In this section, we consider four examples of partial models in the FSM modelling language. The examples are shown in Figure 4 (a), 4 (b), 4 (c) , 4 (d) respectively. The synthesized predicates for these models are shown in Figures 4 (e), 4 (f) and 4 (g) , 4 (h). The example in Figure 4 (a) contains only one State object with none of the properties having been set. The example in Figure 4 (b) contains two State objects and a Transition object not connected. In Figure 4 (c) we consider a more complex model with several State and Transition objects with some properties set and some not. Finally, in Figure 4 (d) we present a model containing at least two State objects with *isInitial* set to True.

We perform the model completion of these models using two methods of setting parameters for completion:

- *Scope* : Here we specify a scope as a model completion parameter. The scope is a unique number that defines the maximum number of objects for all concepts in the meta-model. We choose the default scope to be 10. The corresponding ALLOY run statement generated is:

```
pred partialModel {}
run partialModel for 10
```

  The *partialModel* predicate is empty and is simply used to obtain a complete model instance. We solve for up to a scope of 10 objects for each concept in the meta-model.

- *Exact Number and/or Scope* : Another mechanism to complete a model is to specify the exact number of objects and/or scope for objects we expect in the complete model.

```
pred partialModel {}
run partialModel for exactly 1 FSM, exactly 5 State,
exactly 10 Transition, 5 int
```

  Here we find a solution for a partial model containing exactly 1 FSM object, exactly 5 State objects, exactly 10 Transition objects. Finally we set a bit-width for integers which is 5. This means that all integers range between -2^5 to 2^5.

All the above parameters were initially set in the synthesized AToM^3 modelling environment. The user is only exposed to the graphical syntax of the concepts in the meta-model and with a text-box to specify the exact number of objects or a scope. The

TABLE III
MODEL COMPLETION TIMES

Partial Model	**Description (I=Inconsistent)**	$Time_{Scope}$ (s)	$Time_{Exact}$	$Time_{ScopeScaled}$	$Time_{ExactScaled}$
Fig. 4 (a)	Only one State object with no properties specified	1.283	0.447	118.045	32.002
Fig. 4 (b)	Two State objects and one Transition object	1.289	0.496	115.994	31.488
Fig. 4 (c)	Several State and Transition objects with some properties specified and some not	1.315	0.575	11.4301	32.517
Fig. 4 (d)	Several State and Transition objects with two initial State objects	1.291 (I)	0.402 (I)	111.352 (I)	31.734 (I)

model completions were performed on a Macbook Pro laptop with an Intel Core 2 Duo processor running at 2.6 GHz clock speed and with 2 GB of RAM. We use the ALLOY analyzer API to invoke the SAT solver Minisat [25] [26] from Chalmers University to solve the Boolean CNF synthesized from the ALLOY model. The time to obtain the solutions for the four partial models for the completion parameters is presented in Table III.

We show the complete models themselves in Figure 5 with a scope of 10. Normally, there is more than one solution to a model completion. We show one of the possible solutions. We do not show that the complete models synthesized for the exact number of objects due to large size of the models. However, it is interesting to note in Table III that the time taken to synthesize models with the exact number of objects specified for each class is a lot faster even though the models are larger. This is because the additional knowledge about the number of objects makes the search space of the models much smaller, therefore allowing us to obtain a solution faster.

The complete model in Figure 5 (a) satisfies all the meta-model constraints such that the single State label has a unique value 7. There is at least one final state and exactly one initial state. In addition, the complete model contains a Transition object of the State to itself with an event 7. This new object added to the complete model does not violate any of the knowledge already present in the partial model.

The second complete model in Figure 5 (b) originally was a partial model with two State objects and a Transition object. The complete model now contains two final State objects and exactly one initial State object. There is also an inclusion of a Transition object in the complete model. The synthesized model conforms to all meta-model constraints.

The third complete model in 5 (c) contains a complex partial model with additional objects that preserve the knowledge in the partial model. We can scale up to a model with several hundred atoms using ALLOY to obtain results in a reasonable amount of time (for online user interaction with the modelling environment). An atom consists of any non-divisible entity in the ALLOY model. This includes objects and their properties connected via relations.

The fourth partial model in 4 (d) consisted of two initial State objects which is not permitted by the meta-model constraint which states that the FSM meta-model must contain only one initial State object. Therefore, the SAT solver was unable to find a complete model that could take into account the partial model.

A. Scalability Concerns

In this section we address the question of scalability in our approach. We identify two forms of scalability:

1) Scalability in the size of the meta-models such as in the number of classes, properties and complexity of constraints.
2) Scalability in the size of the models that conform to a meta-model.

We first consider the scalability of a DSML meta-model. A meta-model can be scaled in terms of the concepts in the class diagram. This includes number of classes, number of attributes in a class, and number of references in a class. We can also scale the implicit constraints in the meta-model class diagram such as number of inheritances, containment relations, and range of multiplicities. Finally, we can scale a meta-model based on the number of facts and arity of each fact on the concepts in the meta-model and the coupling between these constraints. We cannot define the notion of scalability as an increase in the

number of classes or attributes to observe greater time to solve. This is because change in constraints on classes and attributes can also lead to higher solution times. Even a simple language such as FSM contains some hard to solve constraints such as *noUnreachableStates*. This constraint states that any non-initial state will have to be connected to the initial State via at most three transitions. This is a finite approximation of the constraint that there can be no unreachable states in an FSM. The time taken to solve the partial model increases considerably if we increase the depth to 5. The scaled constraint is:

```
fact noUnreachableStates
{
    all s: State | (s. isInitial == False) =>
    all inc1 : s.incomingTransition |
    inc1.source.isInitial = True or
    all inc2 : inc1.source.incomingTransition
    | inc2.source.isInitial = True or
    all inc3 : inc2.source.incomingTransition
    | inc3.source.isInitial = True or
    all inc4 : inc3.source.incomingTransition
    | inc4.source.isInitial = True or
    all inc5 : inc4.source.incomingTransition
    | inc5.source.isInitial = True
}
```

The numerical results for model completion are summarized in Table III. In the columns $Time_{Scope}$ and $Time_{Exact}$ we present the time taken to solve a model completion for the unscaled meta-model with *noUnreachableStates* having a depth of 3. In the columns $Time_{ScopeScaled}$ and $Time_{ExactScaled}$ we present the time taken to solve a model completion for a scaled meta-model with *noUnreachableStates* having depth of 5. In general, when the exact number of objects ($<= scope$) is specified the time taken to solve a model is less than the time taken for a given scope. This is because, the solver knows in advance about the number of objects needed to satisfy all requirements in the model. While, when a scope is specified the solver starts from attempting to solve models with no objects for each signature and incrementally increases the number of objects of each signature until a solution is found. The time taken to solve a partial model for scaled meta-model is about 90 times that of time taken to solve a partial model expressed in an an unscaled version of the meta-model. These results are obtained by averaging and dividing the values shown in columns $Time_{ScopeScaled}$ and $Time_{Scope}$ for scope of 10. For exact values the time taken is about 70 times higher for the scaled version of the meta-model. These results are obtained by averaging and dividing the results in the columns $Time_{Exact}$ and $Time_{ExactScaled}$.

We summarize the discussion on the scalability of our approach to variations in meta-model structure. The scalability of this approach cannot be defined based on general rules that increase in the number of classes or properties will increase the time to find solutions. This we see due to the fact that simply increasing the depth of the *noUnreachableStates* constraint for a 3 class language such as FSM led to a many-fold increase in solution time. There is a complex inter-dependence between the classes, its properties, constraints on them, and the partial model. A thorough experimental study needs to be performed for a specific language where each factor such as number of classes, properties, relationships and constraints are varied over several ranges of values. This can then give us an idea about how a specific language performs using our approach for model completion. This study is out of the scope of our paper as we propose a first solution applicable to any modelling language.

The scalability of the size of the model instances of a meta-model is the second form of scalability. The question is for a given fixed DSML, what is the largest model we can obtain in a reasonable amount of time? For example, we synthesize a complete model from Partial Model 3 using the parameters: exactly 1 FSM object, exactly 8 State objects, exactly 10 Transition objects, 5 Int. The solution was found in 72930 ms (= 73 s). For interactive model completion 73 s could be too much when compared to almost instantaneous code completion in tools such as Eclipse. However, we give the user full control over the model completion parameters. With experience, he can put the correct number of objects he is looking for in the complete model resulting in a faster feedback. When our approach is used for automatic model synthesis such as for automatic test model generation [32] or for design space exploration [31] we pay less attention to instantaneous responses. In these cases we obtain several hundred or even thousands of complete and effective models through several hours of model synthesis.

Scaling model completion also provides reflective insight into the design of the language itself. A language with constraints that are highly constrained or not optimally designed can perhaps take a long time to resolve. However, this does not rule out the fact that manual creation of the model will be easier for a user of the language. He/she can also be entangled in the complexity of the constraint while creating his model. Therefore, automatic model completion also reveals design flaws in modelling languages that can be corrected in parallel to obtain reasonable completion times.

VIII. Conclusion

We present a methodology to synthesize domain-specific model editors with meta-model directed model completion for domain-specific modelling languages. Our goal has been to provide model editors with completion capabilities similar to text or code editors in IDEs such as Eclipse or word processors such as Microsoft Word. A potential future application of our approach is generation of test models from partial knowledge. A DSML user draws a partial test model for testing a model transformation and subsequently sets model completion parameters. Then he/she clicks on a button to generate complete test models that are valid test cases for model transformations. Moreover, the model completions are displayed in the concrete visual syntax of the modelling language while evading all the details in the CNF, XML files, or other intermediate low-level representations. This aspect of our tool helps reduce the time to develop models in the modelling environment as the user only works in his domain language. The user does not need to manually transform his models to a different constraint language, solve his models and return the results to the editor anymore since the underlying model completion process is hidden from the user. After all, the goal of MDE is to leverage modelling to the highest possible level of abstraction.

Our approach uses a modelling language metamodel , the syntax, and its *static semantics* in the form of metamodel constraints to perform model completion. However, since the presented approach is modelling language independent we do not consider *dynamic semantics* often realized in a simulator for model completion. Nevertheless, we project several implications to simulation as it goes hand in hand with modelling. Model simulators, such as MATLAB/Simulink for causal block diagrams, often contain *hard-coded declarative constraints or program statements* that check and report on the validity of input models during simulation. For example, a causal block diagram simulator analyzes input models to detect cycles and warns the modeller. These statements that are integrated in simulator code come from heterogenous sources of knowledge such as domain experience, static/dynamic analysis, and testing. This gradual inclusion of model validity knowledge directly into simulator code makes them bulky and slow to execute. This approach also obscures the user from potentially using this knowledge to build correct models. Extracting knowledge from simulators and developing modelling language invariants to guide modellers to create invariant-validated models leverages a *correct by construction* philosophy. Further, using these invariants for automatic model completion of partial models makes the modelling and simulation process less error-prone as models are first checked and then completed to satisfy invariants before simulation.

Our lightweight approach is effective for small yet useful modelling languages. Time to complete models by the state of the art SAT solvers for about 50 objects in the model is not more than a few minutes for FSM. The completion time greatly *depends on the complexity* of the DSML. The time taken to obtain complete models also gives us insight about how restricted a DSML is and how it can be relaxed.

As future work we intend to run thorough performance experiments on a specific industry strength DSML. Such a DSML will have a larger meta-model with a several complex constraints. We will limit ourselves to the confines of first-order relational logic in Alloy as the language to express constraints. We also wish to enlist the set of detailed requirements to synthesize DSML modelling environments with completion. For example, an interesting factor is user interaction time. If a complete model is not returned within a given time then the user can no longer make developments quickly. Other aspects of model completion include completion of models when two or more meta-models are involved, expression of partial models as invariants or constraints, and aiding the user by helping him/her set parameters for model completion.

Appendix I
Alloy Model synthesized from FSM meta-model with Facts and Partial Model Predicates

```
module metamodelFSM

open util/boolean as Bool


    sig FSM
    {
     states:set State,
     currentState: lone State,
     transitions: set Transition
    }

    sig State
    {
     label: Int,
     outgoingTransition: set Transition,
     incomingTransition: set Transition,
```

```
     fsmCurrentState: one FSM,
     fsmStates: one FSM,
     isFinal:one Bool,
     isInitial:one Bool
    }

    sig Transition
    {
     event: Int,
     target: one State,
     source: one State,
     fsmTransitions:one FSM
    }

//Meta-model constraints//

//Exactly one initial state
    fact exactlyOneInitialState
    {
     one s:State|s.isInitial == True
    }

//Atleast one final state
    fact at leastOneFinalState
    {
      some s:State | s.isFinal == True
    }

//Exactly one HFSM
    fact exactlyOneFSM
    {
     one FSM
    }


    fact sameSourceDiffTarget
    {
     all t1:Transition,t2:Transition|  (t1!=t2 and t1.source==t2.source) =>
     t1.target!=t2.target
    }

    fact setTargetAndSource
    {
     all s:State| s.incomingTransition.target = s and
     s.outgoingTransition.source=s
    }

    fact noUnreachableStates
    {
     all s: State | (s. isInitial == False) =>
     all inc1 : s.incomingTransition |
     inc1.source.isInitial = True or
     all inc2 : inc1.source.incomingTransition
     | inc2.source.isInitial = True or
     all inc3 : inc2.source.incomingTransition
     | inc3.source.isInitial = True
    }
```

```
    fact uniqueStateLabels
    {
     #State>1 => all s1:State,s2:State | s1!=s2=>s1.label != s2.label
    }

    fact containmentState
    {
     State in FSM.states
    }

    fact containmentTransition
    {
     Transition in FSM.transitions
    }

//Partial Model Facts

//Partial Model 1

    pred partialModel1_Fact
    {
     some State
    }

//Partial Model 2

    pred partialModel2_Fact
    {
     some s1:State,s2:State,t1:Transition |s1!=s2 and
     t1 in s1.outgoingTransition and t1 in
     s2.incomingTransition
    }

//Partial Model 3

    pred partialModel3_Fact
    {
     some s1:State,s2:State,s3:State,s4:State,
     t1:Transition, t2:Transition|
     s1!=s2 and s2!=s3 and s3!=s4 and s1!=s3 and
     s1!=s4 and s2!=s4 and t1!=t2 and
     t1 in s2.incomingTransition and t2 in
     s3.incomingTransition and t1 in s1.outgoingTransition
     and t2 in s1.outgoingTransition and
     s2.isInitial = True and s4.isFinal = True
    }


//Partial Model 4

    pred partialModel4_Fact
    {
     some s1:State,s2:State,s3:State,s4:State,
     t1:Transition,t2:Transition|
     s1!=s2 and s2!=s3 and s3!=s4 and s1!=s3 and
     s1!=s4 and s2!=s4 and t1!=t2 and
     t1 in s2.incomingTransition and t2 in
     s3.incomingTransition and t1 in s1.outgoingTransition
```

```
 and t1 in s1.outgoingTransition and
 s2.isInitial=True and s3.isInitial=True
}



run partialModel1_Fact for 10
run partialModel2_Fact for 10
run partialModel3_Fact for 10
run partialModel4_Fact for 10
run partialModel1_Fact for exactly 1 FSM, exactly 5 State,
exactly 5 Transition, 5 int
run partialModel2_Fact for exactly 1 FSM, exactly 5 State,
exactly 5 Transition, 5 int
run partialModel3_Fact for exactly 1 FSM,  exactly 5 State,
exactly  5 Transition, 7 int
run partialModel4_Fact for exactly 1 FSM, exactly 5 State,
exactly 5 Transition, 5 int
```

ACKNOWLEDGMENT

We thank the ANR Domino and European S-Cube Projects for research funding.

REFERENCES

[1] http://www.openarchitectureware.org/.
[2] A. D. Brucker and B.Wolff. The HOL-OCL book. Technical Report 525, ETH Zurich, 2006.
[3] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, pages 44–51, November 2001.
[4] A. Queralt and E. Teniente. Reasoning on UML class diagrams with OCL constraints. *Lecture Notes in Computer Science ER 2006, Springer-Verlag*, 4215:497512, 2006.
[5] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmer's text editing. In *CHI 2005*, 2005.
[6] Archer J.E. Jr. and Delvin M.T. Rational's experience using Ada for very large systems. In *The First International Conference on Ada Programming Language Applications for the NASA Space Station*, pages B.2.5.1–B.2.5.11, Houston, Texas, NASA, June 1986.
[7] Behzad Bordbar and Kyriakos Anastasakis. Mda and analysis of web applications. *In Trends in Enterprise Application Architecture (TEAA) in Lecture notes in Computer Science, Trondheim, Norway*, 3888:44–55, 2005.
[8] Behzad Bordbar and Kyriakos Anastasakis. Uml2alloy: A tool for lightweight modelling of discrete event systems. In Nuno Guimares and Pedro Isaas, editors, *IADIS International Conference in Applied Computing*, volume 1, pages 209–216, Algarve, Portugal, 2005. IADIS Press.
[9] D. Berardi, D. Calvanese, and G. D. Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168:70118, 2005.
[10] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, March 2006.
[11] Juan de Lara and Hans Vangheluwe. Atom^3: A tool for multi-formalism modelling and meta-modelling. In *Lecture Notes in Computer Science*, number 2306, pages 174–188, 2002.
[12] Donzeau-Gouge, V. Huet, G. Kahn, and Lang B. *Interactive Programming Environments*, chapter Programming environments based on structured editors: The Mentor experience, pages 128–140. McGraw-Hill, New York, 1984.
[13] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap,. In *FOSE '07: 2007 Future of Software Engineering*, 2007.
[14] Budinsky Frank. *Eclipse Modeling Framework*, volume 1 of *The Eclipse Series*. Addison-Wesley, 2004.
[15] Gail E. Kaiser. Incremental Dynamic Semantics for Language-Based Programming Environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169–193, April 1989.
[16] Habermann A.N. and David Notkin. Gandalf: Software development environments. *IEEE Trans. of Softw. Eng., SE-12*, 12:1117–1127, December 1986.
[17] Hans Vangheluwe and Juan de Lara. Domain-Specific Modelling with AToM3. In *The 4th OOPSLA Workshop on Domain-Specific Modeling*, Vancouver, Canada, October 2004.
[18] Ian Horrocks. Using an expressive description logic: FaCT or fiction? In *6th International Conference on Principiles of Knowledge Representation and Reasoning*, pages 636–647, 1998.
[19] J. Cabot, R. Claris, and D. Riera. Verification of UML/OCL Class Diagrams Using Constraint Programming. In *ICST Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa'2008)*, 2008.
[20] Daniel Jackson. http://alloy.mit.edu. 2008.
[21] Karsten Ehrig, Claudia Ermel, and Stegan Hansgen. Generation of Visual Editors as Eclipse Plug-ins. In *The 20th IEEE/ACM Internation Conference on Automated Software Engineering*, pages 134–143, 2005.
[22] Krysztof R. Apt and Mark G. Wallace. *Constraint Logic Programming with ECLiPSe*. Cambridge University Press, 2007.
[23] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In *MoDELS*, pages 436–450, 2007.
[24] M. Cadoli, D. Calvanese, G. D. Giacomo, and T. Mancini. Finite satisfiability of UML class diagrams by Constraint Programming. In *Int.Workshop on Description Logics (DL2004), CEUR Workshop*, number 104, 2004.
[25] Niklas Een and Niklas Srensson. An Extensible SAT-Solver. In *SAT 2003*, 2003.
[26] Niklas Een and Niklas Srensson. MiniSat A SAT Solver with Conflict-Clause Minimization, Poster. In *SAT 2005*, 2005.

[27] OMG. The Object Constraint Language Specification 2.0, OMG Document: ad/03- 01-07, 2007.
[28] Reiss S.P. Graphical program development with PECAN program development systems. In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30–41, Pittsburg, Pa., April 1984. ACM, New York.
[29] Sagar Sen, Benoit Baudry, and Doina Precup. Partial Model Completion in Model Driven Engineering using Constraint Logic Programming. In *International Conference on the Applications of Declarative Programming*, 2007.
[30] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Domain-specific Model Editors with Model Completion. In *Multi-paradigm modelling workshop associated with MoDeLs 2007*, Nashville, TN, USA, October 2007.
[31] Sandeep Neema, Janos Szitpanovits, and Gabor Karsai. Constraint-Based Design Space Exploration and Model Synthesis. In *Proceedings of EMSOFT 2003, Lecture Notes in Computer Science*, number 2855, pages 290–305, 2003.
[32] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining mullti-formalism knowledge to select test models for model transformaion testing. In *ACM/IEEE International Conference on Software Testing*, Lillehammer, Norway, April 2008.
[33] Teitelbaum T. and T. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.
[34] Teitelman W. and L. Masinter. The Interlisp Programming Environment. *IEEE Computer*, 14(4):25–34, 1981.
[35] Volker Haarslev and Ralf Mller. RACER system description. In *International Joint Conference on Automated Reasoning (IJCAR 2001)*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 701–705. Springer, 2001.
[36] Y. S. Mahajan, Z. Fu, and S. Malik. ZChaff2004: An Efficient SAT Solver. In *Lecture Notes in Computer Science SAT 2004 Special Volume LNCS 3542.*, pages 360–375, 2004.

Sagar Sen Sagar Sen is a 2nd year PhD student at INRIA Rennes Bretagne-Atlantique, France. He obtained his M.Sc. in Computer Science from McGill University, Canada as a Commonwealth Fellow and B.E. from Viveswaraiah Technological University, Bangalore, India. He has held the position of research assistant at the Jawaharlal Nehru Center for Advanced Scientific Research, Bangalore. His current research interests are automated validation in model-driven development and service-oriented software systems.

Benoit Baudry Benoit Baudry is a researcher in software engineering at INRIA Rennes Bretagne Atlantique. He received his PhD in computer science from the University of Rennes, France in 2003. He first worked at CEA (French government nuclear agency) before joining INRIA in 2004. In 2008 he was an invited scientist at Colorado State University. His research interests include software testing, fault localization, aspect-oriented software development, model transformation and model-driven development. He is the vice-chair of the steering committee of the International Conference on Software Testing Verification and Validation.

Hans Vangheluwe Hans Vangheluwe is a Professor of Computer Science at Antwerp University in Belgium and an Associate Professor at McGill University, Montreal, Canada. He received his D.Sc. M.Sc. in Computer Science, and B.Sc. degrees in Theoretical Physics and Education from Ghent University in Belgium. His current interests are in domain-specific modelling and simulation, including the development of graphical user interfaces using the tool AToM3 (A Tool for Multi-formalism and Meta-Modelling), developed in collaboration with Prof. Juan de Lara. Recently, he has become active in the design of Automotive applications.

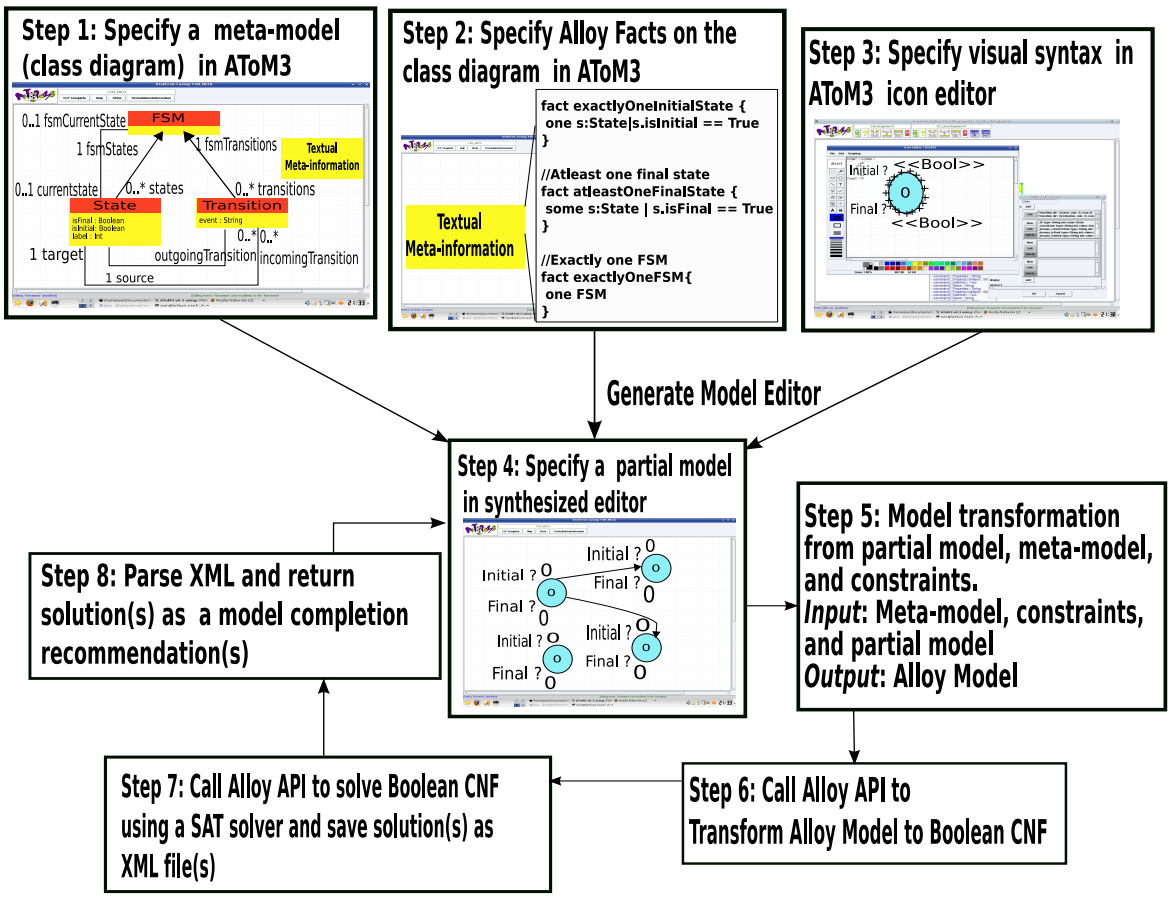


Fig. 1. Methodology Overview

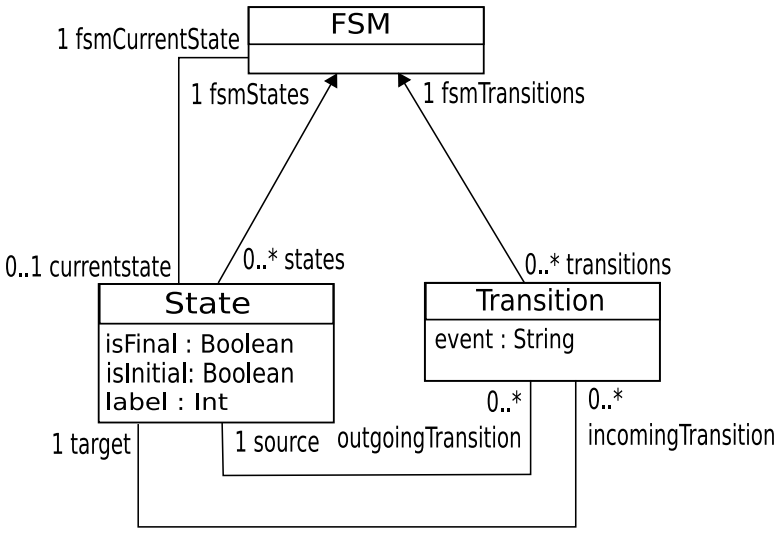
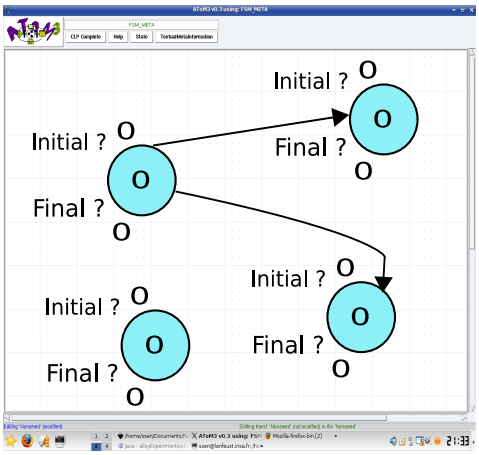


Fig. 2. The Finite State Machine Meta-model



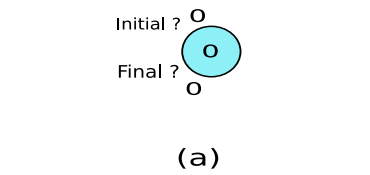
(a)

Synthesized Predicate

```
pred partialModel
{
some s1 : State, s2 : State, s3 : State, s4 : State,
        t1 : Transition, t2 : Transition |
s1 != s2 and s2 != s3 and s3 != s4 and s1 != s3 and
s1 != s4 and s2 != s4 and  t1 != t2 and
t1 in s2.incomingTransition and t2 in s3.incomingTransition
and t1 in s1.outgoingTransition  and t2 in s1.outgoingTransition and
s2.isInitial = True and s4.isFinal = True
}
```

(b)

Fig. 3. (a) Partial Model (b) Synthesized Predicates from Partial Model

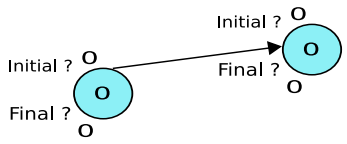


(a)

Synthesized Predicates

```
fact partialModel_Fact1
{
  some State
}
```

(e)

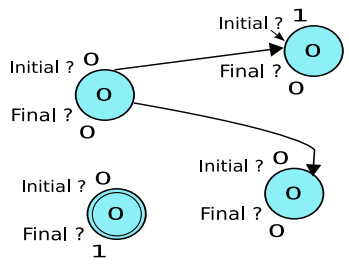


(b)

Synthesized Predicates

```
pred partialModel2_Fact
{
some s1 : State, s2 : State, t1 : Transition | s1 != s2 and
t1 in s1.outgoingTransition and t1 in s2.incomingTransition
}
```

(f)

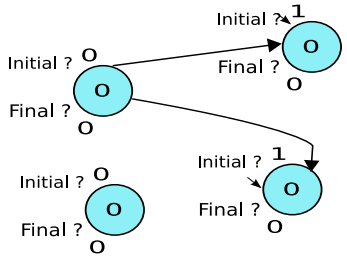


(c)

Synthesized Predicates

```
pred partialModel3_Fact
{
some s1 : State, s2 : State, s3 : State, s4 : State, t1 : Transition, t2 : Transition |
s1 != s2 and s2 != s3 and s3 != s4 and s1 != s3 and s1 != s4 and s2 != s4 and
 t1 != t2 and t1 in s2.incomingTransition and t2 in s3.incomingTransition and
t1 in s1.outgoingTransition  and t2 in s1.outgoingTransition and
s2.isInitial = True and s4.isFinal = True
}
```

(g)



(d)

Synthesized Predicates

```
pred partialModel4_Fact
{
some s1 : State, s2 : State, s3 : State, s4 : State, t1 : Transition, t2 : Transition |
s1 != s2 and s2 != s3 and s3 != s4 and s1 != s3 and s1 != s4 and s2 != s4
and t1 != t2 and t1 in s2.incomingTransition and t2 in s3.incomingTransition
and t1 in s1.outgoingTransition  and t1 in s1.outgoingTransition and
s2.isInitial = True and s3.isInitial = True
}
```

(h)

Fig. 4. (a) Partial model 1, (b) Partial model 2, (c) Partial model 3, (d) Partial model 4, (e) Predicate synthesized for Partial model 1 (f) Predicate synthesized for Partial model 2, (g) Predicate synthesized for Partial model 3, (f) Predicate synthesized for Partial model 4

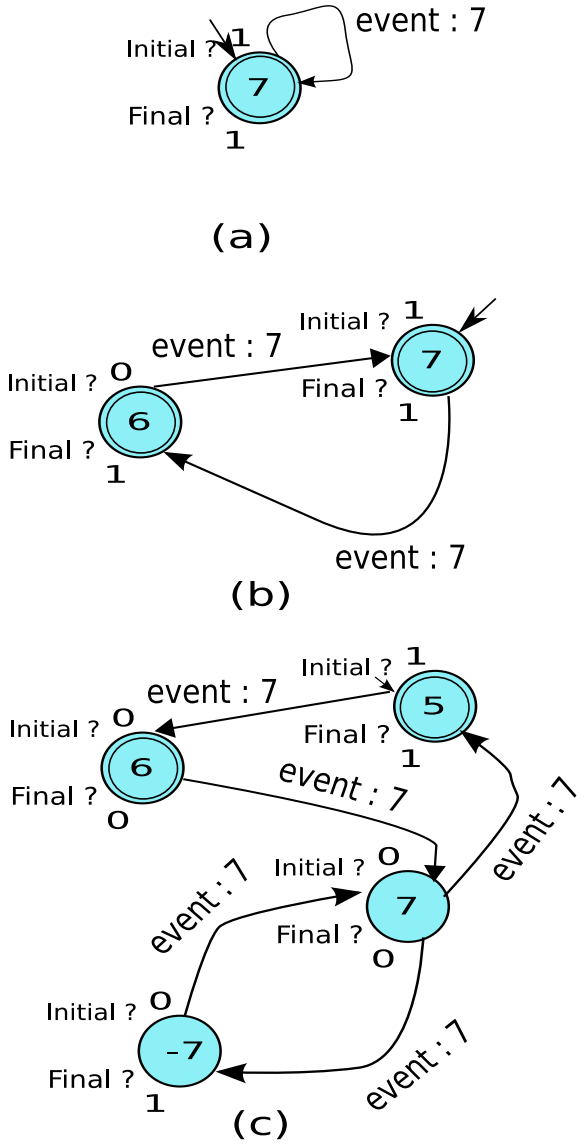


Fig. 5. (a) Complete Model for Partial Model 1 (b) Complete Model for Partial Model 2 (c) Complete Model for Partial Model 3