Functional Alloy Modules

Loïc Gammaitoni and Pierre Kelsen
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi
L-1359 Luxembourg

TR-LASSY-14-02

Abstract. The Alloy language was developed as a lightweight modeling language that allows fully automatic analysis of software design models via SAT solving. The practical application of this type of analysis is hampered by two limitations: first, the analysis itself can become quite time consuming when the scopes become even moderately large; second, determining minimal scopes for the entity types (limiting the number of entities of each type) to achieve better running times is itself a non-trivial problem.

In this paper we show that for the special case of Alloy modules specifying transformations we may be able to circumvent these limitations. We define the corresponding notion of functional module and define precise conditions under which such functional modules can be efficiently interpreted rather than analyzed via SAT solving and we also explain how interpretation of functional Alloy modules can be seamlessly integrated with the SAT-based analysis of other modules. We provide evidence that for complex transformations interpreting functional modules may result in significant time savings.

1 Introduction

Alloy [5] is a formal language based on a first-order relational logic with transitive closure. It is based on a small set of core concepts, the main one being that of a mathematical relation. It was developed to support agile modeling of software designs. It does this by allowing fully automatic analysis of software design models using SAT solving. By providing immediate feedback to users, the use of Alloy is meant to facilitate identifying design errors early.

The analysis of Alloy models using the associated tool, the Alloy Analyzer, has some important limitations though. Analysis is possible only because each type of entity carries an implicit or explicit scope limiting the number of instances of each type, thus permitting exhaustive search of the space of model instances via SAT solving. Two problems hamper the practical application of Alloy:

– Despite many advances in the performance of SAT solvers the analysis can become quite time consuming when the model requires larger scopes to find a suitable instance.
– The problem of finding minimal scopes for the different entity types is itself non-trivial (in fact it is undecidable). This is particularly problematic for complex models with many different entity types.

Because of the undecidability of first-order logic both limitations cited above cannot be eliminated in all cases. It is however reasonable to expect that in some special cases these limitations can be dealt with. This observation is the starting point of the present paper. The main contribution of the paper is to show that in the case where an Alloy module specifies a transformation we may be able to circumvent the two limitations mentioned above by substituting analysis with interpretation.

More precisely we introduce the notion of a functional module that represents an Alloy module specifying a transformation. We show that under certain conditions such a functional module can be efficiently interpreted (instead of being analyzed via SAT solving). We also explain how the interpretation of functional modules can be integrated with analysis of other modules.

The paper is structured as follows. In the next section we introduce the notion of functional Alloy modules and illustrate their use for specifying execution semantics of a software system. In section 3 we show how interpretation is integrated to the analysis provided by the Alloy Analyzer. In section 4 we define a subset of the Alloy language for expressing functional modules. This sub-language allows an efficient interpretation of rules expressed. We evaluate our approach in section 5 . We discuss related work in section 6 and present concluding remarks and future work in the final section.

2 From Transformations to Functional Modules

The goal of this section is to introduce the notion of functional module. This notion is based on the observation that Alloy modules can contain purely structural informations but they may also specify transformations. In our view a transformation represents a mathematical function from instances of an input model to instances of an output model. The Alloy Analyzer analyzes transformations via SAT solving. We propose to compute or interpret these transformations rather than analyzing them via SAT solving.

After considering examples of transformations we will introduce later in this section the notion of functional Alloy module, which is essentially an Alloy module specifying a transformation.

2.1 Examples of transformations

Before we give a precise definition of functional modules we want to provide an example of transformations that could arise in the specification of software systems. Each software system can be viewed from a structural or behavioral viewpoint. The structural part of a software system encompasses the types of entities that compose it as well as the relations linking those entities among each other. These structural features are represented in Alloy by *signatures* (corresponding to the entity types) and their *fields* (corresponding to the relations). The structural description is usually complemented by additional integrity constraints that more precisely define the set of valid instances.

The behavioral aspects of a software design (if present) characterize how the software can react to external stimuli. This typically involves the notion of state: the state is a snapshot in the lifecyle of a (software) system. Assuming we know the sequence of external inputs we could compute the trace of execution of the system, i.e., the sequence of states the system traverses when reacting to those inputs. Here we assume that execution is deterministic. For a given sequence of inputs we can view the relation of the system to the trace (which defines the behavior of the system) as a transformation. Thus in principle it should be possible to compute this transformation.

As a concrete example consider finite state machines (FSMs). In figure 1 the Alloy module defining an FSM is shown. Figure 2 shows the Alloy module describing traces of finite state machines and finally figure 3 specifies how traces relate to a given FSM.

2.2 Functional modules

We are now ready to precisely define the notion of functional module. We will illustrate the formal definition with the example given above. The *projection* of an instance x on a module m consists only of those atoms and links that correspond to signatures and relations of m, respectively.

Definition 1. *A functional module m from m_1 to m_2 is an Alloy module m which imports Alloy modules m_1 and m_2 with the property that for any two instances x_1 and x_2 of m the following holds: if x_1 and x_2 have the same projection on m_1, then they also have the same projection on m_2.*

```
module FSM

sig State{}{this in (Transition.source+Transition.target) }
one sig Start extends State{}{this in Transition.source}
one sig End extends State {}{this  in Transition.target}
abstract sig Symbol{}
one sig A,B,C extends Symbol{}
sig Transition{
    source: State,
    target: State,
    trigger: Symbol
}
fun getNext[ src: State, s: Symbol ]: State{
  (src.~source & s.~trigger).target
}
fact reachableEnd{
  End in Start.^((~source).target)
}
fact oneTransition{
 all st: State| all sy: Symbol|  one t:Transition|
     t.source=st and t.trigger=sy
}
one sig Input{
    s: seq Symbol
}{ s[0]=A && s[1]=B && s[2]=A && s[3]=C && s[4]=B}
```

Fig. 1. FSM module

In other words a functional module m from m_1 to m_2 can be viewed as a mathematical function from instances of m_1 to instances of m_2.

To illustrate this definition, consider the module FSM2Trace describing the relation from FSMs to traces (from figure 3) . If m denotes this module and m_1 and m_2 are the modules specifying the FSM metamodel (from figure 1) and the trace metamodel (from figure 2) respectively, then it is not difficult to see that in this case m is indeed a functional module from m_1 to m_2.

In this paper we will show that under certain conditions such functional modules can be computed (or interpreted) efficiently rather than being analyzed via SAT solving, which may result in substantial time savings. In the next section, we introduce how such an interpretation mechanism can be integrated to the analysis performed by Alloy and in section 4 we explain under which conditions this interpretation can be done efficiently.

```
module Trace
open FSM

sig Trace{
 s: seq State
}
```

Fig. 2. Trace module

```
module FSM2Trace
open FSM
open Trace

one sig Bridge{
    map1 :Input one->lone Trace
}{ all i: Input| value_map1[i,map1[i]] }

pred value_map1( inp : Input ,  out:Trace){
  out.s[0]=Start
  all i:Int |( i>=0 && i <#inp.s ) implies
        out.s[add[i,1]] = getNext[ out.s[i] ,inp.s[i]]
}
```

Fig. 3. FSM2Trace module

3 Integrating Analysis and Interpretation

In this section, we give an overview of how the interpretation of functional modules we propose is integrated with the already existing analysis features provided by the Alloy Analyzer.

We start by formalizing the notion of interpretation. For a module m denote by I_m the set of instances of m and by $\pi_m(x)$ the projection of an instance x on m.

Following definition 1, a functional Alloy module m from m_1 to m_2 can be seen as a function $f : I_{m_1} \mapsto I_{m_2}$.

For integrating functional interpretation with analysis it turns out that it is more suitable to view m as a mapping $g : I_{m_1} \mapsto I_m$. At the code level we represent this function by $Interpret(m, x)$ which takes module m and an instance x of m_1 as inputs and produces an instance y of m as output with the property $\pi_{m_1}(y) = x$.

The pseudo code in fig. 4 describes how analysis is integrated with interpretation. The function *GenerateInstances* generates a set of instances for module m. If m is functional, it recursively computes a set of instances for m_1 and for each of those instances returns an m-instance by calling the *Interpret* function (on line 6). If m is not a functional module, *Generate* simply invokes the Alloy

Analyzer. The implementation of the *Interpret* function will be detailed in the next section.

```
GenerateInstances(Module m) {// return set of instances of m
  if(m is a functional module from m_1 to m_2){
    S= GenerateInstances(m_1)
    let T = emptySet; // set of m-instances
    for each x in S do {
      y = Interpret(m,x);
      T= T U {y}
    }
    return T;
  }
  else // m is not a functional module
    return Analyze(m);// call Alloy Analyzer for m
}

```

Fig. 4. Pseudo code used to integrate the interpretation of functional modules to Alloy analysis.

If m is a functional module, then $Generate(m)$ will generally not return the same set of instances of m as $Analyze(m)$. From the definition of functional modules given in the last section, we know however that the sets returned by these two functions are "transformationally equivalent" in the following sense: for any two instances y and y' of m computed via analysis and interpretation, respectively, we have: $\pi_{m_1}(y) = \pi_{m_1}(y') \Rightarrow \pi_{m_2}(y) = \pi_{m_2}(y')$.

We consider two example applications of the *Generate* function. Calling this function for the FSM2Trace module (from fig. 3) will result in first launching the Analyzer for the FSM module, thus producing a set of FSM instances, and then producing one FSM2Trace instance for each FSM instance using interpretation.

For a more complex example suppose one is interested in defining a visualization for the previous transformation in order to present the FSM and the Trace corresponding to the given input in an intuitive way. The domain of possible visualizations can be defined by a visual language model (VLM). Thus, the visualization of model instances can be seen as a transformation from model instances to VLM-instances [4]. Such a transformation can be expressed by a functional module. In our example, we call this module Trace2Viz and depict the way it is integrated to the other modules in fig. 5. Calling *Generate* for the Trace2Viz module would recursively generate FSM2Trace instances via interpretation, and for each such instance it would produce a Trace2Viz instance, again via interpretation.

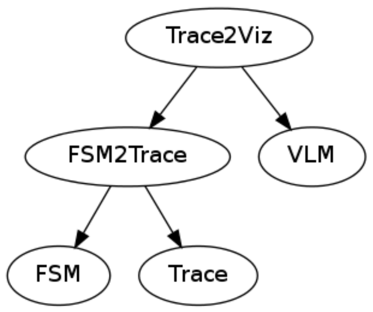

Fig. 5. Trace2Viz importation tree

4 Efficient Interpretation of Functional Modules

In this section, we restrict the syntax for expressing functional modules to a subset of the Alloy language so that an efficient interpretation of the module is possible. Modules expressed in this restricted syntax are still valid Alloy modules and as such are analyzable by the Alloy Analyzer instead of being interpreted.

4.1 Structure Overview

We shall assume that a functional module m from m_1 to m_2 consists of two parts: the first part defines a set of mappings between sets of elements of the input model (m_1) and output model (m_2). The second part contains a set of predicates detailing the mappings from the first part.

The mappings in the first part are defined in Alloy via binary relations (referred to as *maps*) inside a singleton signature (referred to as *Bridge*). We call this part of the functional module its *backbone*.

In fig.6 we provide the Bridge signature used to define the mapping between inputs declared in our FSM example and their corresponding Traces.

```
one sig Bridge{
    map1 :Input one->lone Trace
}
```

Fig. 6. Backbone mapping of the FSM2Trace transformation

The second part of the functional module consists of a set of predicates of two kinds. For each backbone mapping we define a *guard* predicate and a *value* predicate. Guard and value predicates have the particularity to be named after their associated map prefixed by the keyword `guard_` and `value_` respectively.

The body of a guard is the condition under which an element of the input model has an image under the associated mapping. In our FSM example, the guard of map1 (depicted in fig. 7) specifies that for an Input to have an image in map1, its sequence of symbols should not be empty.

```
pred guard_map1( i : Input){
         #i.s>0
}
```

Fig. 7. Guard of the backbone mapping given in fig. 6

Value predicates provide a set of *rules* defining the values of fields for an image of a given input element.

In our FSM example, we define the mapping between inputs and traces to be such that the first element of the trace should be the starting state, and that each next element should correspond to the state targeted by the transition taking its source in the current element and whose trigger corresponds to the current symbol of the input. We express this with the value predicate depicted in fig.8 .

```
pred value_map1( inp : Input ,  out:Trace){
  out.s[0]=Start
  all i:Int |( i>=0 && i <#inp.s ) implies
      out.s[add[i,1]] = getNext[ out.s[i] ,inp.s[i]]
}

```

Fig. 8. Rules defining the properties of the backbone mapping given in figure 6

In the next subsection we introduce the syntax of rules composing value predicates.

4.2 Rule Syntax

Our syntax allows to express three kinds of rules: strict assignment rules, loose assignment rules, and loop rules.

Strict Assignment Rules Those rules are certainly the most straightforward ones. They are used to directly specify the value of a given field for the output element created and are generally constructed as follows:

```
output.field=value
```

Values which can be assigned to output elements this way can be expressed as a function of the input instance or as a function of already processed maps.

We use a strict assignment rule in our FSM example when we define the first element of the trace to be the start state:

```
out.s[0]=Start
```

Loose Assignment Rules Loose assignment rules can be viewed as generalizations of strict assignment rules to the case where a field has multiplicity greater than 1.

As our FSM example does not use any loose assignment rules, we use another example. The following rule aims at expressing that the output element(a given text) should be contained in a given shape.

```
text in shape.contains[Int]
```

Note that this rule does not put any constraint on the position of the text in the sequence representing this containment.

Loop Rules Loop rules consist of universally quantified Alloy expressions.

A loop rule is composed of :

- an initialisation part in which we declare the loop control variable and the set of values it will take over the iterations. The loop control variable can be used in the expression composing the other parts of the loop as it will be replaced by its current value before each part is processed.
- a condition part that defines under which condition the body of the loop has to be executed. This condition can be any Alloy expression that can be evaluated over the input instance.
- a step part that defines the rule to be processed if the condition holds in the input solution.

An example of a loop rule is shown in fig. 9

BNF To summarize the above in a more formal way, we define the syntax to be used in the definition of rules by the following BNF:

```
<rule> ::= <strict> | <loose> | <loop>
<strict> ::= <leftEq> "=" <rightEq>
<loose> ::= <b> " in " <bridgeMap> "." <field>
<leftEq> ::= <b> "." <field>
<field> ::= <f> | <f> "[" <var> "]"
<rightEq> ::= <out> | <ExprIn> | <bridgeMap>
<bridgeMap> ::= "Bridge." <map> "[" <ExprIn> "]"
<map> ::= <mapName> | "(" <mapName> "+" <map> ")"

<loop> ::= "all" <x> : <ExprIn> "|" <condition> "implies" <rule>
<condition> ::= <ExprIn>
```

In this grammar, the syntactic category :

- $<$ f $>$ represents declared field names
- $<$ b $>$ represents the output element being created during the execution of the predicate pred[a:A,b:B] containing the rule
- $<$ ExprIn $>$ is an expression that can be evaluated over the input instance
- $<$ out $>$ is a signature not belonging to the input model
- $<$ mapName $>$ represents the name of maps composing the backbone of the transformations
- $<$ x $>$ is a loop control variable name. (as Loop control variable are replaced at every step by the value they take, we consider them as part of ExprIn in $<$ condition $>$ and $<$ rule $>$)

4.3 Efficient Interpretation

As aforementioned, every rule is expressed inside a value predicate. Each value predicate is associated to a given $A \rightarrow B$ mapping, A and B being part of the transformation input and output respectively. The interpretation of a value predicate consists of creating a B element for each A element concerned by the mapping [1], and of interpreting each rule declared in the predicate. The purpose of those rules is to clearly define how to integrate those newly created B elements in the resulting transformation instance.

Recall that for a given input instance x of a module m_1 rules are processed in order to build up an instance y of a functional module m. Processing a rule will result in tuples being added to a field relation of y. Consider as an example the strict assignment rule

$$b.field = expr$$

where b denotes a B-element. Assuming that expression $expr$ is of type C then $field$ can be viewed as a relation from B to C. Processing this rule will evaluate $expr$ - to a value c, say - and add the tuple (b, c) to this relation. Processing a loose assignment rule

$$b\ in\ expr.field$$

will result in the tuple (c, b) being added to the field relation where c denotes the value of $expr$.

Processing loop rules amounts to performing the above processing for all possible values of the quantified variables.

A note about the computational complexity of processing rules: in general the complexity will be polynomial in the size of the instance x of m_1, which is generally much smaller than the size of the domain of all possible instances.

4.4 Module Interpretation

Now that the specific structure of functional modules has been introduced, we are able to describe in more detail the *Interpret* function introduced in section 3. The pseudo code corresponding to that function is given in fig. 10. Note that the function *process* invoked on line 11 was detailed in the previous subsection.

[1] The A elements which previously satisfied the guard related to that map

```
all i:Int |( i>=0 && i <#inp.s ) implies
      out.s[add[i,1]] = getNext[ out.s[i] ,inp.s[i]]
```

Fig. 9. Example of loop rule

```
Interpret(m,x) {// return an m-instance, given an m1-instance
    solution = empty m1-instance // the m-instance to return
    solution.add(x) // the solution contains x
    for each map in backbone_mappings do {// let map be A -> B map
      for each A-element a in x do{
        if guard_map(a) holds in x then {
          b=new B //create a new B-element
          solution.add(b) //  add this new element in the solution
          solution.add(map,a->b)  //  add the a->b link to the map relation
          for each rule in value_map(A,B){
            process rule; // add tuples to field relations of solution
          }
        }
      }
    }
    return solution;
}
```

Fig. 10. Pseudo code of a functional module's interpretation

Regarding computational complexity the same observation we already made for the processing of rules (see previous subsection) holds for the entire *Interpret* function: the complexity is polynomial in the size of the input instance x. Since the size of this input instance is in general much smaller than the size of the domain of all instances, we would expect a solution based on interpretation to be much faster than a SAT-solving based approach, at least for non-trivial domains. This will be confirmed by our experiments in the next section.

5 Evaluation

In this section we evaluate our approach by analysing to what extent it addresses the previously mentioned Alloy limitations (time complexity and optimal scope calculation).

5.1 The choice of case-studies

In order to illustrate the efficiency of interpretation over analysis, we choose to measure the analysis and interpretation performances for functional module FSM2Viz from FSM to VLM since this module is sufficiently complex to illustrate potential performance gains. This transformation defines for a given FSM instance a visualization expressed as a visual language model instance.

Here is an excerpt of the mappings defined in this module:

```
one sig Bridge{
   map1: State one -> lone ELLIPSE,
   map2: End one -> one DOUBLE_ELLIPSE,
   map3: Transition lone -> one CONNECTOR,
   map4: Start lone -> one CONNECTOR,
   map5: State one -> one TEXT
}
```

5.2 Time Complexity

We evaluate in this subsection the effectiveness of interpretation over analysis when it comes to generate instances of a transformation declared in a functional module. To do so we have measured the time required to obtain an instance of the FSM2Viz module following the two approaches for a various number of states. The analysis of the FSM2Viz module uses optimal scopes (discussed in sec. 5.3) for the different signatures to minimize execution times. The interpretation follows the pseudo-code given in fig.4. The measurements gathered can be found in table 1.

number of states in input	Analysis	Interpretation		
	FSM2Viz analysis (ms)	FSM analysis (ms)	FSM2Viz interpretation (ms)	Total Time (ms)
3	1 277	5	25	30
10	18 073	945	42	987
20	1 110 676	2756	73	2829

Table 1. Time performance comparison table (times in ms)

As we can observe, the time complexity of the interpretation of a functional module can be reduced to the time complexity of the analysis of its input model, the interpretation being quasi instantaneous. Those measurements are thus evidence that time complexity for complex transformation analysis can be greatly improved using interpretation of functional modules.

5.3 Scope calculation

The analysis of Alloy modules requires the definition of scopes for the signatures in order to limit the domain of possible instances. The time complexity of the analysis is a function of this domain's size. There is thus a need, in order for the analysis to take as little time as possible to complete, to define optimal or at least small scopes. In our analysis of FSM2Viz, we fixed the number of State elements for each run, and derived from the mappings the expected number of visual elements needed to represent the FSM. We ended up applying the formulas defined in fig.11 for each State's scope used in our measurements.

```
#Transition= #(Symbol*State)
#ELLIPSE = #(State-End)
#DOUBLE_ELLIPSE = #End
#CONNECTOR= #Transition +1
#TEXT = #State
#VisualElement=#(ELLIPSE+DOUBLE_ELLIPSE+CONNECTOR+TEXT)
```

Fig. 11. Formulas used to calculate the scopes of each element types

Calculating scopes in such a precise manner improves drastically the time complexity of analysis. As an example, the command given in fig.12 takes over 90 second to complete in the FSM2Viz module while the command given in fig.13 takes only 1 second to complete. The time complexity is thus highly dependent on the scope defined. The automatic generation of an optimal scope is an undecidable problem, and as such it requires to be approximated manually.

Scopes have no meaning in our interpretation approach as we do not perform instance finding in a finite domain. The bothersome task of assigning an optimal scope to each element can thus be dismissed in the case of functional module.

```
run for 17 but exactly 3 states
```

Fig. 12. non-optimal scope to generate all possible instances of FSM2Viz containing exactly 3 States

```
(run for 5 but exactly 3 State, exactly 2 ELLIPSE,exactly 1 DOUBLE ELLIPSE,
exactly 1 INVISIBLE CONTAINER, exactly 10 CONNECTOR, exactly 3 TEXT,
exactly 17 VisualElement, exactly 9 Transition)
```

Fig. 13. optimal scope to generate all possible instances of FSM2Viz containing exactly 3 States

6 Related Work

We are not aware of another work that attempts to circumvent the inherent limitations of the analysis via SAT solving done by Alloy. We now discuss some related work that considers the use of Alloy in the context of (model) transformations.

Anastasakis et al. [2] use Alloy to analyze the correctness of model transformations. They resort to their tool UML2Alloy [1] to transform the source and target metamodels into Alloy and translate the transformation rules into mapping relations and predicates at the Alloy level. The goal of their work is to check that the target instances are conforming to the target metamodel of the transformation. This is done by checking an Alloy assertion using the Alloy analyzer. In a similar line of work Baresi et al. [3] use Alloy to represent graph transformations represented in the AGG formalism. They use the Alloy analyzer to verify the correctness of the transformation by generating possible traces.

Perhaps more relevant for our work is the paper by Macedo et al. [6] which studies the use of Alloy to specify the execution semantics of QVT-Relational [7]. In this paper it is noted that there are some issues relating to the incomplete and ambiguous semantics of QVT-R given in the specification [7]. They propose a semantics of QVT-R based on Alloy; they do this by describing a translation from QVT-R specifications to Alloy. Besides contributing a semantics this approach allows to use Alloy as an execution platform: to compute the transformation from a source model to a target model, one defines an Alloy model whose sole instance is the desired source model and then one uses the Alloy analyzer to compute the target model. This approach is of course rather inefficient because of the inherent inefficiencies of analysis based on SAT solving. It would be interesting to investigate whether the transformation can be adapted so that the resulting Alloy modules are functional, thus permitting an efficient execution of QVT-R transformations.

7 Conclusion and Future Work

In this paper we have introduced the notion of functional module which corresponds to an Alloy module representing a transformation. We have identified a sublanguage of Alloy which, when used to express functional modules, allows efficient interpretation of these modules. We have provided evidence that for complex transformations replacing SAT-based analysis by interpretation may result in a significant time reduction.

Adding the possibility to interpret Alloy code opens up new application areas for Alloy which need to be investigated. Thus it remains to be seen whether we could use Alloy as an execution engine for model transformations; this would be appealing especially in the case of QVT-Relations where there is a lack of proper execution platforms. It would be also worthwhile to investigate under which condition the notion of functional module can be extended to the case where the transformation is relational rather than functional (i.e., there may be more than one possible target model for a given source model) while maintaining efficient interpretation.

References

1. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy: A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007.
2. Kyriakos Anastasakis, Behzad Bordbar, and Jochen M Küster. Analysis of model transformations via alloy. In *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*, pages 47–56, 2007.
3. Luciano Baresi and Paola Spoletini. On the use of alloy to analyze graph transformation systems. In *Graph Transformations*, pages 306–320. Springer, 2006.
4. Loïc Gammaitoni and Pierre Kelsen. Domain-specific visualization of alloy instances. ABZ 2014, to appear.
5. Daniel Jackson. *Software abstractions*. MIT press Cambridge, 2012.
6. Nuno Macedo and Alcino Cunha. Implementing qvt-r bidirectional model transformations using alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.
7. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.