Exploiting attributed type graphs to generate metamodel instances using an SMT solver

Hao Wu, Rosemary Monahan and James F. Power
Computer Science Department,
National University of Ireland, Maynooth
Email:{haowu,rosemary,jpower}@cs.nuim.ie

***Abstract*—In this paper we present an approach to generating instances of metamodels using a Satisfiability Modulo Theories (SMT) solver as a back-end engine. Our goal is to automatically translate a metamodel and its invariants into SMT formulas which can be investigated for satisfiability by an external SMT solver, with each satisfying assignment for SMT formulas interpreted as an instance of the original metamodel. Our automated translation works by interpreting a metamodel as a bounded Attributed Type Graph with Inheritance (ATGI) and then deriving a finite universe of all bounded attribute graphs typed over this bounded ATGI. The graph acts as an intermediate representation which we then translate into SMT formulas. The full translation process, from metamodels to SMT formulas, and then from SMT instances back to metamodel instances, has been successfully automated in our tool, with the results showing the feasibility of this approach.**

I. Introduction

From software development to programming language engineering, the *metamodelling* approach to development is widely adopted by software designers. The metamodel is the core of the metamodelling approach, where a metamodel is a formalism for defining sets of models, typically conforming to some metamodelling architecture such as the Meta-Object Facility (MOF) [16]. The MOF is based on the Unified Modelling Language (UML), particularly the UML class diagram notation, and thus a metamodel represented in MOF can also be defined as a UML class diagram. The goal of the metamodelling approach is to provide software engineers with a high-level abstraction, enabling them to refine the design of their models in order to alleviate the complexity of the code level. In this sense, the correctness of a metamodel becomes extremely important. But this raises one problem: how can one make sure the metamodel is correct? One solution to this problem, examined in this paper, is to automatically generate metamodel instances, and use these generated instances to test or verify the properties of a metamodel.

An *instance* of a metamodel is a concrete model that conforms to its metamodel. Here, "conformance" means that the concrete model satisfies each relationship (such as generalisation and association) and any constraints (expressed, for example using the Object Constraint Language (OCL)) defined in a metamodel [17]. A practical example might be a metamodel for the Java programming language, where the classes are the elements of the abstract syntax, where the constraints define the static semantics, and where the instances of the metamodel are valid and compilable Java programs. Generating instances for such a Java programming language metamodel is equivalent to producing syntactically and semantically correct Java programs. This contrasts with traditional algorithms based on generating a set of testing sentences from a context free grammar (CFG), leading to a large number of syntactically correct but semantically error programs [18].

However, generating instances of a metamodel is difficult [3]. This is mainly because the relationships (generalisation and association) in a metamodel impose limitations on how the instances of each class relate to each other in a model. In addition, generating one instance that conform to a metamodel and also satisfy additional constraints (such as OCL invariants) at the same time is undecidable. Nonetheless, its solution is important when testing or reasoning about the correctness of a metamodel.

The first contribution of this paper is that we present a way of translating a metamodel and its invariants into SMT2 formulas based on a attributed graph notation [8]. The second contribution of this paper is that we automate the approach proposed in this paper and implement it in a tool which can generate valid metamodel instances.

II. Background and Related Work

In this section we review some of the background in attributed and typed graphs, as well as SMT solvers, and discuss existing research related to our work.

A. Metamodels and Graphs

Elements in a UML class diagram can be represented as a graph, where each class can be considered as a node, and relationships between classes are edges linking from one node to another. Since a metamodel can be defined using a UML class diagram, a metamodel can also be interpreted as a graph. We consider all metamodels used in this paper as being presented as UML class diagrams, and formally represented as graphs.

The structure of interest to our work is an *Attributed Type Graph with Inheritance*, and can be defined in terms of a basic graph in three stages (following [8]):

Attributed Graph: An attributed graph is a tuple $AG = (G, D)$, with $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (s_j, t_j)_{j\in\{G,NA,EA\}})$ and $D = (S_D, OP_D)$ where

1) V_G and V_D denote sets of a *graph* and *data* nodes (vertices).

2) E_G, E_{NA} and E_{EA} denote a set of *graph attributes*, *node attributes* and *edge attributes*
3) $s_G : E_G \to V_G$, $t_G : E_G \to V_G$ are *source* and *target* functions mapping graph edges to nodes.
4) $s_{NA} : E_{NA} \to V_G$, $t_{NA} : E_{NA} \to V_D$ are source and target functions for the node-attribute edges.
5) $s_{EA} : E_{EA} \to E_G$, $t_{EA} : E_{EA} \to V_D$ are source and target functions for the edge-attribute edges.
6) S_D denotes a set of *attribute value sorts*.
7) OP_D denotes a set of *operations* over S_D.
8) D denotes a *data signature algebra* such that $\dot{\cup}_{s \in S_D} D_s = V_D$.

Attributed Type Graph: An attributed type graph is an *attributed graph* $ATG = (TG, Z)$, where Z is the final data signature algebra. A typed attributed graph (AG, t) has an attributed graph AG with a morphism $t : AG \to ATG$.

Attributed Type Graph with Inheritance: An attributed type graph with inheritance is a triple $ATGI = (ATG, I, A)$ where

1) ATG is an attributed type graph.
2) $I = (V_I, E_I, s, t)$ is an inheritance graph (where $V_I = TG_{V_G}$).
3) A denotes a set of *abstract nodes*, and $A \subseteq V_I$.

For any node $x \in V_I$, we define $clan_I(x) = \{y \in V_I | \exists\ path\ y \xrightarrow{*} x\ in\ I\} \subseteq V_I$.

Figure 1(a) shows an example of a metamodel represented as an $ATGI$ in an explicit notation. Each type node (nodes in TG_{V_G} are called type nodes) is depicted as a rectangle with a name to indicate a particular type. The italic font is used to indicate an abstract node, and an edge such as $worksIn$ describes a relation between two type nodes. The nodes for data types (nodes in TG_{V_D} are called data type nodes) are depicted as rectangles with round corners, with a dashed line connecting a type node and a data type node. Inheritance is depicted as in a UML class diagram, i.e. a solid line with a hollow arrow connecting two type nodes. The sets of nodes in the ATGI of Figure 1(a) can be enumerated as follows:

$$\begin{aligned} TG_{V_G} &= \{Person, Worker, Department\}. \\ TG_{V_D} &= \{Integer, Gender\}. \\ TG_{E_G} &= \{(Person, worksIn, Department), \\ &\quad (Worker, worksIn, Department)\}. \\ TG_{E_{NA}} &= \{(Person, gender, Gender), \\ &\quad (Person, age, Integer), \\ &\quad (Department, code, Integer). \\ TG_{E_{EA}} &= \emptyset. \\ A &= \{Person\}. \end{aligned}$$

In addition, $clan(Person) = \{Person, Worker\}$, $clan(Worker) = \{Worker\}$, and $clan(Department) = \{Department\}$.

A metamodel, represented as an $ATGI$, can also be represented in compact notation as shown in Figure 1(b).

B. Using a SMT2 Solver

We use a Satisfiability Modulo Theories (SMT) solver as our back-end engine since the SMT solver's rich background theories allow us to represent our logical formula that encode metamodels more naturally than a plain Satisfiability (SAT) solver [7]. For example, OCL constraints in a metamodel which use integer arithmetic can be easily represented as a formula using the *Integer* background theory. We encode our formulas using the SMT-LIB 2.0 standard (SMT2) which is well-supported by many fast SMT solvers [2].

C. Related Work

The *Alloy* tool operates as a model finder, translating a problem to a format suitable for a SAT solver [14]. The current version of Alloy relies on its engine *kodkod* which provides an efficient translation scheme for large problems [21]. However, it still performs poorly when handling integer arithmetic operations and this is mainly due to bit-blasting [22]. For example, an integer value over 1000 requires a bit width of 11 in Alloy which significantly slows down translation time. In contrast, SMT solvers are particularly good at dealing with relatively large numbers, since the integer arithmetic operations can be easily translated to corresponding SMT2 formulas.

Another disadvantage of Alloy is that it uses its own language, different from the metamodelling standards, necessitating an extra translation phase. Unfortunately, such a translation is not always straightforward, for example a one-to-one binary association in a metamodel requires a series of additional Alloy facts to constrain it appropriately. Though some of the research has already been conducted in this area, such a mapping can only increase the complexity of the instance generation process [1].

Similar to Alloy, the *Formula* tool is also designed to solve general constraint problems, but Formula uses a SMT solver as its back-end engine [15]. Thus, it has an advantage over Alloy's approach, which is a pure SAT-based approach when dealing with integer arithmetic. However, Formula is based on algebraic data types and constraint logic programming, thus any metamodels in MOF or UML class diagrams need to be translated into the Formula language. Since no automated tools support such translation, mapping a huge metamodel with OCL invariants without human interaction is almost infeasible. In addition, visualising any instances found from Formula would require an extra mapping back from the instance found by Formula to an instance of a metamodel. Thus, using Formula to find instances for a metamodel requires a lot of manual work.

Graph grammars offer a natural way to describe the derivation process and so have an advantage for generating metamodel instances [9], [12]. However, parsing a graph is expensive in terms of algorithmic complexity because a graph matching is not always deterministic, as a rule may match several sub graphs. Furthermore, OCL constraints must be manually transformed into graph constraints which can cause a problem when one has to deal with a large number of OCL constraints, even with very basic ones [23].

Cabot at el. propose a procedure that can transform a UML class diagram with OCL constraints into a Constraint Satisfaction Problem (CSP) according to a set of rules [6], [5], [11]. However, they do not provide translation rules for translating unidirectional associations, which we have found to be the most common kind of association used in metamodels. Although their translation process is automated, their approach

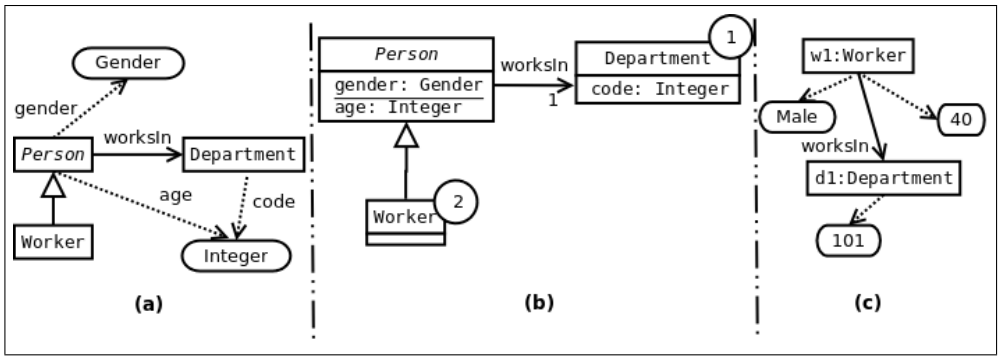


Fig. 1: Examples of (a) an ATGI in explicit notation. (b) a Bounded ATGI in compact notation. (c) an *instance* of a Bounded ATGI in explicit notation

cannot be used to enumerate instances and only supports much smaller metamodels than ours.

Soeken et al. encode a UML class diagram in a set of operations on bit-vectors which can be solved by SMT solvers using bit-vector theory [19]. A successful assignment for each bit-vector can be interpreted as an instance of the UML class diagram. Soeken et al. also propose an approach to encode a subset of OCL constraints as bit-vectors [20], and provide a list of corresponding mappings between OCL collection data types (Set, Bag, Sequence) and bit-vector operations. However, their approach does not provide an encoding for different multiplicities defined for a unidirectional and bidirectional association, especially those that are most commonly used. Furthermore, their approach does not allow a solution enumeration, and has not been fully automated. Thus, with a large metamodel and many OCL invariants, their approach involves manually translating the metamodel and OCL constraints into bit-vectors.

III. Representing a Metamodel as a Graph

In our approach, we bound the search space to find all valid metamodel instances within that bound. Thus, we introduce the definition of a bounded attributed type graph with inheritance ($ATGI_b$). Based on this, our approach forms a finite universe of all bounded attribute graphs AG_u typed over a given $ATGI_b$, and this AG_u represents the superset of the instances will be found.

Bounded Attributed Type Graph with Inheritance:
A bounded $ATGI$ is a tuple $ATGI_b = (TG, Z, I, A, b, mult_s, mult_t)$, where

- (TG, Z) are the elements of an ATG, and I, A are the elements of an ATGI, as above.
- $mult_s$, $mult_t : TG_{E_G} \rightarrow \mathbb{Z}^+ \bigcup \{*\}$.
  $mult_s$ and $mult_t$ are two functions which define the multiplicities at an edge's source and target nodes.
- b is a function, $b : TG_{V_G} \rightarrow \mathbb{Z}^+$, defining a bound for each type node in TG with a constraint $\{\exists n \; in \; TG_{V_G} | b(n) > 0\}$.

Since each ATGI contains a bounding function b, each instance of $ATGI_b$ is also bounded and finite. Thus, a bounded attributed graph typed over $ATGI_b$ can be represented as $(AG_b, type_b)$ with a morphism $type_b : AG_b \rightarrow ATGI_b$, where $type_b = (type_{b,V_G}, type_{b,V_D}, type_{b,E_G}, type_{b,E_{NA}}, type_{b,E_{EA}}, type_{b,D})$.

Each particular type node n in TG_{V_G} is assigned a bound by b, and this bound indicates the maximum number of instances of an $exact$ type node n that may appear in each metamodel instance. A bound n_b for a type node n can be assigned manually by a user or automatically calculated according to the different multiplicities in the metamodel. For an abstract node, no explicit bound is assigned, but this can be calculated by summing the bounds of its concrete descendants. Thus, the bound n_b of a general or an abstract type of node n can be calculated $n_b = \sum b(clan(n))$. Ideally, n_b will be greater than zero, or else we consider that type node n cannot be instantiated through its descendants.

With respect to the bound defined in $ATGI_b$, we can form a finite universe of all bounded attributed graphs typed over $ATGI_b$. Each instance of $ATGI_b$ can be derived from this universe. To simplify the following definitions, an edge e is represented by a pair $e = (a, b)$, where $a = s_j(e)$, $b = t_j(e)$, and $j \in \{G, NA, EA\}$.

Finite Universe: The finite universe of all bounded attributed graphs typed over $ATGI_b$ can be defined as a pair $(AG_u, type_u)$, where

1) AG_u is the finite universe of all bounded attributed graphs typed over $ATGI_b$,
2) $type_u : AG_u \rightarrow ATGI_b$, with
   $type_u = (type_{u,V_G}, type_{u,V_D}, type_{u,E_G}, type_{u,E_{NA}}, type_{u,E_{EA}}, type_{u,D})$, and where:
   - $V_G = \{g_1, g_2, ..., g_n\}$, and $\{type_{u,V_G}(g_i)\} \subseteq TG_{V_G}$, for $1 \leq i \leq n$.
   - $V_D = \{d_1, d_2, ..., d_m\}$, and $\{type_{u,V_D}(d_i)\} \subseteq TG_{V_D}$, for $1 \leq i \leq m$.
   - $E_{NA} = \{e_1, ..., e_j\} \subseteq V_G \times V_D$, and $\{type_{u,V_G}(s_{NA}(e_i))\} \subseteq TG_{V_G}$, and $\{type_{u,V_D}(t_{NA}(e_i))\} \subseteq TG_{V_D}$, for $1 \leq i \leq j$.
   - $E_G = \{e_{j+1}, ..., e_k\} \subseteq V_G \times V_G$, and $\{type_u(s_G(e_i))\} \subseteq TG_{V_G}$, and $\{type_u(t_G(e_i))\} \subseteq TG_{V_G}$, for $j+1 \leq i \leq k$.
   - $E_{EA} = \{e_{k+1}, ..., e_p\} \subseteq E_G \times V_D$, and $\{type_u(s_{EA}(e_i))\} \subseteq TG_{E_G}$, and $\{type_u(t_{EA}(e_i))\} \subseteq TG_{V_D}$, for $k+1 \leq i \leq p$.

V_G contains all type nodes in TG_{V_G} to be instantiated with respect to the bound defined for each type node in TG_{V_G}. V_D is defined similarly to V_G except that the type of each node in V_D is a type node in TG_{V_D}. Consequently, E_{NA} defines a containing relation over V_G and V_D by joining two nodes from different sets into an edge e. Similarly, E_G defines the set of all possible links between two type nodes in TG_{V_G}.

For example, Figure 1(b) shows an $ATGI_b$, where the bound for each type node is depicted as a circled number in the top-right corner of each type node. Here, there is a bound of 2 for type node $Worker$, a bound of 1 for type node $Department$ and no bound for abstract node $Person$. The multiplicity function $mult_t(worksIn) = \{1\}$ and $mult_s(worksIn)$ are unused.

With respect to this $ATGI_b$, the finite universe of all bounded attributed graphs typed over the $ATGI_b$ depicted in Figure 1(b) can be defined as follows:

1) $V_G = \{w1, w2, d1\}$.
2) $V_D = \{age1, gender1, age2, gender2, code1\}$.
3) $E_{NA} = \{e_1 = (w1, age1), e_2 = (w1, gender1), e_3 = (w2, age2),\ e_4 = (w2, gender2), e_5 = (d1, code1)\}$.
4) $E_G = \{e_6 = (w1, d1), e_7 = (w2, d1)\}$.
5) $type_{u,V_G}(w1) = type_{u,V_G}(w2) = Worker, type_{u,V_G}(d1) = Department$,
6) $type_{u,V_D}(age1) = type_{u,V_D}(age2) = type_{u,V_D}(code1) = Integer$,
7) $type_{u,V_D}(gender1) = type_{u,V_D}(gender2) = Gender$.
8) $b(type_{u,V_G}(w1)) = b(type_{u,V_G}(w2)) = 2, b(type_{u,V_G}(d1)) = 1$.

Figure 1(c) shows a sample instance derived from the universe of attributed graphs by selecting $\{w1\}$ from V_G, $\{age1, gender1, code1\}$ from V_D, $\{e_1, e_2, e_5\}$ from E_{NA} and $\{e_6\}$ from E_G, along with three assignments, of 40 and 101 to $age1$ and $code1$, and $gender1$ to the literal $Male$.

IV. Translating a Metamodel to SMT2 Formulas

Since, in our approach, a metamodel is represented as an $ATGI_b$, generating instances for a metamodel becomes the process of instantiating an $ATGI_b$. Thus, the idea of generating metamodel instances using a SMT solver is that we translate nodes and edges defined in the finite universe of all bounded attributed graphs typed over $ATGI_b$ into SMT2 quantifier freed formulas, and let the SMT solver assign appropriate values for those nodes and edges. Then:

- Each *successful* assignment by the SMT solver is interpreted as a bounded attributed graph typed over $ATGI_b$ (an instance of a metamodel).
- A *unsuccessful* assignment indicates that no graphs (instances) can be found in the current bounds for each type node in $ATGI_b$, and a user is confident enough to conclude that the metamodel is inconsistent in the current bounds. It is still, of course, possible to find an instance within larger bounds [13].

Figure 2 gives a summary of the translation rules for graph nodes, data nodes, node attributes and graph edges defined in AG_u. We use $\longrightarrow$ to denote a translation. In this figure, rules 1-4 show the translation rules for translating nodes and edges, while rules 5-7 show additional constraints for nodes and edges. In Figure 2,

1) $\{n_i \in V_G : 1 \leq i \leq |V_G|\} \longrightarrow \{(\mathbf{declare\text{–}fun}\ n_i\ ()\ \mathbf{Bool})\}$,
2) $\{d_i \in V_D : 1 \leq i \leq |V_D|\} \longrightarrow \{(\mathbf{declare\text{–}fun}\ d_i\ ()\ \mathbf{T}) : \mathbf{T} \in \{\mathbf{Bool}, \mathbf{Int}\}\}$,
but when $(type_u(d_i) = \{Enum\}) \longrightarrow \{(\mathbf{declare\text{–}fun}\ d_i\ ()\ \mathbf{Int})\}$
add $\{(\mathbf{assert}\ (\mathbf{and}\ (>= d_i\ 0)\ (<= d_i\ |Enum| - 1)))\}$
Here $|Enum|$ denotes the number of literals in an enumeration type.
3) $\{e_i \in E_{NA} : 1 \leq i \leq |E_{NA}|\} \longrightarrow \{(\mathbf{declare\text{–}fun}\ e_i\ ()\ \mathbf{Bool})\}$
4) $\{e_i \in E_G : 1 \leq i \leq |E_G|\} \longrightarrow \{(\mathbf{declare\text{–}fun}\ e_i\ ()\ \mathbf{Bool})\}$
5) $\{(\mathbf{assert}\ (=>\ e_i\ (\mathbf{and}\ s_{NA}(e_i)\ t_{NA}(e_i)))) : 1 \leq i \leq |E_{NA}|\}$
6) $\{(\mathbf{assert}\ (=>\ e_i\ (\mathbf{and}\ s_G(e_i)\ t_G(e_i)))) : 1 \leq i \leq |E_G|\}$
7) $\{(\mathbf{assert}\ (=>\ (\mathbf{not}\ s_{NA}(e_i))\ (= t_{NA}(e_i)\ \mathbf{v}))) : 1 \leq i \leq |E_{NA}|\}$
where if $type_u(t_{NA}(e_i)) = \{Integer\}$, then $\mathbf{v} = (-1)$.
and if $type_u(t_{NA}(e_i)) = \{Bool\}$, then $\mathbf{v} = false$.
and if $type_u(t_{NA}(e_i)) = \{Enum\}$, then $\mathbf{v} = 0$.

Fig. 2: Translation rules for V_G, V_D, E_{NA} and E_G defined in AG_u.

- Every node in V_G is translated into a Boolean constant in SMT2, representing whether or not it will be present in the instance.
- The nodes in V_D representing basic data types are translated into different types of constants according to their types. In the current approach, we support three different basic data types which are Boolean, integer and enumeration type. A node in V_D which has an integer type is directly mapped to an Int constant in SMT2. Similarly, a Boolean type node is mapped to a $Bool$ constant. We translate a node whose type is an enumeration type to an Int constant in SMT2 with an extra constraint. This constraint limits the domain of an Int constant to between zero and the number of literals defined in an enumeration data type minus one.
- Each edge in E_G and E_{EA} is mapped to a Boolean constant (rules 3 and 4). An additional constraint for each edge e in E_{NA} is imposed to indicate that when e is selected, both nodes $s_{NA}(e)$ and $t_{NA}(e)$ encoded by e are also forced to be selected (rule 5). Similarly, an additional constraint is also defined for each edge in E_G (rule 6).
- For each data node d in V_D, we also need an extra constraint specifying that when a graph node n in V_G is not selected, none of its data attributes need to be selected. We restrict the assignment for d to a single $fixed$ value v so that each time the graph node n is not selected, the corresponding d's value is always a fixed default value. This constraint is encoded in rule 7 in Figure 2, with the $fixed$ value for d determined by its type.

A. Translating Associations

A metamodel can have inheritance and association relationships between entities, and these two relationships can be considered as additional constraints for a metamodel, encoded as extra SMT2 formula for the nodes and edges contained in V_G, V_D, E_{NA}, and E_G.

For a generalisation relationship between two nodes $parent \in V_G$ and $child \in V_G$ with $clan(type_u(child)) \subset clan(type_u(parent))$, we simply add edges that encode all data nodes contained by $parent$ into E_{child}. We use E_{child} to denote the set of edges that encode all data nodes contained by $type_u(child)$, where $E_{child} \subseteq E_{NA}$. For a $child$ that has two or more parents, we combine data nodes contained by all parents into E_{child}.

Associations in a metamodel can be categorised into two kinds: unidirectional and bidirectional. In a metamodel, these two kinds of associations can be decorated with different multiplicities that impose a constraint on how two nodes can be linked in a metamodel instance. To encode constraints for different multiplicities defined on an association, we extract a subset of edges E_r from E_G that encode all the links between two different type nodes, group them into a 2D-array, and then use logical connectives to specify the different multiplicities.

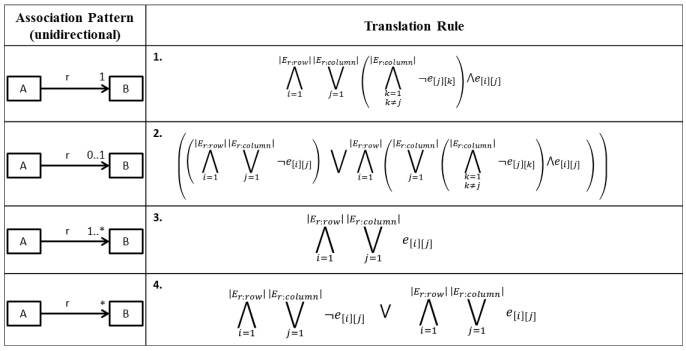
In this 2D-array:

- The rows and columns of E_r are denoted as $E_{r:row}$ and $E_{r:column}$ respectively.
- $E_{r:row}$ represents all links from one instance of A to one instance of B.
- $E_{r:column}$ represents all links from one instance of B to one instance of A.
- The edge at ith row and jth column of E_r is represented as $e_{[i][j]}$.

Figure 3 summarises the translation rules for translating unidirectional associations. Here, we only consider $E_{r:row}$. For example, for an association r with the multiplicity $1..*$ defined at the one end, we apply the SMT2 *or* function through each row of the array (rule 3 in Figure 3). This forces the SMT solver to select at least one edge from each row. Rule 1 captures that each instance of A can have $exactly$ one reference to an instance of B. That is, when an edge at the ith row and jth column in the 2D-array is selected, the rest of the edges in the ith row cannot be selected. Rule 2 is formed by the disjunction of rule 1 and the negation of rule 3. It captures that each instance of A can either have $zero$ references or $exactly$ one reference to an instance of B. Finally, rule 4 states that either $zero$ edges or $at\ least$ one edge is selected from each row in the 2D-array.

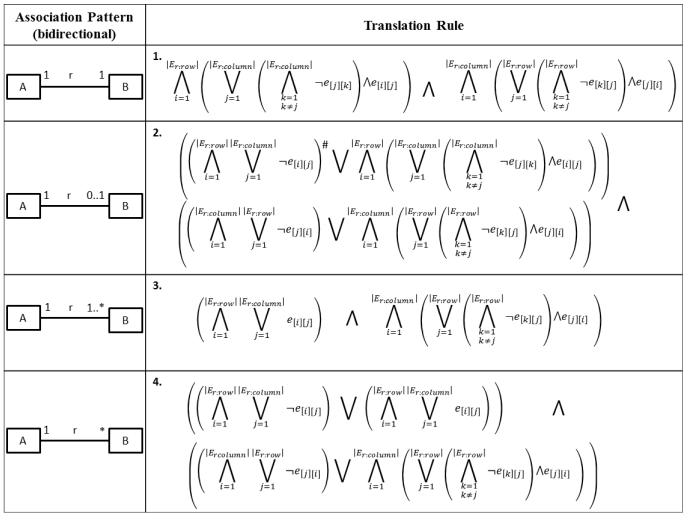
Figure 4 summarises the translation rules for the most commonly used bidirectional associations. The rules for bidirectional associations are similar to unidirectional association. However, for a bidirectional association r, we apply our translation rules in two dimensions ($E_{r:row}$ and $E_{r:column}$) according to different multiplicities defined for both ends of the association r. After applying the translation rules, each dimension results in one subformula, which we join via conjunction.

Since a bidirectional link is symmetric we interpret each edge $e = (a, b)$ as two links, i.e. a link from a to b, and



E_r: the set of edges are associated with a particular association r, $E_r \subseteq E_G$.

Fig. 3: Translation rules for unidirectional association



E_r: the set of edges are associated with a particular association r, $E_r \subseteq E_G$.

Fig. 4: Translation rules for bidirectional association

a link that goes back from b to a. Rule 1 describes one-to-one bidirectional associations. It is similar to rule 1 for unidirectional associations except that the translation rule is also applied through $E_{r:column}$, thus forcing the SMT solver to select exactly one edge from two dimensions.

In Figure 4, rules 1 and 2 are similar, except that the sub formula marked with # in rule 2 ensures that no edges from a row are selected, and since all the links are bidirectional, the same sub formula is applied through $E_{r:column}$. Rule 3 conjoins a subformula from rule 1 with a formula that allows at least one edge from each column to be selected. Similarly, rule 4 is a conjunction that ensures that no instances of A are associated with any instances of B and that, due to the relationships symmetry, no instances of B are associated with any instances of A.

B. Translating OCL Invariants

Apart from the constraints defined on multiplicities for associations, additional invariants expressed in OCL can also be defined over a metamodel. Since we represent everything in a metamodel as an $ATGI_b$, OCL invariants defined on a metamodel can be considered as additional formulas over the

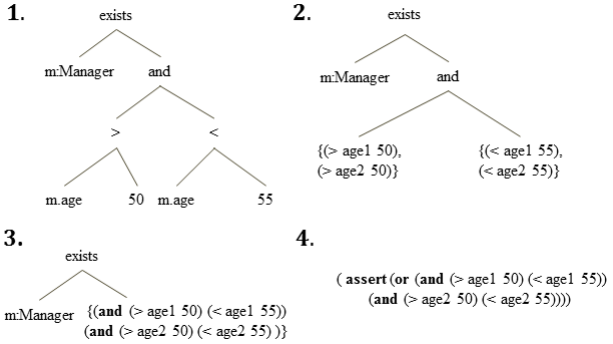


Fig. 5: An example of translating an OCL constraint. Here, the OCL constraint is:
`Manager.allInstances()->exists(m|m.age>50 and m.age<55)`

nodes and edges of the $ATGI_b$. To deal with the additional OCL invariants, we parse OCL into an abstract syntax tree (AST), traverse the AST nodes to extract relevant nodes and edges from AG_u, and translate them into SMT2 formula. We conjoin the formulas with those produced for the metamodel and input the result into the SMT solver to find an assignment.

The OCL invariants that we can handle by our translation can be summarised as follows:

1) OCL integer and logical expression can be directly mapped to corresponding SMT2 functions by using the following function $F : expr \rightarrow SMT2Formula$.
2) Since we represent a metamodel as a bounded attributed type graph, quantifiers over an object type is a set of graph nodes (V_G) that are bounded by it's type node (TG_{V_G}). The following formulas show the translation from a quantified OCL expression to SMT2 formulas.

   a) Obj.allInstances()→exists(expr) $\longrightarrow \bigvee_{i=1}^{b(Obj)} F(expr_i)$

   b) Obj.allInstances()→forAll(expr) $\longrightarrow \bigwedge_{i=1}^{b(Obj)} F(expr_i)$

   c) Obj.allInstances()→one(expr) $\longrightarrow$
   $\bigvee_{i=1}^{b(Obj)} (\bigwedge_{\substack{j=1 \\ j\neq i}}^{b(Obj)} \neg F(expr_j) \wedge F(expr_i))$
3) Quantifiers can be used in a nested chain to indicate an operation over two different sets of instances. For this kind of nested quantifiers, we perform an extra translation step by calculating the cross product of $F(expr_i)$ and $F(expr_j)$. The following SMT2 formulas show the translation for nested quantifier OCL expression A.allInstances()→forAll(B.allInstances→exists(expr)).
   $\bigwedge_{i=1}^{b(A)} \bigvee_{j=1}^{b(B)} F(expr_i) \times F(expr_j)$
4) A navigation used in an OCL expression can be interpreted as a reference to a set of edges in our graph representation of a metamodel. The translation for a navigation r thus is performed by extracting relevant edges from E_G, and use the following translation rule to translate them into SMT2 formuals.
   $\bigwedge_{i=1}^{|E_r|} D(e_i) \Rightarrow F(expr_i)$, where $D : E_G \rightarrow SMT2Var$.

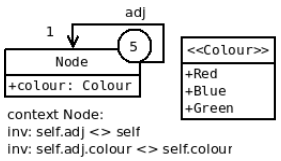


Fig. 6: Graph colouring metamodel

It is obvious that not every OCL invariant can be translated to SMT2 formulas by using the rules above, but these rules support most practical invariants. For example, Figure 5 shows a full translation of an OCL invariant step by step. The expression $Manager.allInstances()$ indicates that all graph nodes having type $Manager$ are extracted from V_G. The operators $>$,$<$ and and are mapped to corresponding SMT2 functions, and the quantifier $exists$ is translated to or over the relevant instances.

V. An Example

In this section, we use a metamodel of graph colouring as an example to demonstrate our approach. The metamodel itself can be seen in Figure 6. This metamodel is constrained with two OCL invariants. The first one specifies that no node can have itself as its neighbour, while the second one indicates that a node and its neighbour cannot share the same colour. From the metamodel to $ATGI_b$, and $ATGI_b$ to SMT2 formulas, the translation proceeds as follows:

1) The metamodel itself is first translated into an $ATGI_b$ with $I = \emptyset$ by defining $b(Node) = 5$ (depicted in the right top corner of $Node$), and a finite universe AG_u is formed from $ATGI_b$.
2) The typed node $Node$ is translated to a boolean function in SMT2 with defined bound of 5.
3) Data node $Colour$ is an enumeration type and thus gets translated into an integer function in SMT2 in the range $0-2$ inclusive (since we only have three colours here.).
4) Every edge in E_{adj} is also translated to a boolean function in SMT2.
5) For the first invariant, the translation process eliminates all the edges e such that $s_G(e) \neq t_G(e)$. Thus, the graphs left are those that do not contain cycles of length 1.
6) The translation for second invariant iterates the remaining edges such that each $s_G(e).colour \neq t_G(e).colour$. Therefore, this step makes sure each edge's source and target object selected by SMT solver will not share the same literal.
7) Finally, the formulas from steps 2-4 conjoined with formulas from step 5 and 6, and fed into the SMT solver. In this case it is Z3. One successful assignment found by Z3 is interpreted as an instance of the metamodel. Figure 7 shows one of the interpreted instances.

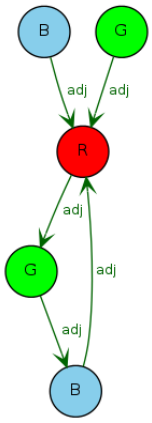


Fig. 7: One of the instances of graph colouring metamodel found by Z3. The letter in each node of this graph represents a colour (R:Red, G:Green, B:Blue), and an edge between two nodes means they are adjacent.

VI. Implementation and Results

We have implemented this approach in a tool called *A Small Metamodel Instance Generator (ASMIG)*. ASMIG is a fully automated tool, which:

1) reads in a metamodel in *Ecore* format [4],
2) represents it as a bounded attributed graph (AG_u) typed over $ATGI_b$,
3) translates AG_u and OCL invariants into SMT2 formulas,
4) invokes the Z3 SMT solver to find an assignment for SMT2 formulas,
5) interprets each successful assignment as a valid instance of metamodel.

Figure 8 depicts this process, which is fully automatic.

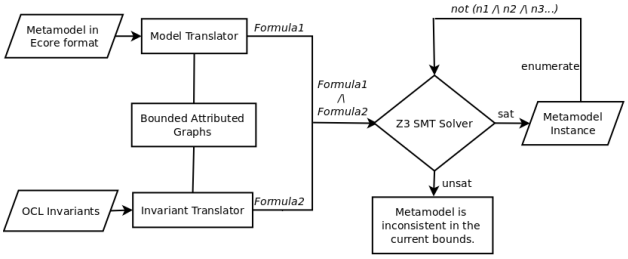


Fig. 8: The flow diagram for ASMIG. The inputs to the process are an Ecore metamodel and a set of OCL constraints, and the outputs are a set of valid instances of the metamodel.

To speed up translation, all nodes and edges are stored in red-black trees within a hash table. To prevent generating repeated solutions for each enumeration, any previous successful assignments for a formula are negated and added as an extra SMT2 formula. ASMIG also supports partial models, but at present the partial model needs to be defined internally via relevant APIs. To visualise each instance that is generated, ASMIG generates a GraphViz representation for each successful assignment and caches the generated formula to speed up each enumeration [10]. Furthermore, ASMIG can also generate instances that do not conform to the metamodel (negative test cases) by negating one or more of the SMT2 formula.

Metamodel	Number of Classes	Assocs	Attribs	Time in ms Translation	Avg Finding
Company [2]	7	6	6	476ms	38ms
C 1.0 [1]	34	4	0	388ms	54ms
C++ 1.0 [1]	16	4	5	372ms	26ms
Java [5]	233	104	1	666ms	441.7ms
Royal&Loyal [3]	15	41	2	403ms	36.1ms
Finite State Machine 1.0 [1]	6	7	0	368ms	26ms
Ecore [4]	22	40	0	439ms	42.8ms
UML2 Class Diagram [4]	40	26	46	442ms	41.6ms
Web App: Conceptual Model[1]	19	24	0	368ms	42.7ms
KM3 [1]	12	7	0	365ms	33.4ms
Business Process Model [1]	26	15	0	375ms	62.2ms
CPL1.0 [1]	32	16	0	384ms	94.9ms
DoDAF-SV5 [1]	31	54	1	391ms	99.8ms
GraphML [1]	11	13	2	392ms	37.8ms
Hierarchical State Machine 1.0 [1]	15	16	0	378ms	42.2ms
Maven(maven.xml) 0.3 [1]	58	32	0	403ms	74.3ms
MoDAF0.1 [1]	48	35	0	398ms	49.0ms
QualityofService [1]	24	26	0	376ms	51.9ms
DOT1.0 [1]	26	20	0	386ms	58.7ms
BibTexML1.2 [1]	28	4	0	379ms	39.8ms

TABLE I: Details of 20 metamodels; 100 instances of these metamodels (except for the Finite State Machine) were generated by the ASMIG tool.

Table I shows the results of evaluating our approach against 20 different metamodels. In this table, "Classes", "Assocs" and "Attribs" denote the number of (non-abstract) classes, associations and attributes translated for each metamodel. To evaluate the feasibility of our approach, we show the translation time for each metamodel, and also calculate the average time spent on finding an instance. This is based on the average time for the first 100 instances enumerated (except for the Finite State Machine 1.0 metamodel were there were only 16 instances enumerated in total with the bound of 1 for each type).

All instances for the metamodels are generated on a machine with a 2.8GHz Intel Core2Quad CPU and 4GB of RAM. In the current version of ASMIG we use Z3 as the back-end solving engine, so the average time spent on finding an instance depends on both the formulas we generated and the Z3 solving time. All these metamodel instances are generated by using ASMIG with a default bound of 1 for each $exact$ type in 19 metamodels. In order to examine the translation for OCL invariants, we choose bounds of 2 and 3 for classes in the metamodel.

It is difficult to compare our results with existing work, since existing approaches only evaluate their work on much simpler metamodels [9], [12], [6], [19]. We use different sized metamodels collected from different sources to evaluate the speed of translation, average instance finding time and also

practical applications. We hope that this set of metamodels can be considered as well-worked examples, so that others can use it as a benchmark for future comparisons.

The translation time is affected by two factors, the size of the metamodels and the type of associations in a metamodel. For example, the full Java metamodel referred to in Table I has 233 classes and 104 associations and the average instance finding time is much longer than that for the *Finite State Machine 1.0* which only has 6 classes and 4 associations. Regarding the type of associations used in a metamodel, a $one–to–one$ bidirectional association produces more formulas than a $one–to–many$ bidirectional association as it is more constrained.

Having OCL invariants defined on a metamodel also affects the translation time. For example, the $Company$ metamodel has 7 OCL invariants, which require an additional $30ms$ for their translation. We cannot present all the instances generated from ASMIG due to space limitations, but these instances are available from our website [6].

VII. Conclusion

In this paper we have described an approach that represents a metamodel as a Bounded Attributed Type Graph with Inheritance ($ATGI_b$), and translates a universe of all bounded attribute graph typed over $ATGI_b$ into SMT2 formulas. A fully automatic tool called ASMIG implementing this approach has been used to generate instances within a short time, even for relatively large metamodels. The advantage of our approach is that we closely bind attributed type graphs with an SMT solver to find instances for metamodels, and our tool $ASMIG$ demonstrates the feasibility of automation of this approach.

In future work we intend to extend our approach to provide more useful instances for software engineering, for example, finding all instances that can achieve given testing or coverage criteria.

References

[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, pages 436–450, Nashville,Tennessee, USA, 2007. Springer.

[2] Clark Barrett, Aaron Stump, Cesare Tinelli, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krsti, Michal Moskal, Leonardo De Moura, Roberto Sebastiani, To David Cok, and Jochen Hoenicke. The SMT-LIB Standard: Version 2.0. In *8th International Workshop on Satisfiability Modulo Theories*, Edinburgh, UK, 2010. Elsevier Science.

[3] Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(12):70–118, 2005.

[4] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[1] available at: http://www.emn.fr/z-info/atlanmod/index.php/Ecore

[2] simple example similar to Figure 1, available at our website

[3] from Eclipse Modeling Framework Royal and Loyal Example Project

[4] extracted from Eclipse Modeling Framework

[5] available at: http://www.jamopp.org/index.php/JaMoPP_Download

[6] http://www.cs.nuim.ie/∼haowu/ASMIG/Results/MM

[5] Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *22nd International Conference on Automated Software Engineering*, pages 547–548, Atlanta, Georgia, USA, 2007. IEEE Computer Society.

[6] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of UML/OCL class diagrams using constraint programming. In *IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 73–80, Berlin, Germany, 2008. IEEE Computer Society.

[7] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Budapest, Hungary, 2008. Springer.

[8] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer New York, Inc., Secaucus, NJ, USA, 2006.

[9] Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. Generating instance models from meta models. *Software and Systems Modeling*, 8(4):479–500, 2009.

[10] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - Open Source Graph Drawing Tools. *Graph Drawing*, pages 483–484, 2001.

[11] Carlos Alberto González Pérez, Fabian Buettner, Robert Clarisó, and Jordi Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Formal Methods in Software Engineering: Rigorous and Agile Approaches*, Zurich, Suisse, 2012.

[12] Berthold Hoffmann and Mark Minas. Generating instance graphs from class diagrams with adaptive star grammars. In *3rd International Workshop on Graph Computation Models*, 2011.

[13] D. Jackson and C.A. Damon. Elements of style: analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, 1996.

[14] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering Methodologies*, 11(2):256–290, 2002.

[15] Ethan Jackson, Tihamr Levendovszky, and Daniel Balasubramanian. Reasoning about metamodeling with formal specifications and automatic proofs. In *14 International Conference on Model Driven Engineering Languages and Systems*, pages 653–667, Wellington, New Zealand, 2011. Springer.

[16] Object Management Group. Meta Object Facility Core Specification v2.4.1, August 2011.

[17] Object Management Group. Object Constraint Language Version 2.3.1, January 2012.

[18] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[19] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla, and R. Drechsler. Verifying UML/OCL models using boolean satisfiability. In *Design, Automation Test in Europe Conference Exhibition*, pages 1341–1344, Dresden, Germany, 2010.

[20] Mathias Soeken, Robert Wille, and Rolf Drechsler. Encoding OCL data types for sat-based verification of UML/OCL models. In *5th International Conference on Tests and Proofs*, pages 152–170, Zurich, Switzerland, 2011. Springer.

[21] Emina Torlak and Daniel Jackson. Kodkod: a relational model finder. In *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, Braga, Portugal, 2007. Springer.

[22] Eminate Torlak. *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 2009.

[23] Jessica Winkelmann, Gabriele Taentzer, Karsten Ehrig, and Jochen M. Küster. Translation of restricted OCL constraints into graph constraints for generating meta model instances by graph grammars. *Electronic Notes in Theoretical Computer Science*, 211:159–170, 2008.