Towards Testing Model Transformation Chains Using Precondition Construction in Algebraic Graph Transformation

Elie Richa[1,2], Etienne Borde[1], Laurent Pautet[1]
Matteo Bordin[2], and José F. Ruiz[2]

[1] Institut Telecom; TELECOM ParisTech; LTCI - UMR 5141
46 Rue Barrault 75013 Paris, France
`firstname.lastname@telecom-paristech.fr`
[2] AdaCore, 46 Rue d'Amsterdam 75009 Paris, France
`lastname@adacore.com`

Abstract. Complex model-based tools such as code generators are typically designed as chains of model transformations taking as input a model of a software application and transforming it through several intermediate steps and representations. The complexity of intermediate models is such that testing is more conveniently done on the integrated chain, with test models expressed in the input language. To achieve a high test coverage, existing transformation analyses automatically generate constraints guiding the generation of test models. However, these so called *test objectives* are expressed on the complex intermediate models. We propose to back-propagate test objectives along the chain into constraints and test models in the input language, relying on precondition construction in the theory of Algebraic Graph Transformation. This paper focuses on a one-step back-propagation.

Keywords: testing, model transformation chains, algebraic graph transformation, weakest precondition, ATL

1 Introduction

Tools used in the production of critical software, such as avionics applications, must be thoroughly verified: an error in a tool may introduce an error in the critical software potentially putting equipment and human lives at risk. Testing is one of the popular methods for verifying that such tools behave as specified. When testing critical applications, a primary concern is to ensure high *coverage* of the software under test (*i.e.* ensure that all features and different behaviors of the software are tested). The recommended way to achieve this is to consider each component separately, identify its functionalities, and develop dedicated tests (*unit testing*). This guideline is therefore reflected in industrial software quality standards such as DO-330 [13] for tools in the avionics domain.

However, with complex model transformation tools such as code generators, applying unit testing is very costly and impractical. In fact, such tools are often

designed as a chain of several model transformations taking as input a model developed by the user in a high-level language and transforming it through several steps. Unit testing then boils down to testing each step of the chain independently. In practice, intermediate models increase in detail and complexity as transformations are applied making it difficult to produce test models in the intermediate representations: manual production is both error-prone because of the complexity of the languages and tedious since intermediate languages do not typically have model editors [1]. It is often easier for the tester to create a model in the input language of the chain, with elements that he knows will exercise a particular feature down the chain. Existing approaches [7,8,11] can automate the production of tests for model transformations, thus producing unit tests. However, when a test failure uncovers an error, analyzing the complex intermediate representations is difficult for the developer.

Given these factors, we propose an approach to the testing of model transformation chains that aims to ensure test coverage of each step while preserving the convenience of using test models in the input language. First we rely on existing analyses [7,8,11] to generate a set of so-called *test objectives* that must be satisfied by test models to ensure sufficient coverage. Then we propose to automatically propagate these test objectives *backward* along the chain into constraints over the input language. The back-propagation relies on the construction of preconditions in the theory of Algebraic Graph Transformation (AGT) [5].

Within this general approach, we focus in this paper on the translation of postconditions of one ATL transformation step to preconditions, which is a key operation in the propagation of test objectives. We thus propose a first translation of the ATL semantics into the AGT semantics where we use the theoretical construction of *weakest preconditions* [9] . We illustrate our proposal on a realistic code generation transformation using a prototype implementation based on the *Henshin*[3] and *AGG*[4] frameworks. This first prototype allowed us to back-propagate test objectives across one transformation step.

In the remainder of the paper, section 2 gives an overview of the testing approach, explaining the role of precondition construction. Section 3 recalls the main concepts of ATL and AGT. Section 4 introduces an example of ATL transformation that will serve to illustrate (i) the translation of ATL to AGT in section 5 and (ii) the construction of preconditions in section 6. Finally, we present our prototype in section 7 and conclude with our future plans in section 8.

2 General Approach

As highlighted in [1], one of the major challenges in achieving thorough testing is producing test models that are relevant, *i.e.* likely to trigger errors in the implementation. Several approaches address this challenge for standalone transformations. In [7], [8] and [11], the authors propose to consider a transformation

[3] The Henshin project, `http://www.eclipse.org/henshin`

[4] The Attributed Graph Grammar development environment, `http://user.cs.tu-berlin.de/~gragra/agg`

and analyse one or more of (i) the input metamodel, (ii) the transformation specification and (iii) the transformation implementation. This analysis results in a set of constraints, each describing a class of models that are relevant for finding errors in the transformation. We refer to such constraints as *test objectives* in the remainder of the paper. Constraint satisfaction technologies such as the Alloy Analyzer[5] and EMFtoCSP [6] are then used to produce model instances such that each test objective is satisfied by at least one test model.

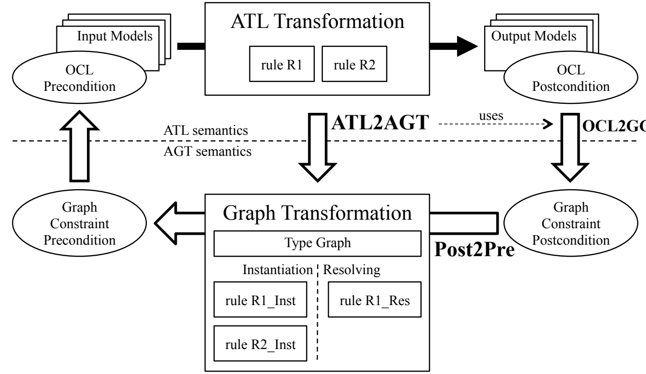


Fig. 1: Transformation of Postcondition to Precondition

Let us now consider a transformation chain $M_i \xrightarrow{T_i} M_{i+1}$ for $0 \leq i < N$ where an input model M_0 is processed by N successive transformation steps T_i into intermediate models M_i and ultimately into the final output model M_N. Focusing on an intermediate transformation T_i such that $i > 0$, we can apply the above approaches to obtain a set of test objectives $\{to_{i,j} \mid 0 \leq j\}$ ensuring the thoroughness of the testing of T_i. Each test objective $to_{i,j}$ is a constraint expressed over the input metamodel of T_i. At this point we want to produce a model M_0 at the beginning of the chain, which ultimately satisfies $to_{i,j}$ after being processed by the sequence $T_0 \ ; \cdots ; \ T_{i-1}$. We propose to automate this operation by transforming $to_{i,j}$ into a test objective $to_{i-1,j}$ at the input of T_{i-1} and thus iterate the process until we obtain $to_{0,j}$ that can serve to produce a model M_0. The key challenge of this paper is to devise an analysis that takes as input a *constraint* $to_{i,j}$ and a transformation specification T_{i-1}, and produces as output a *constraint* $to_{i-1,j}$. Such a method exists in the formal framework of *Algebraic Graph Transformation* (AGT) [5] in the context of the formal proof of correctness of graph programs. It is the transformation of postconditions into preconditions [9] that we propose to adapt and reuse in our context. Since we consider transformations specified in ATL [10], a translation to AGT is necessary.

As shown in Figure 1, we propose to translate the ATL transformation T_{i-1} into a graph transformation program (*ATL2AGT* arrow) and $to_{i,j}$ into a graph

[5] Alloy language and tool, `http://alloy.mit.edu/`

constraint (*OCL2GC* arrow). Assuming the constraint is a postcondition of T_{i-1}, we automatically compute the precondition $to_{i-1,j}$ that is sufficient to satisfy the postcondition (*Post2Pre* arrow) using the formal foundation of AGT. Since ATL embeds OCL constraints, *ATL2AGT* also uses *OCL2GC*. However this is a complex translation [14] that will not be addressed given the space limitations. We thus focus on a first proposal of *ATL2AGT* in section 5 and *Post2Pre* in section 6, both limited to the structural aspects of the semantics and constraints. First, we recall the main elements of ATL and AGT in the next section.

3 Semantics of ATL and AGT

3.1 ATL and OCL

ATL [10] is a model-to-model transformation language combining declarative and imperative approaches in a hybrid semantics. A transformation consists of a set of declarative *matched rules*, each specifying a source pattern and a target pattern. The source pattern is a set of objects of the input metamodel and an optional OCL[6] constraint acting as a guard. The target pattern is a set of objects of the output metamodel and a set of bindings that assign values to the attributes and references of the output objects. The execution of a transformation consists of two main phases. First, the matching phase searches in the input model for objects matching the source patterns of rules (*i.e.* satisfying their filtering guards). For each match of a rule's source pattern, the objects specified in the target pattern are instantiated. A tuple of source objects may only match one rule, otherwise an error is raised. For this reason the order of application of rules is irrelevant. Second, the target elements' initialization phase executes the bindings for each triggered rule. Bindings map scalar values to target attributes, target objects (instantiated by the same rule) to target references, or source objects to target references. In the latter case, a *resolve* operation is automatically performed to find the rule that matched the source objects, and the first output object created by that rule (in the first phase) is used for the assignment. If no or multiple resolve candidates are found, the execution stops with an error.

As the current proposal is limited to structural aspects, we only consider bindings of target references and not those of attributes. OCL constraints are not considered as *OCL2GC* (Figure 1) is too complex to address within this paper [14]. Instead, we will use test objectives in the form of AGT graph constraints.

3.2 AGT and Graph Constraints

Several graph transformation approaches are proposed in the theory of Algebraic Graph Transformation [5]. We will be using the approach of *Typed Attributed Graph Transformation with Inheritance* which we found suitable to our needs and which is supported in the AGG tool allowing for concrete experimentation of our proposals (see section 7). There are 3 main elements to a graph transformation:

[6] Object Constraint Language (OCL), `http://www.omg.org/spec/OCL`

a *type graph*, a set of *transformation rules*, and a *high-level program* specifying the order of execution of rules.

Graphs consist of *nodes* connected with directed *edges*. Much like models conform to metamodels, typed graphs conform to a *type graph*. As introduced in [3], *metaclasses*, *references* and *metaclass inheritance* in metamodels correspond to *node types*, *edge types*, and *node type inheritance* in type graphs which allows an easy translation between the two. Even though multiplicities and containment constraints are not addressed in type graphs, they are supported in AGG.

A graph transformation is defined as a set of *productions* or *rules* executed in a graph rewriting semantics. There are two major approaches to defining rules and their execution. Even though the theory we use is based on the Double Pushout (DPO) approach, we will use the simpler Single Pushout (SPO) approach and notation which is also the one implemented in AGG. A rule consists of a *morphism* from a *Left-Hand Side* (LHS) graph to a *Right-Hand Side* (RHS) graph. The LHS specifies a pattern to be matched in the transformed graph. Elements mapped by the morphism are preserved and elements of the RHS that are not mapped by the morphism are new elements added to the transformed graph. We do not address element deletion since our translation will not need it (see section 5). Thus the execution of a rule consists in finding a match of the LHS in the transformed graph and adding the new nodes and edges.

With the transformation rules defined above, we can construct so called *high-level programs* [9] consisting of the sequencing or the iteration of rules. A program can be (1) elementary, consisting of a rule p, (2) the sequencing of two programs P and Q denoted by $(P;Q)$, or (3) the iteration of a program P as long as possible, denoted by $P\downarrow$, which is equivalent to a sequencing $(P;(P;(P\cdots)$ until the rule no longer applies.

Graph constraints are similar to OCL constraints for models. They are defined inductively as nested conditions, but for the sake of simplicity we consider a very basic form $\exists(C)$ where C is a graph. A graph G satisfies such a constraint if G contains a subgraph isomorphic to C. This form is suitable to express test objectives which typically require particular patterns to exist in models.

Next, we present the example that will help us illustrate our proposal.

4 Example: Code Generation

We aim to apply our approach to a realistic code generator from Simulink[7] to Ada/C source code, under development in the collaborative research project *Project P*[8]. Simulink is a synchronous data flow language widely used by industrials for the design of control algorithms. The code generator consists of a chain of up to 12 model transformations (depending on configuration options), including flattening of nested structures, sequencing, code expansion and optimisation. We consider the *Code Model Generation* (CMG) transformation step of

[7] MathWorks Simulink, `http://www.mathworks.com/products/simulink/`

[8] Project P, `http://www.open-do.org/projects/p/`

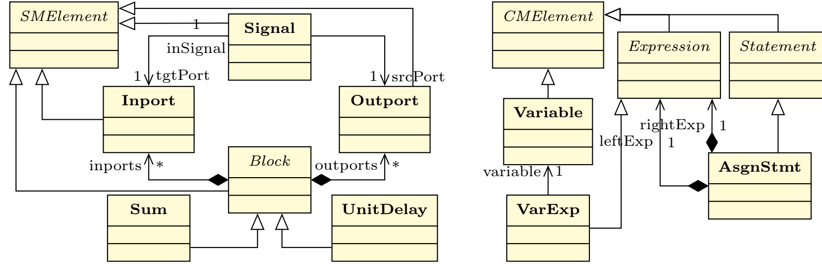


Fig. 2: Input and Output Metamodels

this chain to illustrate our translation to the AGT semantics. Then, considering a postcondition on the output of CMG, we construct a precondition on its input.

CMG transforms a Simulink model into a model of imperative code. A simplified version of the input metamodel is shown on the left side of Figure 2. Computation blocks such as *Sum* or *UnitDelay* receive data through their *Inports* and send the result of their computation through their *Outports*. *Signals* convey data from a source *Outport* to a target *Inport*. The output metamodel of CMG shown on the right side of Figure 2 features variables (*Variable*), expressions (*Expression*), references to variables (*VarExp*) and imperative code statements. In particular, an assignment statement (*AsgnStmt*) assigns its *rightExp* expression to its *leftExp* which typically is a reference to a variable.

Listing 1.1: The Code Model Generation ATL transformation

```
rule O2Var { from oport : SMM!Outport
             to   var   : CMM!Variable }

rule S2VExp { from sig    : SMM!Signal
              to   varExp : CMM!VarExp (variable <- sig.srcPort) }       -- Resolve

rule UDel {
  from block        : SMM!UnitDelay
  to   assgnStmt    : CMM!AsgnStmt (rightExp <- outVarExp,
                                    leftExp <- memVarExp1),
       memAssgnStmt : CMM!AsgnStmt( rightExp <- memVarExp2,
                              leftExp <- block.inports->at(1).inSignal), -- Resolve
       memVar       : CMM!Variable,
       outVarExp    : CMM!VarExp(variable <- block.outports->at(1)),     -- Resolve
       memVarExp1   : CMM!VarExp(variable <- memVar),
       memVarExp2   : CMM!VarExp(variable <- memVar) }
```

The ATL implementation of the CMG transformation consists of the 3 matched rules in Listing 1.1. The first rule creates a *Variable* for each *Outport* of the input model, and the second one creates a *VarExp* for each *Signal*. Note that the second rule requires resolving the *Outport* at line 5 into a *Variable* and will be used to illustrate our modeling of the resolve mechanism in AGT. The last rule creates 2 assignment statements referencing a *Variable* created by the same rule at line 13, a *VarExp* resolved at line 12, and a *Variable* resolved at line 14.

As for the test objective, we consider it directly in the graph constraint form in Figure 3. It requires that an assignment statement exists where both the source

Fig. 3: Example Test Objective

and the target of the assignment are references to variables. This pattern matches the objects created by ATL rule *UDel*, and thus requires resolve operations.

5 ATL to Algebraic Graph Transformation

This section introduces our main contribution, the translation of ATL transformations to artifacts of an algebraic graph transformation: a type graph, graph transformation rules and a high-level program. Given the rewriting semantics of AGT and the exogeneous nature of the transformations we consider, we choose to model the ATL transformation as a rewriting of the input graph that adds the output elements. Consequently, the type graph includes types corresponding to both the input and the output metamodels. As explained in Section 3.2, the correspondence of metamodel elements to graph type elements is straightforward [3], and the resulting type graph is depicted in Figure 4. In addition, *tracing* node types are added to support the ATL resolve mechanism. First, an abstract *Trace* node relates source objects (*SMElement*) to target objects (*CMElement*) of ATL rules. Second, for each ATL rule, a concrete trace node (named *<atlrule-name>_ Trace*) references the actual source and target types of this rule. These trace nodes will be used by the graph transformation rules, as explained next.

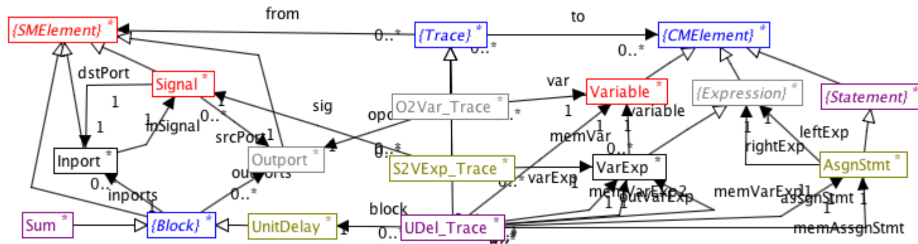
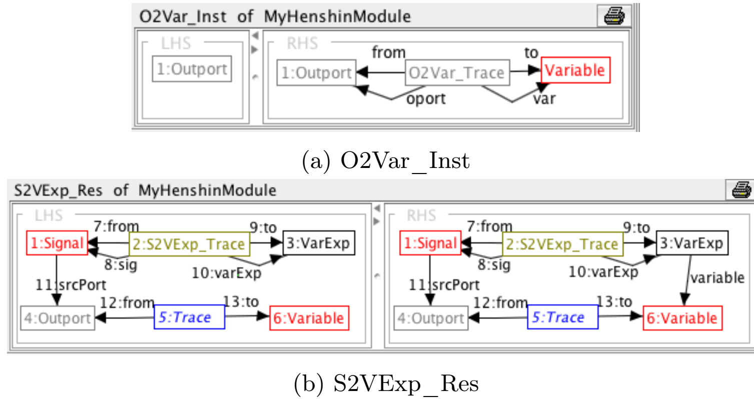


Fig. 4: Resulting Type Graph in AGG

Much like the execution semantics of ATL, the graph transformation starts with a set of *instantiation* rules that create output nodes without linking them. For example, *O2Var_ Inst* in Figure 5a matches an *Outport* and creates a *Variable* and a concrete trace *O2Var_ Trace* relating the source and target nodes (numbers indicate mapping by the rule morphism). Then, a second set of *resolving* rules rely on the trace nodes produced in the first phase to link output nodes. For example, *S2VExp_ Res* in Figure 5b matches an *Outport* and a *Trace* node

to find the resulting *Variable* and create the *variable* edge. Thus the elements created in the RHS of Figure 5a (*O2Var_ Trace* and *Variable*) are matched later by the LHS in Figure 5b (*Trace* and *Variable*). Note the use of abstract *Trace* nodes in the resolving rules to allow resolving with any rule as long as the number and types of source and target elements match, as per the ATL semantics.

Finally, a high-level program implements the two phases by iterating instantiation rules first and resolving rules second, yielding the following for CMG: $P = O2Var_Inst\downarrow; S2VExp_Inst\downarrow; UDel_Inst\downarrow; S2VExp_Res\downarrow; UDel_Res\downarrow$



(a) O2Var_Inst

(b) S2VExp_Res

Fig. 5: GTS rules translated from ATL rules

Having translated the ATL transformation to the AGT semantics, we next explain how we use precondition construction to back-propagate test objectives.

6 Transformation of Postcondition to Precondition

In [9], Habel, Pennemann and Rensink formally define a construction of *weakest precondition* for high-level programs in the interest of proving transformation correctness. Given a program and a postcondition, the weakest precondition is a constraint that characterizes all possible input graphs that lead to the termination of the program with a final graph satisfying the postcondition. A precondition construction is defined for one rule application and applied inductively to the sequence of rules defined by the program. In the case of $P\downarrow$ programs each number of iterations of P from 0 to ∞ must be considered, making the construction theoretically infinite.

However, in contrast with proof of correctness, we actually do not need to compute the *weakest* precondition. Since the final goal is to find a test model satisfying the test objective, computing one sufficient precondition would be enough. To do so, we limit iterations of rules in the program to a bounded number, making the precondition construction finite (the choice of bounds remains

an open point at this stage). For example we can bound the CMG transformation to two applications of *O2Var* and one application of each of the other rules: $P = O2Var_Inst; O2Var_Inst; S2VExp_Inst; UDel_Inst; S2VExp_Res;$ $UDel_Res$

As for the precondition construction of each rule, the theoretical construction requires to consider all possible overlaps of the RHS of the rule with the graph of the postcondition. Each overlap represents a way in which the rule may contribute to the postcondition. For each overlap, we perform an operation similar to a backwards execution of the rule[9] and thus construct a sufficient precondition.

7 Prototype and Results

We have prototyped our approach using the *Henshin* and *AGG* frameworks. *ATL2AGT* is implemented with the Henshin API, and an existing service is used to export the artifacts to AGG. Precondition construction is not readily available in AGG, so we have implemented *Post2Pre* using the existing services such as generating overlaps of two graphs and constructing a pushout complement. For the example test objective introduced in Figure 3, two of the preconditions we obtain are shown in Figure 6. The existence of one of these patterns in input models ensures that the *UDel* rule is able to execute and resolve the necessary elements to produce the pattern required by the test objective.

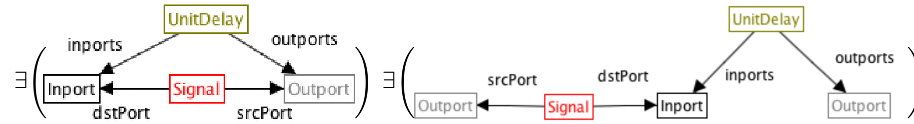


Fig. 6: Preconditions Computed for the Example Test Objective

8 Conclusion

In this paper we have approached the problem of testing model transformation chains with two main concerns: achieving high test coverage and using test models in the input language of the chain to ease the analysis of detected errors. To this end, we have proposed to extend existing approaches of test objective generation with a method to propagate intermediate test objectives back to the input language. Central to this method is the transformation of postconditions of one transformation step into preconditions, which was the focus of this paper. We have contributed a first translation from ATL semantics into the AGT semantics and adapted the theoretical precondition construction to achieve our goal.

[9] the formal construction is a *pushout complement*

In future work, we plan to investigate the *OCL2GC* step of our approach and alleviate the limitation to structural aspects by handling object attributes based on works such as [4,12,14]. Moreover, we plan to work towards test-suite minimality [2] by allowing a test model to cover several test objectives across the chain and only back-propagating non-satisfied test objectives.

References

1. B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, June 2010.
2. E. Bauer, J. Küster, and G. Engels. Test suite quality for model transformation chains. In *Objects, Models, Components, Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 3–19. Springer Berlin Heidelberg, 2011.
3. E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software & Systems Modeling*, 11(2):227–250, 2012.
4. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Synthesis of OCL pre-conditions for graph transformation rules. In *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 45–60. Springer Berlin Heidelberg, 2010.
5. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer, 2006.
6. C. Gonzalez, F. Buttner, R. Clariso, and J. Cabot. EMFtoCSP: A tool for the lightweight verification of EMF models. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*, pages 44–50, June 2012.
7. C. González and J. Cabot. ATLTest: A white-box test generation approach for ATL transformations. In *Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 449–464. Springer Berlin Heidelberg, 2012.
8. E. Guerra. Specification-driven test generation for model transformations. In *Theory and Practice of Model Transformations*, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin Heidelberg, 2012.
9. A. Habel, K.-H. Pennemann, and A. Rensink. Weakest preconditions for high-level programs. In *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 445–460. Springer Berlin Heidelberg, 2006.
10. F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin Heidelberg, 2006.
11. J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static analysis of model transformations for effective test generation. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 291–300, 2012.
12. C. M. Poskitt and D. Plump. A Hoare calculus for graph programs. In *Graph Transformations*, volume 6372 of *Lecture Notes in Computer Science*, pages 139–154. Springer Berlin Heidelberg, 2010.
13. RTCA. *DO-330 : Software Tool Qualification Considerations*. 2011.
14. J. Winkelmann, G. Taentzer, K. Ehrig, and J. 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(0):159 – 170, 2008. Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006).