Uncovering errors in ATL model transformations using static analysis and constraint solving

Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara
Universidad Autónoma de Madrid
{Jesus.Sanchez.Cuadrado, Esther.Guerra, Juan.deLara}@uam.es

***Abstract*—Model transformations play a prominent rôle in Model-Driven Engineering (MDE), where they are used to transform models between languages; to refactor and simulate models; or to generate code from models. However, while the reliability of any MDE process depends on the correctness of its transformations, methods helping in detecting errors in transformations and automate their verification are still needed.**

To improve this situation, we propose a method for the static analysis of one of the most widely used model transformation languages: ATL. The method proceeds in three steps. Firstly, it infers typing information from the transformation and detects potential errors statically. Then, it generates OCL *path conditions* for the candidate errors, stating the requirements for a model to hit the problematic statements in the transformation. Last, it relies on constraint solving to generate a test model fragment or *witness* that exercises the transformation, making it execute the problematic statement.

Our method is supported by a prototype tool that integrates a static analyzer, a testing tool and a constraint solver. We have used the tool to analyse medium and large-size third-party ATL transformations, discovering a wide number of errors.

***Keywords*-Model-Driven Engineering, Model Transformation, Static Analysis, Constraint Solving, Verification and Testing.**

I. Introduction

Model transformation is the main enabler of automation and model manipulation in MDE [30]. The definition of a transformation is typically used many times, likely in different projects. Hence, transformations need to be thoroughly tested to guarantee the reliability of any MDE solution [24].

The verification of model transformations is an active area of research [2], and much effort has been spent in verifying transformations defined with formal languages (e.g., based on graph transformation [10]). However, most transformation languages used in practice lack a fully formal foundation or a theory enabling their verification. One reason is that many of them rely on the Object Constraint Language (OCL) [22], which provides further expressiveness but makes transformation analysis difficult. This is the case of the Atlas Transformation Language (ATL) [17], one of the most widely used languages in industry and academia.

ATL is a weakly typed language, which makes ATL transformations prone to typing errors, like using a property of a subtype class in an expression yielding a supertype, or omitting the initialization of a mandatory feature. Such ill-typed transformations are accepted by the ATL engine; hence, errors can only be discovered upon feeding the engine with an input model making the transformation execute the incorrect statement. This testing process is manual, which poses several drawbacks: it is difficult to manually identify the erroneous parts of a transformation, the manual creation of input test models is tedious and time-consuming, and developers need to manually check that the model resulting from the transformation is well-formed. Even if some existing works automate the generation of input test models, they mainly propose black-box generation criteria like meta-model [27] or requirements [14] coverage, neglecting the detection of typing errors.

In this paper, we propose a method directed to discover errors in ATL transformations by combining static analysis and constraint solving. First, static analysis detects statements of the transformation that contain errors or are potentially problematic. While some of these problems will always raise errors when executing the transformation, others can never occur either because the transformation is written in such a way that it prevents the error, or because the only input models that would trigger the error are not valid according to the input meta-model integrity constraints.

Hence, as a complementary technique, we use constraint solving to find an input model that makes the transformation execute the erroneous statement, thus confirming the existence of a problem in the transformation, and helping the developer to understand and reproduce the error. For this purpose, given a potentially problematic statement, we build an OCL path condition stating the features needed in an input model to enforce the execution of the statement. A constraint solver uses this condition and the meta-model definition to generate a candidate input model. If no model is found, we discard the problem. We call the generated model a *witness* [31], because it signals a transformation error.

The method is supported by a tool integrated with the ATL editor. It uses a type checker for ATL transformations [9], extended with type error detection and OCL path condition generation. The tool integrates the USE Validator [18] for witness generation, and the *mtUnit* transformation testing language [14] to automate the testing process. We have evaluated our proposal against third-party transformations released as ATL use cases[1], discovering a wide number of errors.

Altogether, the *contribution of this work* is a novel method to detect errors in ATL transformations which relies on: (i) static analyis to detect potential problems based on the typing information, (ii) the construction of OCL path conditions

[1] http://www.eclipse.org/atl/usecases/

leading to problematic statements, and (iii) the generation of witness models from the path conditions using model finders. While we have developed the method for ATL, it could be adapted for other languages relying on OCL, such as those of the QVT family [23]. Additionally, we report an initial evaluation over a set of public transformations.

Paper organization. Section II outlines our approach, which is detailed in the following sections: Section III introduces a running example, Section IV explains the static analysis phase and the recognizable problems, and Section V shows how to generate OCL path expressions and witness models. Section VI presents our tool, while Section VII evaluates our technique. Section VIII compares with related work, and Section IX draws conclusions and lines for future work.

II. Overview

Our approach detects potential problems in a transformation using static analysis. If a problem cannot be statically guaranteed to be an error, we generate a witness model that confirms (or falsifies if it does not exist) the problem.

Fig. 1 outlines our proposal. First, we perform the static analysis of the transformation. For this purpose, the ATL transformation is parsed to obtain its abstract syntax model. Then, we perform a type checking of the transformation that annotates the ATL model with type information and detects some typing errors and warnings, some of which may be only potential and need to be confirmed by a witness model. Next, we perform an additional analysis to uncover more complex problems. This step generates a dependence graph [11] of the transformation that makes explicit the control flow dependencies between the elements of the transformation, such as which rules may map a source object to a target object.

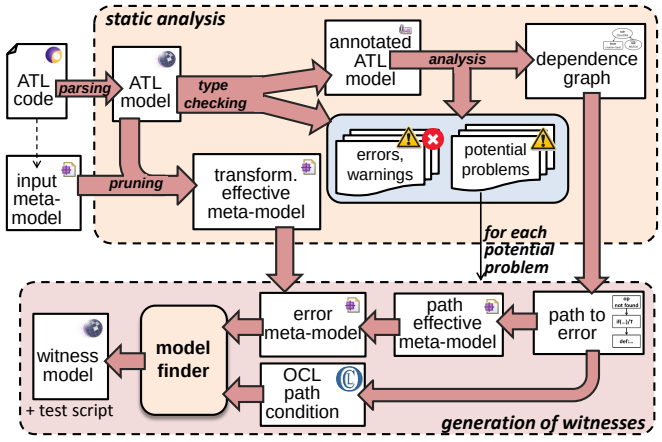


Fig. 1. Overview of the process.

Then, for each potential problem or a particular error selected by the user, we search a witness model that allows reproducing the error. To improve the performance of the search, the static analysis phase calculates the transformation effective meta-model [28], which is the smallest subset of the input meta-model that is relevant for the transformation (i.e., the classes and features accessed by the transformation code). This process is called *pruning*. Then, for each problematic statement in the ATL transformation, the dependence graph is traversed to obtain the corresponding *error path*. This is a subgraph (a slice of the transformation) that represents the set of elements with their dependencies that the execution flow must traverse to trigger the error. From the error path, we generate the following two artefacts: an *OCL path condition* stating the features required from the witness model, and the *path effective meta-model* of the error, which includes the meta-model elements involved in the error. The path effective meta-model and the transformation effective meta-model are used to compute the *error meta-model*, which extends the path effective meta-model with the mandatory classes and features needed to obtain a model conformant to the original meta-model. We compute this error meta-model to improve the performance of the model search by providing a smaller scope than the real source meta-model. Finally, we feed the error effective meta-model and the OCL path condition into a model finder (i.e., a constraint solver) in order to obtain a witness model that triggers the error at runtime; if no model is found up to a given bound, we discard the problem as spurious.

The next section introduces a running example that will be used to illustrate the different steps of this process.

III. Running Example

A transformation is written against its source and target meta-models. These describe the structure of the models manipulated by the transformation, including the allowed types and relations, type and cardinality of features, and additional constraints typically expressed as OCL invariants.

Transformation languages vary between *strongly typed* such as Kermeta [16], where all types are resolved at compile time and the abstract syntax is annotated with type information, and *dynamically typed* such as ATL, where type checking is performed at runtime (although the ATL IDE has autocompletion facilities, types are not enforced). In this paper, we describe our approach to analyse transformations written in dynamically typed languages, focusing on its application to ATL. Some aspects that can be analysed statically include type-correctness relative to the source and target meta-models, applicability and dependency of transformation rules, detection of unused helpers, and certain performance issues.

As a running example, Listing 1 shows a transformation from UML Activities to Intalio BPMN[2], exhibiting common errors in faulty ATL transformations. Fig. 2 shows an excerpt of its source/target meta-models. We have modified some cardinality in Intalio's meta-model to illustrate some errors.

```
1  module UML2Intalio;
2  create OUT : Intalio from IN : UML;
3
4  helper context UML!Activity def :
5    allPartitions : OclAny =
6      self.partition->collect(p | p.allPartitions)->flatten();
7
8  helper context UML!ActivityPartition def :
9    allPartitions : Sequence(UML!ActivityPartition) =
10     self.subpartition->collect(p | p.allPartition)->flatten();
```

[2]http://www.intalio.com/products/bpms

```
11
12 rule activity2diagram {
13   from a : UML!Activity
14   to m : Intalio!BpmnDiagram (
15     name <− a.name,
16     pools <− pool
17   ),
18   pool : Intalio!Pool (
19     name <− 'main (' + a.allPartitions−>size() + ' lanes)',
20     lanes <− a.allPartitions,
21     sequenceEdges <− a.edge−>
22       select(e | e.oclIsKindOf(UML!ControlFlow))−>
23       collect(e | if e.source.oclIsKindOf(UML!InitialNode) then
24                     thisModule.flow_initial(e)
25                   else e endif ),
26     vertices <− a.partition−>collect(p | p.node)
27   )
28 }
29
30 lazy rule flow_initial {
31   from cf : UML!ControlFlow
32   to b : Intalio!SequenceEdge (
33     name <− if cf.name.oclIsUndefined() then
34               'initial' else cf.Name endif,
35     source <− cf.source,
36     target <− cf.target
37   )
38 }
39
40 rule initialnode {
41   from initial : UML!InitialNode ( initial.incoming.isEmpty() )
42   to activity : Intalio!Activity (
43     name <− 'empty initial activity'
44   )
45 }
46
47 rule initialnode_timer {
48   from initial : UML!InitialNode (
49     initial.incoming−>exists(edge |
50       edge.source.oclIsKindOf(UML!AcceptEventAction))
51   )
52   to activity : Intalio!Activity (
53     activityType <− #EventStartMessage
54   )
55 }
56
57 rule objectnode {
58   from obj : UML!ObjectNode
59   to art : Intalio!Artifact
60 }
```

Listing 1. ATL transformation.

An ATL transformation can include rules and OCL helpers. A rule identifies a configuration of source elements defined by a source pattern, normally made of a single element, for which it generates one or more target elements. The most commonly used kinds of rules in ATL are *matched* rules and *lazy* rules. A matched rule is implicitly executed once for each occurrence of its source pattern, while a lazy rule is executed only when it is explicitly invoked. For instance, the activity2diagram matched rule (line 12) transforms every Activity object into an object of type BpmnDiagram and a Pool. In contrast, the flow_initial lazy rule (line 30) will be executed upon its explicit invocation through the statement thisModule.flow_initial(e) (line 24), and hence its execution depends on the execution of the caller rule activity2diagram.

The features of the target objects created by a rule are initialized using *bindings* with the syntax $feature <- OclExpr$. In the case of references, each source object appearing in $OclExpr$ is looked up in the transformation execution trace to retrieve the target object in which it was transformed by some rule. This is called *binding resolution*.

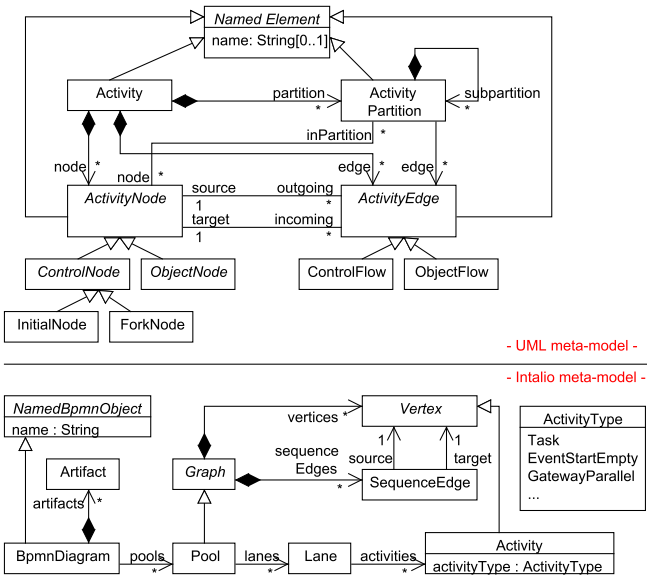


Fig. 2. Excerpts of UML meta-model (up) and Intalio meta-model (down).

Finally, *helper*s can be seen as operations or derived features attached to a given type (lines 4 and 8). Helpers that are not attached to a type act as global functions.

While the ATL compiler processes the example transformation without reporting problems, its execution may fail or produce incorrect target models for the following reasons:

1) **Unresolved binding** (line 20). There is no rule transforming ActivityPartition (the result of the allPartitions helper) into Lane (the type expected by feature lanes).
2) **Possibly unresolved binding** (lines 35 and 36). The right part of both bindings has an ActivityNode object, which can be resolved by rules initialnode, initialnode_timer, or objectnode. However, there are no rules dealing with ForkNode, which is also a subclass of ActivityNode, and there may be InitialNode objects not accepted by the guards in lines 41 and 49. Hence, we need to find a witness model to confirm whether there are valid models containing objects rejected by any of these three rules, or otherwise discard this error.
3) **Invalid target in resolved rule** (line 26). If the binding is resolved by rule objectnode, it will assign an Artifact object to the vertices property, which has type Vertex, hence producing an ill-typed target model. Thus, we need to check whether this resolution is possible by generating a witness model.
4) **Missing binding for compulsory feature** activityType of Activity (lines 42–44).
5) **Potentially conflicting rules** (lines 40 and 47). ATL does not allow two matched rules to be applicable on the same source object. This can be controlled by defining exclusive guards for the rules, but checking this property manually is difficult.
6) **Unnecessary flatten operation** (line 6).
7) **Invalid property** (name in line 33) **and helper call** (allPartition instead of allPartitions in line 10).

In the following, we explain how we detect these errors by gathering type information and performing static analysis, and how we generate witness models for the detected errors.

IV. Static Analysis

The first phase of our approach is a static analysis, which includes the type checking of the transformation and the analysis of the transformation dependence graph.

A. Type checking ATL transformations

We first perform a type analysis of the transformation to determine if it satisfies the syntactic constraints imposed by the source and target meta-models. This is a complex task in ATL due to its dynamic nature. For instance, the allPartitions helper in lines 4–6 specifies OclAny as return type, but its body makes clear that Sequence(ActivityPartition) is a more precise typing. The call a.allPartitions->size() in line 19 works fine at runtime, but a naive type checker that takes OclAny as the type for a.allPartitions would signal an error. The ATL compiler does not report an error because it does not perform any type checking. We use type inference to determine the type of OCL expressions and compare it with the declared type, reporting warnings if needed. This allows a correct type checking of the previous expression.

Type checking is performed in two passes. First, we annotate the variable declarations, rule pattern types and helpers with the types they explicitly declare. Then, we perform a bottom-up traversal of the Abstract Syntax Tree (AST) propagating types, annotating each node in the AST, and reporting errors and warnings. Currently, we do not support recursive helpers, but we just use their declared type.

Fig. 3 shows the types used to annotate the AST nodes, including typical OCL types such as primitive and collection types. Metaclass refers to a class defined in the source or target meta-model. TypeError is a marker indicating that a node in the AST is problematic and cannot be given a type. ThisModule refers to the transformation. Reflective allows handling queries to an object's metaclass at runtime, although our support for this is limited. Undefined values (i.e., *null*) are represented with Undefined. A special undefined value called EmptyCollection is used for expressions like Sequence {} to indicate that the type of the collection's elements is unknown.

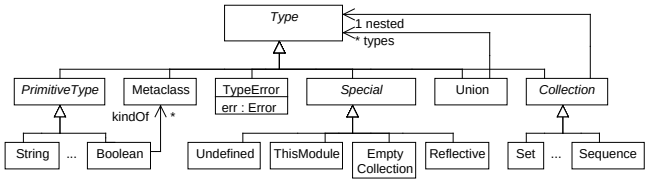


Fig. 3. Types used to annotate the AST.

We use the type Union to keep track of the multiple types that an expression may yield. For example, the expression anActivity.node->union(anActivity.partition)->first() is given the type Union(ActivityNode, ActivityPartition). This allows reasoning about the operations that can be performed on the result of the expression. In this case, it is possible to access the name feature because both classes inherit this feature from NamedElement, but an access to incoming raises a warning because only ActivityNode defines the feature.

Another issue is that ATL does not support the oclAsType casting operation, which complicates the analysis as there is no explicit way for downcasting. In the example, line 22 downcasts implicitly by selecting only ControlFlow edges, ensuring that the call to flow_initial is safe. We deal with this form of downcasting by means of the kindOf feature in the Boolean type. Thus, the expression e.oclIsKindOf(UML!ControlFlow) is given the type "Boolean plus ControlFlow", and if it appears in iterators like select, the type of the resulting collection is limited to types appearing in the kindOf feature (Sequence(ControlFlow) in this case). We also track the uses of oclIsKindOf in if conditions and rule filters to implicitly downcast the checked expressions.

In some cases, the type errors detected statically need to be confirmed by finding a witness model. For example, the aforementioned implicit downcasting mechanism may report a false problem. As we will see in Section V, to speed up the generation of witnesses, we use the effective meta-model of the transformation (i.e., classes and features accessed by the transformation). This is calculated from the meta-model footprint obtained in the analysis phase, using a pruning algorithm similar to the one presented in [28].

Altogether, the type checking phase annotates the nodes of the AST of the transformation. This allows identifying some typing errors and warnings. Table I shows the most important errors we are able to detect (at this point, the errors in phase *typing*). We will explain this table in Section IV-C.

B. The transformation dependence graph

The type checking phase annotates the transformation in a separate model conforming to an extended ATL meta-model enriched with the type of the nodes and the control and data flow of the transformation (see Fig. 4). This model, called *transformation dependence graph* (TDG), is analysed in a second stage to uncover further potential problems.

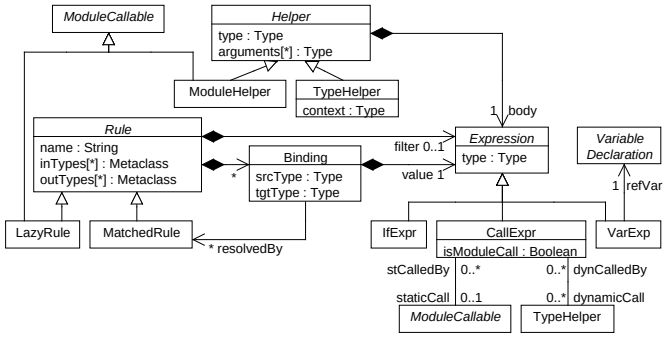


Fig. 4. Excerpt of extended ATL meta-model.

In Fig. 4, each Expression refers to the type assigned by the type checker. The control and data flow of OCL expressions is represented in the abstract syntax of ATL following the containment relationships of expressions. This suffices also

for loops, which can only be implemented through explicit iterators and there are no breaking statements (e.g., break, return). Moreover, since there are no variable reassignments, any variable usage VarExp only needs to refer to the original variable declaration.

Analysing the dependencies between a call expression and the helpers that may process the call is challenging. We use a similar approach to [20], where every helper that may resolve a call is added to the TDG (reference dynamicCall). Similarly, given a binding, we compute all matched rules that may resolve it (reference resolvedBy). Lazy rules are statically resolved because they are invoked like global functions (class ModuleCallable and reference staticCall).

Example. The TDG excerpt in Fig. 5 shows a call to the lazy rule flow_initial from the matched rule activity2diagram, and the rules resolving the binding source <- cf.source. The type of cf.source in the right part of the binding is ActivityNode, and hence rules initialnode, initialnode_timer, and objectnode may resolve it, as all have a subtype of ActivityNode in its from part.

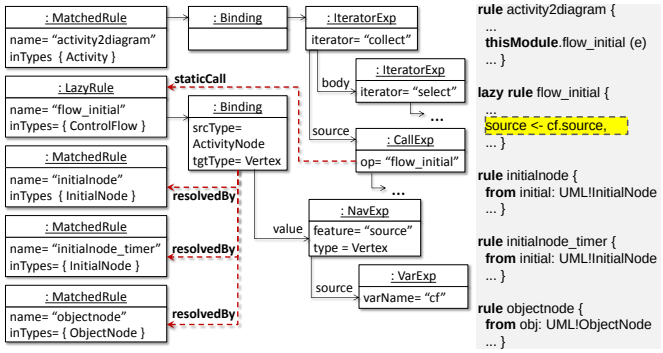


Fig. 5. Excerpt of TDG with elements involved in the "Possibly unresolved binding" error (line 35 in Listing 1).

C. Analysis of problems

Table I summarizes the problems we detect statically. The second column indicates in which phase of the analysis the error is detected, either the type checking (Section IV-A) or the analysis of the TDG (Section IV-B). The third column states if the problem needs to be confirmed by generating a witness: never (i.e., static analysis), sometimes, or always. The last column indicates the severity of the problem, distinguishing between execution errors and warnings regarding style, performance, or unexpected transformation behaviour. Moreover, the severity of errors can be *error-always* if the transformation execution always yields a runtime error; *error-source* if runtime errors will occur only for models with certain features; and *error-target* if the transformation will not crash, but it will yield an incorrect target model.

We consider five kinds of errors, corresponding to the five blocks in Table I. *Type errors* refer to disconformities between the types used in the transformation and those declared in its source and target meta-models. For instance, using an invalid type name, or accessing a feature in an object whose type does not define the feature, raises an error.

Description	Phase	Precision	Severity
Typing (with respect to source/target meta-model and helper definitions)			
Invalid meta-model or type name	typing	static analysis	error-always
Operation/feature not found	typing	static analysis	error-source
Operation/feature not found, but declared in subclass	typing	sometimes solver	error-source
Feature not found, but declared as operation	typing	static analysis	error-source
OCL navigation			
Wrong iterator body type	typing	static analysis	error-source
Wrong kind of accessor operator (→ vs. ".")	typing	static analysis	error-source
Flatten over non-nested collection	typing	static analysis	warning-style /performance
Transformation integrity constraints			
Read access to target model	typing	static analysis	warning-behaviour
Filter in lazy rule	typing	static analysis	warning-style
Matched rule without output pattern	typing	static analysis	warning-style
Target meta-model conformance			
Compulsory target feature not initialized	analysis of TDG	static analysis	error-target
Binding resolved by rule with invalid target	analysis of TDG	sometimes solver	error-target
Transformation rules			
No rule to resolve binding	analysis of TDG	static analysis	warning-behaviour
Possibly unresolved binding	analysis of TDG	always solver	warning-behaviour
Conflicting rules	analysis of TDG	always solver	error-source
Non-invoked rule	analysis of TDG	static analysis	warning-behaviour

TABLE I
PROBLEMS DETECTED STATICALLY IN ATL TRANSFORMATIONS.

Errors can also be in *OCL navigation* expressions. The first two errors of this kind raise a runtime error when executing the erroneous statement. Expressions that include the flatenning of a non-nested collection are performing an unneccesary operation, hence this is reported as a warning.

Some *integrity constraints* regarding the semantics of ATL are not enforced by the ATL IDE, and therefore can be violated by transformations. Even if these violations will not produce runtime errors, they may cause unexpected results. For instance, the target model of an ATL transformation should be write-only, and therefore we report on any reading operation over the target model. Another example is the definition of filters in lazy rules, allowed by the ATL IDE, but ignored by the execution engine.

Some problems concern the *conformance to the target meta-model* of the produced output models. These problems are unnoticed during the transformation execution, becoming apparent when the non-conformant output model is processed by another transformation or modelling tool. Ensuring conformance to the target meta-model statically is challenging, but we can detect some problems of this type. For instance, we report an error if a target compulsory feature (i.e., with positive lower cardinality and no default value) is left unbound, as this will always produce an incorrect target model. Bindings resolved by a rule with an invalid target type are a potential problem, which may require finding a witness model.

Finally, problems related to the dependency between *transformation rules* include missing rules for the types used in bindings; conflicting rules, e.g., if two rules have non-exclusive

guards; and dead-rules that will never be executed, e.g., lazy rules that are invoked nowhere.

Example. In Fig. 5, an error occurs if cf.source (line 35 in Listing 1) holds an object that cannot be resolved by any of the rules initialnode, initialnode_timer and objectnode. This would be the case for an object of type ForkNode (incompatible with the from part of all rules), or an object of type InitialNode that does not satisfy the guards of initialnode and initialnode_timer. Hence, this problem is marked as "possibly unresolved binding". The next section explains how to check whether a model with such characteristics exists.

V. Generation of witness models

Some problems detected in the static analysis phase can be signalled accurately, but others require finding a witness model proving that the error can occur in practice. Failing to find a witness model may happen in two cases: when the meta-model includes constraints preventing the existence of problematic models, or when the transformation contains expressions that prevent the error at runtime.

The third column in Table I shows the problems that require this additional step (cells marked as "sometimes solver" or "always solver"). To generate a witness model for a given problem, we calculate all possible execution paths leading to the problem, and for each path, we derive an OCL expression that characterizes the input models that make the transformation execute the problematic statement. Then, we use constraint solving to assert whether there exists a model that satisfies this OCL expression as well as the meta-model integrity constraints (cardinality of associations, compositions and OCL constraints). Next, we explain these steps in detail.

A. Extracting the error path

To calculate the paths leading to an error, we start from the node that contains the problematic statement, and traverse the transformation control flow back until reaching a matched rule. There are seven types of nodes in the graph (see Fig. 4): *Matched rule* nodes to represent the execution of a matched rule, which typically starts the execution flow; *Loop* to represent the iteration over a collection; *If* to represent a condition and the branch for which the control flow passes towards the error; *Let* to represent a variable definition and its scope; *Call* to represent an invocation to a helper or lazy rule; *Callable* to aggregate several paths starting from a helper or lazy rule; and *Subexpr* to identify the valid part of a problematic expression. Additionally, there is a problem node for each type of problem, which gathers specific error conditions (see Section V-B).

Example. Fig. 6 shows the error path built from the problematic binding source $<-$ cf.source (label 1). To exercise the binding, its owning rule flow_initial should be called. Since it is lazy, it must be explicitly invoked, and we look in the TDG for any call expression that invokes it. This information is stored in the TDG's reference stCalledBy. The Callable node aggregates all possible rule calls (label 2), just one in this case (label 3). The call (thisModule.flow_initial(e)) is performed within the true branch of a conditional expression (If node, label 4). The conditional expression is within a collect iterator (lines 23–25). To reach the iterator, the source collection of the iterator must exist (Loop node, label 5). Finally, the execution starts in the activity2diagram matched rule (label 6).

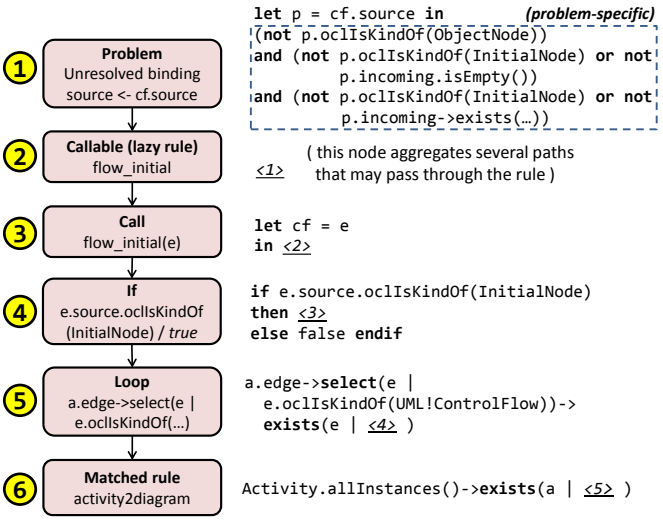


Fig. 6. Error path and OCL path condition.

B. Building the OCL path condition

Given the error path, we derive an OCL expression describing the models that make the transformation execute the problematic statement. To this end, we first collect the conditions found in every path to the error, and then add problem-specific conditions to trigger the error.

Our strategy to build the OCL path condition is a bottom-up traversal of the error path starting from the leaf nodes (i.e., nodes that trigger the execution flow). For each node, an OCL fragment is generated stating the conditions required for the control flow to continue towards the error. Then, the OCL fragment for the parent node is generated, which strengthens the path condition, until the problem node is reached. Note that each node in the error path has a single parent. Table II shows the OCL conditions generated from each path node type.

While the OCL condition generated from the error path is the same for every error type, some kinds of errors require including an additional problem-specific OCL condition. For example, if the problematic statement is an access to a feature of an object o of type T but the feature is defined in a subtype T' of T, and o is obtained iterating a collection c, we need to add c->exists(o | not o.oclIsKindOf(T')) to the OCL path condition to request some object of the problematic type in the collection.

Table III shows the specific conditions generated for problems with precision *sometimes solver* and *always solver*. For the rest of problems, we also support the generation of witness models, but in that case, the extra condition is simply true since it is enough to reach the statement.

Example. In Fig. 6, the OCL path condition is built starting from the *Matched rule* node (label 6). The generated OCL fragment enforces the existence of objects that can be matched

Element	ATL	OCL condition	Description
Matched rule	**rule** r { **from** t : T (guard(t)) ... }	T.allInstances()−>exists(t | **if** guard(t) **then** dependingNode **else** false **endif**)	The model should contain at least one object t with compatible type (T) satisfying the guard.
If expression	**if** condition **then** branchToError **else** theOtherBranch **endif**	**if** condition **then** dependingNode **else** false **endif**	The case in which the false branch leads to error just swaps the then/else parts.
Iterator expression	expr−>collect(it | exprWithError) −>followingOperations	expr−>exists(it | dependingNode)	The operator exists ensures that the collection contains some problematic element. Any following operation is ignored.
Navigation expression (mono-valued property)	expr.property.invalidAtt.otherNav	**not** expr.property.oclIsUndefined()	The condition demands the existence of property; thus, accessing the invalid attribute invalidAtt triggers an error.
Navigation expression (multi-valued property)	expr.property−>invalidOp()	**not** expr.property−>notEmpty()	The condition demands a non-empty property; thus, calling the invalid operation invalidOp triggers an error.
Call to helper with context	**helper context** MM!T **def** : aHelper : OclAny = self.exprWithError ... exprOfTypeT.aHelper	let genSelf = exprOfTypeT in genSelf.exprWithError	The body of the helper is inlined, replacing self with a new value (e.g., genSelf) that contains the receptor object. Parameters are passed in a similar way.
Call to helper without context, or lazy rule, or called rule	**helper def** : aHelper : OclAny = exprWithError ... thisModule.aHelper	exprWithError	Lazy rules and called rules are treated as global helpers, in which the "from" part (of a lazy rule) are parameters.

TABLE II
TRANSLATION OF ATL ELEMENTS INTO OCL CONDITIONS: CONTROL FLOW DIRECTIVES.

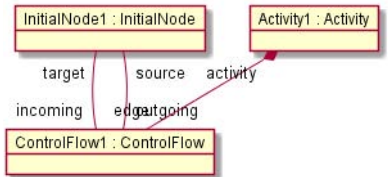
by the rule. Then, the condition for the existence of ControlFlow edges (label 5) is nested within the previous fragment, and similarly for the *If* node (label 4). A call to a lazy rule or a helper implies inlining its body within the generated expression, which requires passing parameters and binding the self variable in the case of helpers. This is done by using let expressions to create new variable scopes (label 3). No OCL is generated from the callable node (label 2) because this node just aggregates the paths passing through it. The last step adds a problem-specific condition to the OCL path condition (label 1). In particular, we need to check whether cf.source can contain an object that is not accepted by the rules initialnode, initialnode_timer and objectnode. Hence, following the third row in Table III, cf.source should either have an incompatible *from* type or not satisfy the rule guard, for each of the three rules.

C. Generation of witness models

The previous procedure yields an OCL path condition for each path to the error. A disjunction of these conditions is declared as the invariant of an artificial class in the meta-model (called ThisModule), which also contains the translation of the global variables and helpers defined in the transformation. This is done because OCL does not support global constraints, but they must be defined in the context of some class. Then, a model finder (like UMLtoCSP [8] or the USE Validator [18]), checks whether there is a model that satisfies all invariants (we force one object of type ThisModule, which later is discarded). Most model finders rely on bounded search, exploring models up to a certain size, typically given in terms of a range for the number of objects of each class. Hence, they employ the *"small scope hypothesis"* [1], [15] and assume that most constraints are satisfied by models of limited size.

To further constrain the search space, we use a pruned version of the input meta-model which only considers the types and features used in the OCL path condition, as well as the mandatory types and features needed to obtain a valid instance of the complete input meta-model. Using a pruned version of the input meta-model can drastically reduce the finding time, especially in the case of large meta-models, because the solver does not need to consider types irrelevant for the error. As an example, the next table shows the size of the pruned meta-model and the time it takes to find a witness model for the path condition in Fig. 6, compared to using the complete meta-model. In this example, the pruned meta-model reduces the finding time by 99.47%.

	pruned MM.	complete MM.
number of classes	84	249
number of associations	3	401
witness generation time (sec.)	0.18	34.40



Example. The figure to the right shows a witness model for the path expression of Fig. 6, obtained with the USE Validator. The model exercises lines 35 and 36 of the transformation, and it has a ControlFlow object with source and target pointing to an InitialNode which does not satisfy the guards of the rules initialnode and initialnode_timer. This makes the transformation to yield an invalid target model, containing a SequenceEdge with undefined source and target references.

VI. TOOL SUPPORT

Our method is supported by an Eclipse plugin (available at http://www.miso.es/tools/anATLyzer.html) that includes an ATL type checker, an analysis module, and relies on the USE Validator to generate witness models. The plugin is integrated in the ATL editor, so that error reporting and witness generation is seamlessly integrated in the IDE. Moreover, we provide a *mtUnit* script that automates the testing of the transformation with the generated witness models and reports the execution results.

Fig. 7 shows a screenshot of the tool. The detected problems are marked in the ATL editor (labels 1 and 2) and reported in

Element	ATL	OCL condition	Description
Operation/feature not found in T, but declared in its subclasses S1,...,Sn	<expr : T>.feature	**not** expr.oclIsKindOf(S1) **and** ... **not** expr.oclIsKindOf(Sn)	expr should have a type compatible with T, different from S1,...,Sn.

Binding resolved by rule with invalid target	feature <− <expr : T>
	...
	rule r { **from** t : T1 (guard(t)) ... }
		// If expr is mono−valued:
		guard(expr)
		// If expr is multi−valued:
		expr−>exists(t | guard(t))
			To ensure that expr contains objects accepted by rule r, expr needs to satisfy the rule guard. When T1 is a subtype of T an additional check, expr.oclIsKindOf(T1), is needed.

Possibly unresolved binding	feature <− <expr : T> ... **rule** r1 { **from** t : T1 (guard1(t)) ... } **rule** rn { **from** t : Tn (guardn(t)) ... }	(**not** expr.oclIsKindOf(T1) **or** **not** guard1(expr)) **and** ... (**not** expr.oclIsKindOf(Tn) **or** **not** guardn(expr))	expr should be incompatible with the rules declared for T or its subtypes. The OCL path condition when expr is multi-valued is similar.

Conflicting rules	**rule** r1 { **from** t1 : T1(guard1(t1)) ... }
	rule r2 { **from** t2 : T2(guard2(t2)) ... }
		// T <: T1 , T <: T2
		T.allInstances()−>exists(t | guard1(t) **and** guard2(t))
			We need an object accepted by both rules. T is the closer descendant of T1 and T2. If T1=T2, we take T=T1.

TABLE III
TRANSLATION OF ATL ELEMENTS INTO OCL CONDITIONS: PROBLEM-SPECIFIC CONDITIONS.

the Problems view (label 3). The quickfix facility of Eclipse allows generating witness models, visualized with PlantUML (label 4). These models can be automatically added to a *mtUnit* test suite for testing the transformation (label 5).

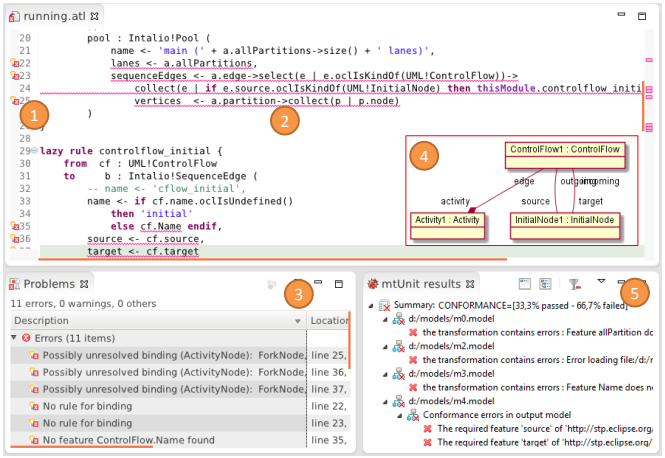


Fig. 7. Enhanced ATL editor.

Some practical issues arose during the development, due to incompatibilities of the OCL used in ATL and USE. For instance, since USE lacks derived attributes, we had to encode helpers as operations. The USE Validator only supports collections of type Set, does not support recursive functions, and attributes with lower bound 0 and a primitive type need a special encoding (as a Set plus a constraint). Built-in features of ATL, like refImmediateComposite, need to be emulated by OCL operations. Finally, the type checker of USE performs a type inference for collections which is less precise than ours, and it signals as errors some expressions accepted by our analyser; this was partly solved by recasting these expressions.

VII. EVALUATION

To evaluate our method, we have used some transformations from the ATL Use Cases repository. This is a documented collection of 26 non-trivial applications of model transformations, some of them from industrial projects. We only selected use cases ranked as "reaching completion" – which is the highest rank – meaning that the transformations are complete or only minor details are missing; hence, they are supposed to have few errors. Moreover, we discarded two use cases because the involved meta-models were not available, and two more because they exercised AMW but not ATL. Altogether, our evaluation considered 6 use cases with the highest rank, which amounts to 19 transformations because some use cases included several transformations.

Fig. 8 shows the results of our evaluation (we omit the error types that were not found in any transformation). The first rows of the table contain metrics of the complexity of the transformations. For use cases with several transformations, we show the average of each metric. The rest of the table shows the number of errors of each type uncovered. For the problems that may require the generation of a witness model (marked with *), the second line shows how many were confirmed statically, how many were confirmed by a witness model, how many were discarded because no witness exists, and how many remained unconfirmed for some reason (e.g., the OCL path condition included recursion).

Only one of the evaluated transformations does not present problems. Thus, our techniques are able to catch elusive errors and can help to improve the quality of transformations.

The most frequent error is the creation of target models where the cardinality of some compulsory feature is violated (*Compulsory target feature not initialized*). This is problematic if the erroneous target model is to be used as input in a chained transformation that expects correct input models.

In three uses cases (C, D and F), some feature or operation defined in a subclass is accessed over an expression whose static type is a superclass. Thus, the transformation may fail if instances of the superclass exist. Our analyser is able to discard this potential problem when a rule filter or the condition in an iterator ensure that such instances cannot reach the problematic statement. When this cannot be statically guaranteed, we generate a witness model. For this error type, we confirmed 111 statically, 11 were discarded because there was no possible witness model, and 29 problems remained unconfirmed.

Another problem signalled by the analyser are the bindings not resolved by any rule, or potentially unresolved. This indicates that the transformation does not handle some configuration of the source model. This is complex to detect by hand, as one needs to know the type of the right part of the binding,

	A	B	C	D	E	F	Total
Transformation metrics (average)							
Number of transformations	1	1	1	8	6	2	19
Lines of code	275	233	831	482	84	155.5	343.3
Number of helpers	6	0	15	39	1.3	2.5	10.6
Number of matched rules	14	5	8	6	4.2	9.5	7.8
Number of lazy/called rules	1	0	15	7	0.2	1.5	4.1
Source meta-model size (classes)	32	5	68	65	7.2	37.5	35.8
Target meta-model size (classes)	77	6	46	27	6.2	24.0	31.0
Detected errors and warnings:							
Typing (with respect to source/target meta-model and helper definitions)							
Invalid meta-model or type name				1			1
Operation/feature not found				10			10
Operation/feature not found, but declared in subclass*			2 [0/0/0/2]	122 [84/0/11/27]		27 [27/0/0/0]	151 [111/0/11/29]
OCL navigation							
Wrong iterator body type				1			1
Wrong kind of accessor operator				33	28		61
Flatten over non-nested collection				4	9		13
Transformation integrity constraints							
Read access to target model					3		3
Matched rule w/o output pattern				1			1
Target meta-model conformance							
Comp. target feat. not initialized	49		10	71[a]	27	24	181
Binding resolved by rule with invalid target*						6 [0/0/0/6]	6 [0/0/0/6]
Transformation rules							
No rule to resolve binding	1			5		2	8
Possibly unresolved binding*	8 [0/0/0/8]		1 [0/0/1/0]	7 [0/3/0/4]		9 [0/9/0/0]	25 [0/12/1/12]
Total number of problems	58	0	12	238	67	68	441

Evaluated use cases:
A: cpl to spl
B: pdl to tina
C: fiacre to lotos
D: models measurements
E: code clone tools
F: sbvr to uml

*Requires witness model. The cells show: total number of errors [errors confirmed statically / errors confirmed by witness / errors discarded because no witness was found / unconfirmed errors]

[a]The analyser reported 6 false errors due to imperative code.

Fig. 8. Result of evaluated transformations.

and then look up all transformation rules that may resolve it. The use cases contain 21 occurrences of this problem, 8 confirmed statically and 12 by generating a witness model.

A. Threats to validity

The main threat to the validity of our evaluation is the relatively low number of transformations analysed. Nonetheless, their large size and the fact that they are real case studies developed by third parties make them representative of the errors in a typical transformation. Moreover, the analysed transformations are public and have been likely revised and improved by a broad audience. It is expected that transformations which are not in public repositories are faultier. We plan to improve our evaluation with other ATL transformations. We also plan to perform an experiment directed to measure to what extent the development of new transformations using our techniques help reducing the number of errors.

Our implementation is so far available only for ATL. The features of other transformation languages may limit or impose other constraints in the static analysis phase. For instance, QVT-Relations does not have the limitation of write-only target models. However, we believe that our method could be adapted to other languages that rely on OCL as well.

Finally, there is no formal semantics for ATL, and most of the errors we capture are based on the explanations in the ATL guide and our own experience. In this sense, there is the risk that we have misinterpreted the semantics of ATL for some errors. To mitigate this, we have written tests to acknowledge our intuition about the behaviour of the ATL engine. Our aim is to extend such tests, so that they can serve as reference and test bed for ATL tooling developers.

B. Discussion

The analysed transformations range from purely declarative (i.e., only matched rules) to purely imperative (i.e., only called rules and imperative blocks). The static analyser always worked as expected and generated the corresponding path expressions except for a few imperative blocks.

For errors of type "always/sometimes solver", we have manually checked that the witness model that confirms or discards the problem is correctly generated. However, sometimes, the OCL code we generate cannot be processed by the USE Validator. This may happen due to three reasons, which we plan to tackle in the future:

- The OCL path condition generated for an error traverses nodes containing other errors. A solution would be to prioritize the order in which errors should be solved, fixing first the errors described by OCL path conditions that do not include other errors, and so on.
- Our type inference is more precise than USE's, which implies that some OCL/ATL expressions are deemed correct by our static analyser, but for USE are not. In fact, this is why most problems in the use case D could not be processed. We plan to develop a transformation from OCL/ATL to OCL/USE able to recast the required expressions using *oclAsType*, which is supported by USE.
- The USE Validator has some limitations, such as no support for Sequence collections and recursive functions.

Altogether, the evaluation shows that transformations thought to be stable may contain errors of diverse nature. Some may due to the dynamic nature of ATL, but others are inherent to rule-based languages (non-initialized features, wrong rule resolution, etc.). Our method has allowed uncovering a

surprisingly high number of problems, proving that this kind of techniques is urgently needed to help developers improving the quality of model transformations.

VIII. Related Work

Next, we review approaches for transformation verification [24], focussing on testing, analysis via constraint solving, static analysis, and slicing.

Transformation testing. A main challenge in transformation testing [3], [26] is the automated creation of input test models. Most proposals employ a black-box approach, and some rely on model finders to generate test models [14]. For example, in [29], *partial models* are used to reflect test intentions, and model finders complete them into full-fledged models conformant to a meta-model. Other approaches rely on coverage of the meta-model [12] or the transformation requirements [14] as criteria for automated model generation.

Few works focus on white-box testing. In [19], the authors use the effective meta-model of each transformation rule to derive test cases. In [13], a white-box testing approach for ATL produces a set of input models ensuring certain coverage. To this aim, the authors build a dependency graph by partitioning the transformation OCL expressions, and traverse the graph in each possible way to compute input models using model finders. They do not perform type checking or static analysis, but their goal is maximizing the variety of obtained models. Instead, our generation of model witnesses is driven by the problems found by the static analysis.

We follow a white-box approach to generate input models (witnesses) driven by static analysis. These models are maximally effective as they allow reproducing an error. However, our technique is not a replacement for the previous testing methods, but it is complementary, directed to typing errors.

Transformation analysis based on constraint solving. Many works use transformation models [4] to express transformations and use model finders for their analysis. A transformation model is the merge of the source and target meta-models, possible trace links between their elements, and OCL invariants expressing correctness conditions.

In [5], [6], ATL transformations are translated into transformation models, and model finders are used to check whether the transformation can produce a model not satisfying the output meta-model constraints. In [7], some analysis properties are defined for transformations based on their translation into OCL and their analysis with model finders. While this branch of works assumes a correctly typed ATL transformation, we focus on discovering typing errors.

Our contribution in this area is a novel technique, based on static analysis, extracting the slice of the transformation that corresponds to the path to an error, building an OCL expression with the conditions for a model to reach the error, and using model finders to generate one such witness model.

Static analysis of transformations. Even though static analysis has been used in graph transformation to detect, e.g., rule conflicts and dependencies [10], its use in transformation languages closer to programming languages, like ATL or QVT, is an exception. The literature reports on three main applications of static analysis: maintainability of transformations [25], [35], generation of input test models, and static detection of errors. We focus on the last two applications.

Regarding test generation, in [21], the effective meta-model of Kermeta transformations is used to generate input test models ensuring a proper coverage of the transformation. While Kermeta is strongly typed, ATL transformations may be ill-typed; thus, we focus on identifying typing errors and potential problems and generate witness models for them.

The use of static analysis to detect errors in transformations is mostly unexplored. In [36], the presented Java Façade for ATL can be used to build static analyses. We opted for a new API to integrate explicit rule dependencies and error handling information. Static type checking of VIATRA2 transformation patterns [33] relies on constraint solving to ensure a correct typing of the patterns' parameters w.r.t. the meta-models. In our case, the type-checking is on ATL, which heavily relies on OCL, thus making type-checking more complex. Moreover, we perform an additional analysis directed to find further problems and generate witness models to signal such errors.

Program slicing and path conditions. Program slicing is a technique to detect the parts of a program that affect a given statement [32], [37]. Few works have adapted this idea to model transformations. One exception is [34], which defines dynamic backward slicing for in-place VIATRA2 transformations. Our approach is similar, but we need to tackle the peculiarities of ATL, like the different types of rules and the implicit resolution mechanisms.

While program slices identify the dependencies of a fault, path conditions characterize the input data leading to a faulty statement. In [31], path conditions characterize safety violations, which a constraint solver can verify by generating a witness. This is a set of values for the input variables, needed to reach a certain statement. To our knowledge, the use of path conditions and witnesses in model transformation is novel.

IX. Conclusions and Future Work

We have presented a novel technique for uncovering errors in ATL transformations. The technique uses static analysis and type checking for finding problematic statements, and constraint solving to generate witness models confirming and explaining the errors. We have built a tool and analysed several public ATL transformations, uncovering a surprisingly high number of problems in most of them. This shows that tools like ours are needed in the model transformation community.

We are extending the approach with quick fixes to solve the errors. We also plan to derive transformation pre-conditions, which might be implicit and undocumented, to make explicit the models the transformation is applicable to. Finally, we are performing a wider analysis of existing transformations to classify the most common errors made by developers.

Acknowledgements. This work was supported by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139) and the EU commission with project MONDO (FP7-ICT-2013-10, #611125).

REFERENCES

[1] A. Andoni, D. Daniliuc, and S. Khurshid. Evaluating the "small scope hypothesis". Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.

[2] B. Baudry, J. Dingel, L. Lucio, and H. Vangheluwe, editors. *Proceedings of the 2nd Workshop on the Analysis of Model Transformations*, volume 1077 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[3] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.

[4] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. Model transformations? Transformation models! In *MoDELS'06*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006.

[5] F. Büttner, M. Egea, and J. Cabot. On verifying ATL transformations using 'off-the-shelf' SMT solvers. In *MoDELS'12*, volume 7590 of *LNCS*, pages 432–448. Springer, 2012.

[6] F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL transformations using transformation models and model finders. In *ICFEM*, volume 7635 of *LNCS*, pages 198–213. Springer, 2012.

[7] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[8] J. Cabot, R. Clarisó, and D. Riera. On the verification of uml/ocl class diagrams using constraint programming. *JSS*, 93:1–23, 2014.

[9] J. S. Cuadrado, E. Guerra, and J. de Lara. Reverse engineering of model transformations for reusability. In *ICMT'14*, volume 8568 of *LNCS*. Springer, 2014.

[10] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag, 2006.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[12] F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon. Qualifying input test data for model transformations. *Software and System Modeling*, 8(2):185–203, 2009.

[13] C. A. González and J. Cabot. ATLTest: A white-box test generation approach for atl transformations. In *MoDELS'12*, volume 7590 of *LNCS*, pages 449–464. Springer, 2012.

[14] E. Guerra and M. Soeken. Specification-driven model transformation testing. *Software and System Modeling*, pages 1–22, 2013.

[15] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Software Eng.*, 22(7):484–495, 1996.

[16] J.-M. Jézéquel, O. Barais, and F. Fleurey. Model driven language engineering with kermeta. In *GTTSE'09*, volume 6491 of *LNCS*, pages 201–221. Springer, 2011.

[17] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1):31–39, 2008.

[18] M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive validation of OCL models by integrating SAT solving into USE. In *TOOLS (49)*, volume 6705 of *LNCS*, pages 290–306. Springer, 2011.

[19] J. M. Küster and M. Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In *MoDELS Workshops*, volume 4364 of *LNCS*, pages 193–204. Springer, 2006.

[20] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *ICSE'96*, pages 495–505. IEEE, 1996.

[21] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot. Static analysis of model transformations for effective test generation. In *ISSRE*, pages 291–300. IEEE, 2012.

[22] OMG. OCL 2.3.1 specification. `http://www.omg.org/spec/OCL/2.3.1/`.

[23] QVT. `http://www.omg.org/spec/QVT/`.

[24] L. A. Rahim and J. Whittle. A survey of approaches for verifying model transformations. *Software and System Modeling*, in press, 2013.

[25] A. Rentschler and P. Sterner. Interactive dependency graphs for model transformation analysis. In *Demos/Posters/StudentResearch@MoDELS*, volume 1115 of *CEUR Workshop Proceedings*, pages 36–40. CEUR-WS.org, 2013.

[26] G. M. K. Selim, J. R. Cordy, and J. Dingel. Model transformation testing: The state of the art. In *AMT '12*, pages 21–26. ACM, 2012.

[27] S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In *ICMT'09*, volume 5563 of *LNCS*, pages 148–164. Springer, 2009.

[28] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model pruning. In *MoDELS'09*, volume 5795 of *LNCS*, pages 32–46. Springer, 2009.

[29] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot. Using models of partial knowledge to test model transformations. In *ICMT'12*, volume 7307 of *LNCS*, pages 24–39. Springer, 2012.

[30] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[31] G. Snelting, T. Robschink, and J. Krinke. Efficient path conditions in dependence graphs for software safety analysis. *ACM Trans. Softw. Eng. Methodol.*, 15(4):410–457, 2006.

[32] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[33] Z. Ujhelyi, Á. Horváth, and D. Varró. Static type checking of model transformation programs. *ECEASST*, 38, 2011.

[34] Z. Ujhelyi, Á. Horváth, and D. Varró. Dynamic backward slicing of model transformations. In *ICST*, pages 1–10. IEEE, 2012.

[35] M. van Amstel and M. G. J. van den Brand. Model transformation analysis: Staying ahead of the maintenance nightmare. In *ICMT'11*, volume 6707 of *LNCS*. Springer, 2011.

[36] A. Vieira and F. Ramalho. A static analyzer for model transformations. In *MtATL'11*, 2011.

[37] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.