Static analysis of model transformations

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara

Abstract—Model transformations are central to Model-Driven Engineering (MDE), where they are used to transform models between different languages; to refactor and simulate models; or to generate code from models. Thus, given their prominent role in MDE, practical methods helping in detecting errors in transformations and automate their verification are needed.
In this paper, we present a method for the static analysis of ATL model transformations. The method aims at discovering typing and rule errors, like unresolved bindings, uninitialized features or rule conflicts. It relies on static analysis and type inference, and uses constraint solving to assert whether a source model triggering the execution of a given problematic statement can possibly exist. Our method is supported by a tool that integrates seamlessly with the ATL development environment.
To evaluate the usefulness of our method, we have used it to analyse a public repository of ATL transformations. The high number of errors discovered shows that static analysis of ATL transformations is needed in practice. Moreover, we have measured the precision and recall of the method by considering a synthetic set of transformations obtained by mutation techniques, and comparing with random testing. The experiment shows good overall results in terms of false positives and negatives.

Index Terms—Model-Driven Engineering, Model Transformation, ATL, Static Analysis, Model Finders, Verification and Testing.

1 Introduction

Model transformation is the main enabler of automation in Model-Driven Engineering (MDE) as it allows the manipulation of models [1]. The definition of a transformation is typically used many times, likely in different projects. Hence, transformations need to be error-free to guarantee the reliability of MDE solutions. Higher quality in transformations can be achieved with thorough testing and powerful static analysis methods able to detect potential errors early.

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, like those based on graph transformation [3]. However, most transformation languages used in practice lack a fully formal foundation or a body of theoretical results enabling their formal verification. One of the reasons is that many of them rely on unformalised variants of the Object Constraint Language (OCL) [4], which provides expressiveness at the cost of making transformations difficult to analyse. This is the case of the Atlas Transformation Language (ATL) [5], one of the most widely used model transformation languages in industry and academia.

ATL is dynamically typed. This makes transformations written in ATL 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 in new objects. In addition, rule dependencies are implicitly resolved by the ATL engine, which may lead to subtle problems difficult to identify, like assigning a value obtained from a rule to an incompatible feature. Such ill-typed transformations are accepted by the ATL compiler, and therefore, errors can only be discovered by running the transformation with an input model triggering the execution of the incorrect statement. This testing process is manual, which poses several drawbacks: (i) it delays the detection of errors that could be statically discovered; (ii) the manual creation of input test models is tedious, time-consuming, and probably biased towards "easy" models[1]; (iii) checking that the output model of a transformation is well-formed and conforms to the transformation specification has to be done manually; and (iv) when a test case catches an error, it is often difficult to identify the parts of the transformation causing the error. Even if some works automate the generation of input test models, they mainly propose black-box generation criteria like meta-model [7] or requirements [8] coverage, neglecting the static 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 might be 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

• *J. Sánchez Cuadrado is with the Department of Languages and Systems of the Universidad de Murcia, Spain. E. Guerra and J. de Lara are with the Department of Computer Science, Universidad Autónoma de Madrid, Spain.*
E-mail: jesusc@um.es, {esther.guerra, juan.delara}@uam.es

1. By "easy models" we mean models which focus only on testing a few transformation requirements while neglecting other more subtle ones, do not consider border cases (e.g., collections without objects or with duplicated objects, uninitialized features, out-of-range values, etc.), or disregard types and features not explicitly mentioned in the transformation [6].

that it prevents the error, or because the only input models triggering the error are invalid according to the integrity constraints of the input meta-model.

Hence, as a complementary technique, we use constraint solving to find an input model making 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* describing the features needed in an input model to enforce the execution of the statement. A constraint solver uses this condition and the meta-model to generate a candidate input model. If no model is found, the problem is discarded. We call the generated model a *witness* [9], because it signals a transformation error.

However, even if a witness model confirms an error, it may happen that this model does not belong to the transformation domain. This is so as some transformations are not expected to work with every possible instance of the input meta-model, but just with a subset. In such a case, we provide the developer with a facility to automatically generate an explicit OCL pre-condition that documents the transformation and provides a mechanism to discard unsuitable models before executing the transformation.

Our method is supported by a tool – called ANATLYZER[2] – integrated within the ATL development environment. The tool implements a type checker for ATL transformations, extended with synthesis of OCL path conditions. For witness generation, it relies on the USE Validator [10], a constraint solver for models. We have evaluated three aspects of our approach. First, we have tested its usefulness by analysing a repository of about 100 ATL transformations developed by third parties[3]. ANATLYZER has discovered problems in all of them, covering a wide range of issues. This demonstrates the applicability of our proposal, and shows that static analysis techniques for ATL transformations are needed in practice. Second, we have studied the *precision* (i.e., the fraction of real errors w.r.t. all signalled problems) and *recall* (i.e., the fraction of errors detected w.r.t. all existing errors) of our method. For this purpose, we generated a set of mutants of a synthetic error-free transformation, and then compared the errors statically missed/discovered on the mutants by our tool, and those discovered by random testing. The tool shows good average results in both aspects. Finally, we have evaluated the performance of our tool, which shows good average efficiency.

Altogether, this work makes the following contributions. First, we introduce a novel method to detect errors in ATL transformations and generate pre-conditions for them. The method relies on: (i) a static analyser to detect potential problems based on the typing information, (ii) the construction of OCL path conditions leading to problematic statements, and (iii) the generation of witness models using the path conditions and constraint solving. While we have developed the method for ATL, as we discuss in Section 8.6, it could be adapted for other languages relying on OCL, such as those of the QVT family [11]. The second contribution is a working tool supporting these ideas, and a description of practical issues in its realization, like the handling of different OCL dialects, the continuous evaluation of problems in the background, and the management of error dependencies by an efficient encoding of error paths. The last contribution is a comprehensive evaluation of our tool over a widely used repository of publicly available transformations, and also using transformation mutants and random testing. This approach could be used to evaluate other verification techniques.

This paper is an extended version of [12], where we have broadened the kind of errors we detect, we provide more details for the most challenging error types, we derive transformation pre-conditions, and we analyse rule conflicts. The tool has been significatively improved by giving support to less frequent ATL constructs, a finer alignment with the USE constraint solver, and better user experience that includes the option of running the solver in the background [13]. We have also expanded the evaluation to cover the complete ATL transformation zoo, to measure the precision and recall of our method, and to analyse the tool performance.

The rest of the paper is organized as follows. Section 2 presents an overview of our approach, which is detailed in the following sections: Section 3 introduces a running example, Section 4 explains the static analysis phase and the problems we are able to recognize, Section 5 shows how to generate OCL path conditions, and Section 6 describes the use of model finding to obtain witness models. Section 7 discusses design and implementation issues of our tool. Next, Section 8 evaluates the usefulness, precision, recall and performance of our technique. We also discuss strengths, limitations and applicability beyond ATL. Finally, Section 9 compares with related work, and Section 10 draws conclusions and future work.

2 OVERVIEW

In MDE, models must conform to a meta-model. This is itself a model – typically a class diagram – that declares the admissible object types, relations and features. Meta-models can also include additional well-formedness rules that any valid meta-model instance should satisfy, frequently expressed using the Object Constraint Language (OCL) [4]. OCL is a formal language that belongs to the UML standard and allows expressing queries and constraints on models.

2. http://www.miso.es/tools/anATLyzer.html
3. http://www.eclipse.org/atl/atlTransformations/

Models are manipulated via model transformations. The transformations we are interested in are called *model-to-model* transformations. These translate a source model (e.g., a state machine) into a target model (e.g., a Petri net). The purpose of such transformations may be to take advantage of the analysis mechanisms provided by the target language, to migrate models between language versions, or to provide tool interoperability, among others.

A model transformation is written against source and target meta-models that describe the structure of the models manipulated by the transformation. Transformations can be implemented using dedicated languages called model transformation languages. The different styles of these languages have been analysed in detail by Czarnecki and Helsen [14], but normally, they provide a rule-based mechanism to specify how to transform source models into target models. In this way, a model transformation rule typically has a source pattern describing the source model elements to match, and a target pattern specifying the target elements to create and how to initialize their properties.

Regardless the model transformation language used, writing a transformation is complex due to the variety of model configurations that transformations must handle [15]. As an example, let us consider the conversion of UML class diagrams into KM3[4], a domain-specific language dedicated to meta-model specification [16]. We will focus on the translation of types and properties, and illustrate an error that developers may make. Figures 1 and 2 show relevant excerpts of the UML and KM3 meta-models.

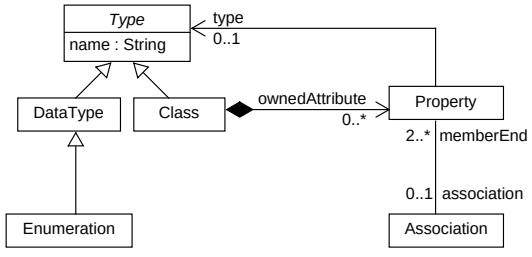

Fig. 1. Excerpt of the UML meta-model (*source*).

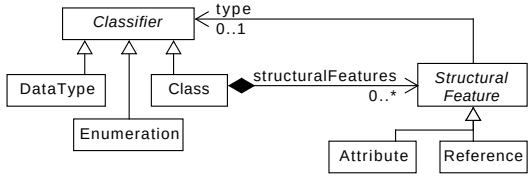

Fig. 2. Excerpt of the KM3 meta-model (*target*).

Figure 3 shows the required transformation rules, using a language-agnostic representation. Each rule is depicted as a box with its name in the upper-left corner. Inside each box, the expression map: source -> target indicates that the rule creates one object of type target from each object of type source. In some rules (property2reference and property2attribute) we use an identifier for the source object. In general, source and target could be complex patterns instead of single objects. Rules can also include a guard to specify conditions that a source object should satisfy to enable the rule application for the object. In the figure, rule guards are preceded by the keyword when and expressed in OCL. Moreover, rules also take care of initializing the features of the created objects (called *bindings* in ATL). This may induce rule dependencies whenever a rule assigns to a feature an object created by another rule. For instance, in the figure, rule property2reference assigns a class to the feature type of the created reference. Hence, there is a dependency from rule property2reference to rule class, as the latter is in charge of creating Class objects in the KM3 model. We represent rule dependencies as arrows.

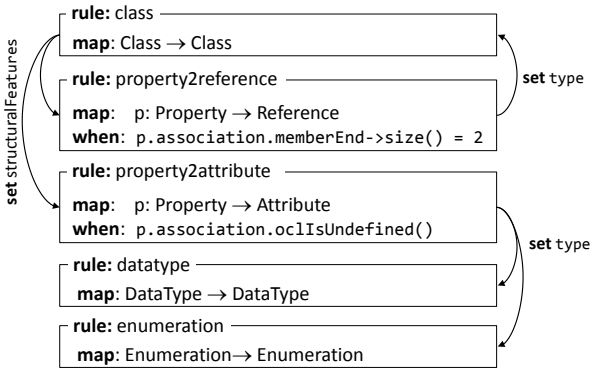

Fig. 3. Rules to map UML types and properties to KM3.

The transformation in Figure 3 contains three rules to map Classes, DataTypes and Enumerations. Another two rules deal with the transformation of UML Properties: The first rule (property2attribute) maps a UML Property to a KM3 Attribute if the property has no attached association, while the second one (property2reference) maps UML Property to KM3 Reference when the property belongs to an association with two member ends. However, the transformation misses a rule to translate n-ary associations (i.e., having more than 2 member ends). If the input model contains n-ary associations, there might be problems ranging from runtime errors to unexpected results, depending on the particular transformation language. Moreover, both rules datatype and enumeration match DataType objects from the source UML model, as class Enumeration inherits from DataType. This might be problematic in some transformation languages, like ATL, which raise a runtime exception if an object gets matched by two rules.

This work aims at enabling the practical analysis and early identification of these and other transformation problems. For this purpose, we use a novel combination of static program analysis and constraint

4. This example is taken from a similar transformation in the ATL Zoo, which tackles the same problem outlined in this section.

solving. Our goal is to uncover a wide range of problems, beyond simple typing problems, in a precise way (i.e., exhibiting a low rate of false positives).

Figure 4 presents the main elements of our technique, and Figure 5 shows its application to the UML to KM3 transformation. While this section introduces the technique in a general setting, the following sections will present the technical details to specialize it for the ATL transformation language. We will focus on ATL since it is a de-facto standard for model transformation. The applicability to other transformation languages is discussed in Section 8.6.

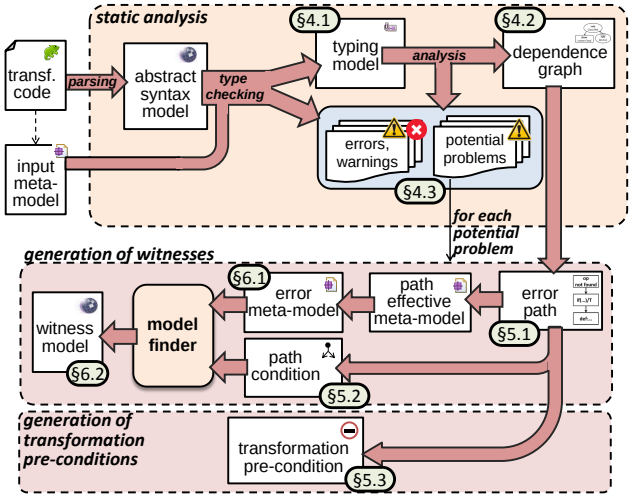

Fig. 4. Overview of our static analysis process. Numbers refer to the paper section explaining the activity.

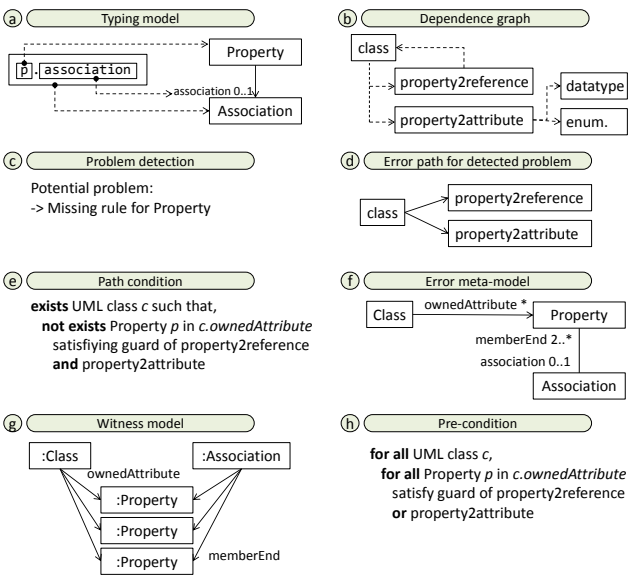

Fig. 5. Static analysis process for rules in Figure 3.

Type checking. The first step in our technique is to parse the transformation code to obtain its abstract syntax model, over which we perform the type checking of the transformation. As a result, the abstract syntax model becomes annotated with typing information. For instance, in Figure 5(a), the typing of the expression p.association creates a link from each subexpression to the corresponding meta-model class or feature. The purpose is to statically guarantee that the transformation is correctly typed with respect to the meta-models, and provide valuable information to the subsequent analysis steps.

Dependence graph creation. Then, we create a transformation dependence graph [17] (TDG) that makes explicit the control and data flow dependencies between the transformation elements. For instance, Figure 5(b) shows the data dependencies between rules as dashed arrows. The TDG is used to improve the scope of the typing analysis and uncover more complex problems, in particular rule-related ones.

Problem detection. The type checking and the analysis of the dependence graph generate a set of detected typing problems. While some of them can be statically guaranteed to be errors (e.g., the use of an incorrect meta-model type), others are classified as *potential problems* because they require further verification. This is the case of the missing rule to transform n-ary associations, since confirming this problem requires analysing the existing rule guards (see Figure 5(c)). To enable this fine-grained analysis we rely on model finding. A model finder is a tool that provides a high-level notation to describe model features, and uses constraint solving to find a model exhibiting such features. Typically, model features are described using structural data models (e.g., class diagrams with OCL constraints [10] or relational logic [18]), and rely on lower-level SAT or SMT solver engines to perform the search (like KodKod [19] or Z3 [20]). The use of a model finder permits generating a witness model that confirms (or falsifies if it does not exist) the potential problem.

Error path computation. For each potential problem, the dependence graph is traversed to build the corresponding *error path*. This is a subgraph (a slice of the transformation) that represents the set of elements that the execution flow must traverse to trigger the error. In the example, the error path would include rules property2reference and property2attribute, but also rule class as this rule must be executed first to reproduce the problem at runtime (see Figure 5(d)). The computation of the error path depends on the particular transformation language, since each language has different ways of handling rule dependencies (e.g., explicit calls in QVTo [11] and implicit rule resolution in ATL [14]).

Path condition computation. From the error path, we generate a so-called *path condition*. This accumulates the constraints that the program input (a source model in our case) must fulfil for the control flow to reach the problematic statement. It is built by conjoining the execution conditions of each node in the error path. For instance, to confirm the potential problem in the example, we need to check whether UML classes can

define properties that are not transformed by any rule. As Figure 5(e) shows, the path condition conjoins the condition to execute the class rule, with the condition that none of the rules for properties are executed. In general, the construction of the path condition depends on the detected kind of error, as we will see in Section 5.2.

Error meta-model computation. From the error path, we generate the *path effective meta-model* which includes just the meta-model elements involved in the error. The path effective meta-model and the input 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 search space than the one provided by the complete meta-model. For instance, the error meta-model in Figure 5(f) provides an important reduction to the size of the complete UML meta-model.

Witness model generation. Finally, we feed the error meta-model and the path condition into a model finder 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. Figure 5(g) shows a witness model for the potential problem. It contains a ternary association, which is a case not handled by the transformation. Hence, the problem is confirmed and reported to the user.

Pre-condition generation. The confirmation of a problem by a witness model may indicate either that there is an error in the transformation, or that the transformation was not designed to accept a given class of models. In the latter case, it is possible to generate a *transformation pre-condition* that rules out the class of models making the transformation execute the problematic statement. Similar to the generation of path conditions, the error path is traversed to generate a constraint that states the shape of the models that are properly handled by the transformation. Figure 5(h) shows the generated pre-condition for the example.

While this section has introduced our approach independently of the transformation language, the following sections detail the realisation of each step for ATL. We will focus on the following aspects: type checking ATL transformations (Section 4.1), construction of dependence graph (Section 4.2), list of errors and warnings detected statically (Section 4.3), construction of error path (Section 5.1), construction of OCL path condition (Section 5.2), generation of transformation pre-conditions (Section 5.3), calculation of error meta-model (Section 6.1) and generation of witness models by model finding (Section 6.2). To illustrate these steps, we first introduce a running example in the next section.

3 Running Example

Transformation languages vary between *strongly typed languages* such as Kermeta [21], where all types are resolved at compile time and the abstract syntax is annotated with type information, and *dynamically typed languages* such as ATL, where type checking is performed at runtime (although the ATL IDE has autocompletion facilities, types are not enforced). In the rest of the paper, we describe in detail our approach to analyse transformations focussing on its application to ATL. It is worth noting that ATL is a challenging scenario in terms of type analysis, as there already exist ill-typed transformations on working systems, at least for the constraints devised (and typically not documented) by their authors. In addition to type-correctness relative to the source and target meta-models, other aspects that can be analysed statically include the applicability and dependency of transformation rules, the detection of unused helpers, and certain performance issues.

As a running example, we use an excerpt of the transformation from the Web Services Description Language (WSDL) [22] to the REWERSE Rule Markup Language (R2ML) [23], available at the ATL Zoo and described in [24]. This transformation automates the reverse engineering of WSDL-based web services into rule-based specifications. Listing 1 contains a slightly simplified version of the transformation, while Figures 6 and 7 show relevant excerpts of the source and target meta-models.

WSDL is a W3C recommendation for describing web services. Its meta-model comprises the abstract description of the services and the messages they exchange. The former includes the service interfaces and operations (in the meta-model, MEP is an enumeration for known message exchange patterns), while concrete messages are described using XML Schema.

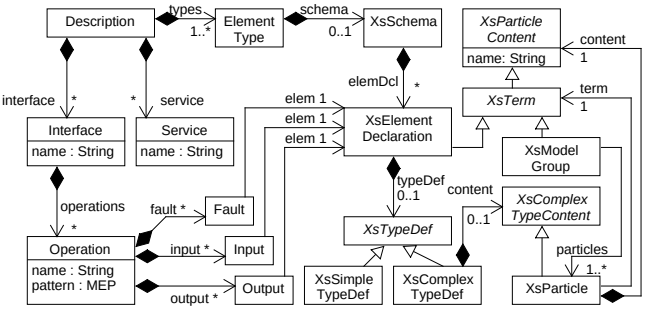

Fig. 6. Excerpt of the WSDL meta-model (*source*).

R2ML is an XML-based rule neutral format that allows exchanging rules between systems, or their integration using ontologies. It supports different kinds of rules. In particular, this transformation considers ReactionRules (a form of Event-Condition-Action rules [25]) to represent how message events are handled. The information managed by the rules is represented as

a vocabulary, which allows the definition of entities with attributes (Class and Attribute).

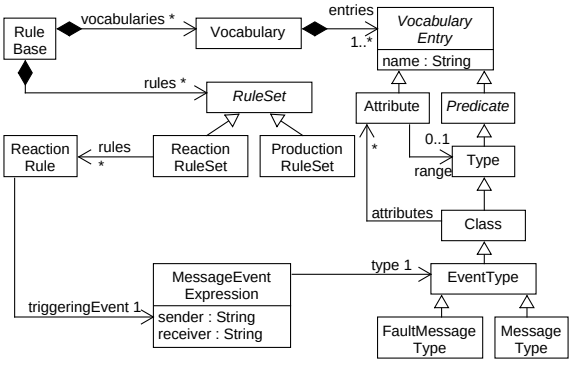

Fig. 7. Excerpt of the R2ML meta-model (*target*).

We have chosen this transformation because it exhibits a wide range of non-trivial problems that are useful for illustration. Moreover, the complexity of the WSDL meta-model reflects the typical scenario faced by transformation developers, who may tend to make mistakes when there is no automatic type checking.

```
1  module WSDL2R2ML;
2  create OUT : R2ML from IN : WSDL;
3
4  rule RuleBase {
5    from i : WSDL!Description
6    to o : R2ML!RuleBase (
7      rules ← Sequence { i.service, i.interface },
8      vocabularies ← i.types
9    )
10 }
11
12 rule Vocabulary {
13   from i : WSDL!ElementType
14   to o : R2ML!Vocabulary (
15     entries ← i.schema.elemDcl
16   )
17 }
18
19 rule MessageType {
20   from i : WSDL!XsElementDeclaration (
21     not i.typeDef.oclIsTypeOf(WSDL!XsSimpleTypeDef) and i.isInOut()
22   )
23   to o : R2ML!MessageType ( ... )
24 }
25
26 rule FaultMessageType {
27   from i : WSDL!XsElementDeclaration (
28     not i.typeDef.oclIsTypeOf(WSDL!XsSimpleTypeDef) and i.isFault()
29   )
30   to o : R2ML!FaultMessageType ( ... )
31 }
32
33 rule ClassR {
34   from i : WSDL!XsElementDeclaration (
35     not i.typeDef.oclIsTypeOf(WSDL!XsSimpleTypeDef) and
36     not (i.isInOut() and i.isFault())
37   )
38   to o : R2ML!Class (
39     attributes ← if i.typeDef.oclIsTypeOf(WSDL!XsComplexTypeDef) then
40         let elems : OclAny = i.typeDef.content.term.particles→
41           select(p | p.content.oclIsKindOf(WSDL!XsElementDeclaration))[5]
42         in elems→collect(e | thisModule.AttributeR(e))
43       else
44         OclUndefined
45       endif,
46       ...
47   )
48 }
49
50 lazy rule AttributeR {
51   from i : WSDL!XsParticle
52   to o : R2ML!Attribute (
53     range ← thisModule.Datatype(i.content.typeDef),
54     name ← i.content.name
55   )
56 }
57
58 rule ReactionRuleSet {
59   from i : WSDL!Interface
60   to o : R2ML!ReactionRuleSet (
61     rules ← i.operations.asSequence()→
62       collect(e| if thisModule.isInOutPattern(e.pattern) then
63         Sequence {thisModule.RrRight(e), thisModule.RrWrong(e)}
64       else
65         thisModule.RrRight(e)
66       endif)
67   )
68 }
69
70 lazy rule RrRight {
71   from i : WSDL!Operation (i.oclIsTypeOf(WSDL!Operation))
72   to o : R2ML!ReactionRule (
73     triggeringEvent ← i.input
74   )
75 }
76
77 rule TriggeringEvent {
78   from i : WSDL!Input ( i.oclIsTypeOf(WSDL!Input) )
79   to o : R2ML!MessageEventExpression (
80     sender ← 'to_be_defined',
81     type ← i.elem  -- Could be resolved by ClassR
82   )
83 }
84
85 helper context WSDL!XsElementDeclaration def: isInOut() : Boolean =
86   WSDL!Operation.allInstances()→exists(o |
87     o.input→exists(e | e.elem = self) or o.output→exists(e | e.elem = self));
88
89 helper context WSDL!XsElementDeclaration def: isFault() : Boolean =
90   WSDL!Operation.allInstances()→exists(o|
91     o.fault→exists(e | e.elem = self));
92
93 helper def: isInOutPattern(value : String) : Boolean[6] =
94   value = #inout;
95
96 ... -- we omit some rules for brevity
```

Listing 1. Excerpt of ATL transformation for the running example. Problems are underlined according to its category: confirmed problem, discarded problem and warning

ATL transformations consist of rules and OCL helpers. A rule defines a source pattern identifying a configuration of source objects – normally one – for which it generates one or more target objects. The most common kinds of ATL rules are *matched* rules and *lazy* rules[7]. A matched rule is implicitly executed once for each occurrence of its source pattern. This execution has as a side effect the creation of a trace link that relates the matched source objects with the target objects created by the rule. The ATL engine keeps this internal execution trace, which is later used to resolve references in *bindings* (see below). In contrast, a lazy rule is only executed when it is explicitly invoked, and does not record trace links. For instance, the Vocabulary matched rule (line 12) transforms every ElementType object into an object of type Vocabulary. In contrast, the AttributeR lazy rule (line 50) will be executed upon its explicit invocation through

5. The filter has been added for illustrative purposes.

6. The helper has been shortened for illustrative purposes.

7. We omit entrypoint and called rules, handled similarly.

the statement thisModule.AttributeR(e) (line 42), and hence, its execution depends on the execution of the caller rule ClassR. ATL also supports *unique lazy* rules (not present in the example). These are explicitly invoked like lazy rules, but they record trace links so that they behave as memoized functions (i.e., subsequent invocations over the same source object do not create a new target object, but they return the object created in the first invocation).

Objects in the source patterns of matched rules may have filters. These are OCL conditions that must be satisfied for the rule to be applicable. For example, rule MessageType has a filter for object i in lines 20–22.

The features of the created target objects are initialized using *bindings* with the syntax *feature* ← *OclExpr*. If *OclExpr* is a primitive value, it is assigned to *feature*. If *OclExpr* is an object or a collection of objects, each such source object is looked up in the transformation execution trace to retrieve the target object in which it was transformed to. This target object is then assigned to *feature*. This is called *binding resolution*. In the example, the binding vocabularies ← i.types in line 8 will be resolved by rule Vocabulary in line 12, because the type of i.types is ElementType. At runtime, for each object in i.types, the ATL engine looks up a trace pointing to the object, and assigns the corresponding target object (which will have type Vocabulary) to the vocabularies feature. Here, a common source of errors is to have no rules able to resolve the binding (i.e., there is no trace link for a given source object), or a rule creating objects of incompatible type with *feature* (i.e., the trace records a target object that is not compatible with the feature to which it is assigned).

Finally, *helper*s can be seen as operations or derived features attached to a given type (called context helpers). Helpers that are not attached to a type act as global functions (called module helpers). The listing contains a module helper in line 93 and two context helpers in lines 85 and 89.

The ATL compiler processes the transformation in Listing 1 without reporting any problem. However, the transformation has several issues that may cause runtime errors or produce incorrect target models. In particular, we observe the following problems:

1) **Unresolved binding** (line 7). There is no rule to resolve Service objects, needed due to the expression i.service. While it is common that parts of the source model are irrelevant for a transformation (i.e., not handled by any rule), in this case, the use of the expression i.service suggests that a rule handling objects of type Service is missing.
2) **Feature access over possibly undefined receptor** (lines 15 and 40). The feature schema in ElementType is optional, but it is accessed without any checking in line 15. This will cause a runtime error if the source model contains ElementType objects without a schema. Similarly, feature typeDef is not mandatory in XsElementDeclaration, and feature content is not compulsory in XsComplexTypeDef. Therefore, accessing them may cause a runtime error in line 40.
3) **Feature found in subtype** (line 40). The expression i.typeDef.content.term.particles accesses features defined in subtypes. For example, feature particles is not defined in XsTerm, but in XsModelGroup. Thus, a model with a term holding an object of a different subtype of XsTerm, like XsElementDeclaration, will cause a "feature not found" runtime error.
4) **Expected access to collection (→) but dot notation found** (line 61). This is a style warning. ATL does not fail to correctly execute the expression, but it is not compliant with the OCL standard [4].
5) **Wrong argument type** (line 62). The helper isInOutPattern declares a String formal argument (line 93), but the type of the actual argument is WSDL!MEP (line 62).
6) **Lazy rule with filter** (line 71). Syntactically, it is possible to define a filter in a lazy rule, however, the filter will not be checked at runtime. As this may confound the developer, it is marked as a warning.
7) **No binding for compulsory target feature** (line 79). Feature receiver is compulsory in type MessageEventExpression, but no binding initializes its value. Thus, any generated target model containing an object of this type will be incorrect since it will not conform to the target meta-model.
8) **Binding resolved by rule with invalid target** (line 81). This binding can get resolved by rule ClassR (line 33). However, this rule creates an object of type Class, which is not compatible with the type of the feature type (EventType). Thus, resolving the binding by this rule will produce an ill-typed target model. In this case, we need to check whether this resolution is possible by generating a witness model that enforces it.

In the following sections, we explain how we detect these errors by gathering type information and performing static analysis, and how we generate witness models to confirm or discard the detected errors.

4 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 TDG.

4.1 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 variable elems in line 40 of Listing 1 declares the type OclAny, while a more precise type would be Sequence(XsParticle). The call elems→collect(...)

in line 42 works at runtime, but a naive type checker that takes OclAny as the type for elems would signal an error. The ATL compiler does not report this kind of 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. Moreover, the type checking phase takes into account the peculiarities of ATL. For example, the expression i.typeDef.content in line 40 is correct because the check in the previous line ensures that the type of i.typeDef is XsComplexTypeDef, which defines the feature content. Although ATL does not provide a *downcasting* operator (e.g., oclAsType), implicit castings following this idiom are common, and hence we provide support for this scenario.

Type checking is performed in two passes. First, the algorithm annotates the variable declarations, rule pattern types and helpers with the types they explicitly declare. Then, a bottom-up traversal of the Abstract Syntax Tree (AST) is performed, propagating types, annotating each node in the AST, and reporting errors and warnings. Currently, we do not support type inference for recursive helpers, but we simply use their declared type. Similarly, we do not perform type inference of formal parameters. For instance, the expression value = #inout in line 94 is reported as an error because the declared type of value is String; however, the only call to this helper, in line 62, will not cause a runtime error because the problem is in the type declared by the formal parameter of the helper, which should be WSDL!MEP. Although we could use techniques like the Cartesian Product Algorithm (CPA) [26] to provide a more accurate typing in these cases, we chose not to due to performance reasons, and because we believe it is important to report misleading formal parameters.

Figure 8 shows the types used to annotate the AST nodes. These include the typical OCL types such as primitive types and collection types. Metaclass refers to a class defined in the source or target meta-model. ThisModule refers to the transformation itself. TypeError is a marker indicating that a node in the AST is problematic and cannot be given a type. Reflective allows queries to an object's metaclass at runtime via oclType(). In such a case, the typing of basic attributes (e.g., name) is precise, but the typing of reflective operations like newInstance and allInstances may not be reliably computed. Undefined values (i.e., *null*) are represented with Undefined. The type EmptyCollection is used for expressions like Sequence {}, to indicate that the type of the collection's elements is unknown until analysing any subsequent operation call such as including. We also support Map and Tuple types (not shown in the figure) in a similar way as collections.

We use the type Union to keep track of the multiple types that an expression may potentially yield. For example, the expression Sequence{i.service, i.interface}

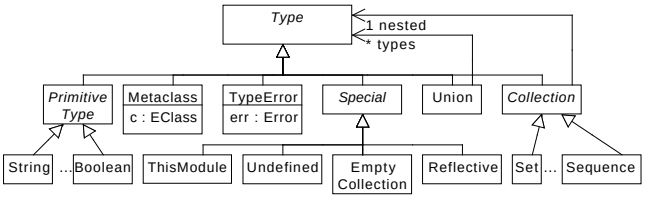

Fig. 8. Types used to annotate the AST.

receives the type Sequence(Union(Service, Interface)). This allows reasoning about the operations that can be performed, or the features that can be accessed, on the result of the expression. In general, no error is reported if an invoked operation or feature is available to all types in the union, which is coherent with the duck typing that ATL implements. Hence, the expression Sequence{i.service, i.interface}→flatten()→collect(o | o.name) is deemed valid because both Service and Interface have a feature name. Instead, accessing to feature operations would be reported as an error because Service lacks this feature.

Altogether, the type checking phase annotates the nodes of the transformation AST. This allows the identification of certain typing errors and warnings. Table 1 shows a summary of the most important errors we are able to detect (at this point, the errors in phase *typing*). We will further explain this table in Section 4.3, while in Section 5, we will show that some type errors detected statically still need to be confirmed by finding a witness model.

4.2 The transformation dependence graph

The type checking phase enriches the abstract syntax model of the transformation with extra information such as the type of the nodes and the control and data flow of the transformation. This model, called *transformation dependence graph* (TDG), is analysed in a second stage to uncover further potential problems. Implementation-wise, the TDG is an instance of an extended version of the ATL meta-model, which is partly shown in Figure 9.

In the meta-model, each OclExpression refers to the type assigned by the type checker via the inferredType reference. Additionally, they hold an extra type, noCastType, which is the type of the expression if no implicit casting is performed. The control and data flow of OCL expressions are represented in the abstract syntax of ATL following the containment relationships of expressions. This suffices also for loops, which in OCL 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 (represented by 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 more challenging. We use a similar approach to [27], where

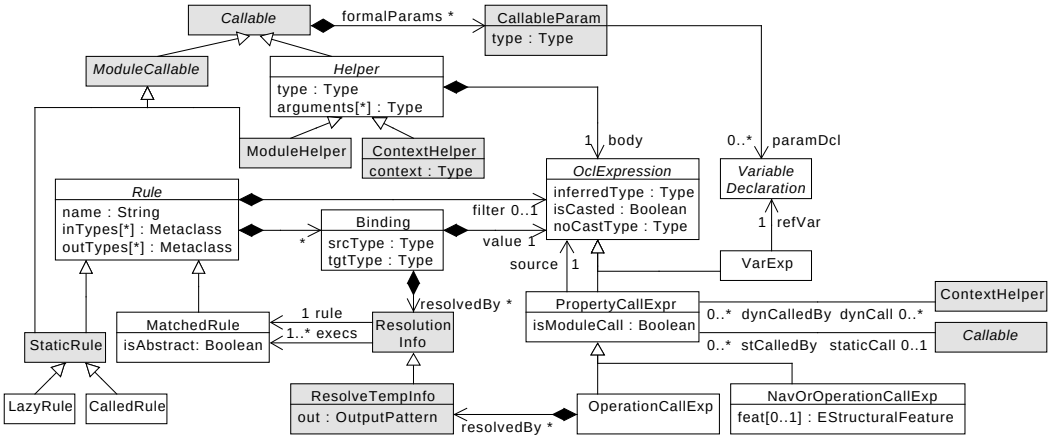

Fig. 9. Excerpt of the extended ATL meta-model. Added classes are highlighted in grey. All shown features have been added to the original meta-model, except Rule.name, Binding.value, VarExp.refVar and MatchedRule.isAbstract.

we attach to each call expression the context helpers that may resolve it (reference dynCall in Figure 9). Similarly, we attach to each binding the matched rules that may resolve it (reference resolvedBy), computing both the resolving rule (reference rule) and the list of rules involved in the resolution (reference execs). The latter list is useful to reason about rule inheritance, since all inherited rules are executed in order. We also give support to resolveTemp operations (an explicit form of rule resolution) via ResolveTempInfo.

Lazy and called rules are statically resolved because they are invoked explicitly and behave like module helpers that return a target element (see class ModuleCallable and reference PropertyCallExpr.staticCall).

For feature access, which is represented by class NavOrOperationCallExp, we keep a reference to the navigated feature. This is required, for example, to compute the meta-model footprint of the transformation.

Example. The TDG excerpt in Figure 10 shows the binding type ← i.elem (label 1), which can be resolved by rules ClassR, MessageType and FaultMessageType (labels 2, 4 and 5). This is indicated by the link resolvedBy in the TDG. On the other hand, the TDG includes a staticCall link which corresponds to the explicit call to the lazy rule AttributeR (label 6) from rule ClassR (label 3). The execution of the lazy rule yields the execution of the problematic expression i.content.typeDef (line 53).

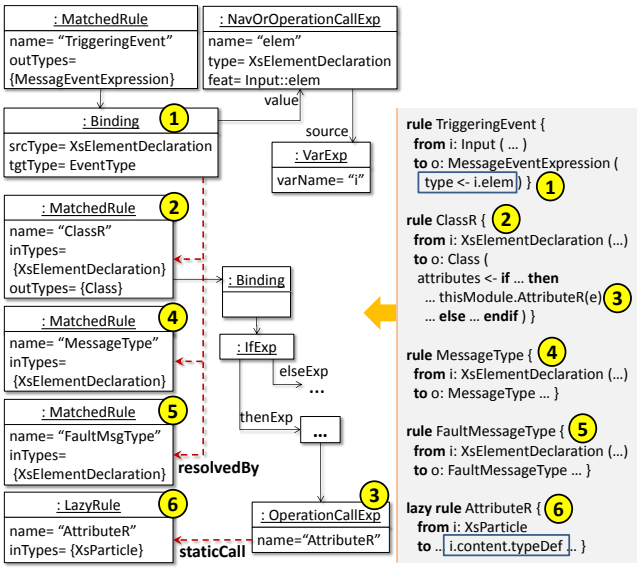

Fig. 10. Excerpt of TDG with the elements involved in errors "Binding resolved by rule with invalid target" (line 81) and "Feature found in subtype" (line 53).

4.3 Analysis of problems

Table 1 summarizes the problems we detect statically. The second column shows the phase of the analysis where the problem is detected, either in the type checking (Section 4.1) or in the analysis of the TDG (Section 4.2). 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, for which we distinguish three kinds of errors and three kinds of warnings. Errors of type *error-load* occur in the loading phase of the transformation execution. Although these errors make the transformation fail regardless the particular input model, the ATL compiler does not detect them statically. *Runtime errors* are the most common. These errors occur during the transformation execution, likely when the transformation is fed a model with certain features that trigger the execution of a problematic statement. Errors of type *error-target* do not make the transformation crash, but they yield an incorrect target model. Regarding warnings, those of type *warning-behaviour* are smells indicating that the transformation may not behave as expected, while types *warning-perf* and *warning-style* signal performance and style issues respectively.

In another dimension, we classify problems in five categories, corresponding to the five blocks in Table 1.

TABLE 1
Problems detected statically in ATL transformations.

Description	Phase	Precision	Severity
Typing (with respect to source/target meta-model and helper definitions)			
Invalid meta-model name	typing	static	error-load
Invalid meta-class name	typing	static	error-load
Invalid enum literal	typing	static	error-load
Feature not found	typing	static	runtime-error
Feature not found in union type	typing	static	runtime-error
Feature found in subtype	typing	sometimes-solver	runtime-error
Operation not found	typing	static	runtime-error
Operation found in subtype	typing	sometimes-solver	runtime-error
Attribute not found in thisModule	typing	static	runtime-error
Operation not found in thisModule	typing	static	runtime-error
Object without container	typing	static	runtime-error
Incoherent variable declaration	typing	static	warning-style
Incoherent helper return type	typing	static	warning-style
Invalid number of actual parameters	typing	static	runtime-error
Invalid actual parameter type	typing	static	warning-behaviour
Navigation			
Collection operation not found	typing	static	runtime-error
Collection operation over no collection (“→” vs. “.”)	typing	static	warning-style
Operation over collection type (“.” vs. “→”)	typing	static	warning-style
Feature access in collection	typing	static	runtime-error
Iterator over empty collection	typing	static	warning-behaviour
Feature access over possibly undefined receptor	typing	sometimes-solver	runtime-error
Feature access over possibly undefined receptor via empty collection	typing	always-solver	runtime-error
Flatten over non-nested collection	typing	static	warning-perf
Foreach statement expected collection	typing	static	runtime-error
Wrong iterator body type	typing	static	runtime-error
Change select-first for any	typing	static	warning-perf
Iterator over no collection type	typing	static	runtime-error
Invalid argument for built-in function	typing	static	runtime-error
Invalid operand	typing	static	runtime-error
Invalid operator	typing	static	runtime-error
Transformation integrity constraints			
Invalid rule inheritance	typing	static	runtime-error
Matched rule without output pattern	typing	static	runtime-error
Matched rule with non-boolean filter	typing	static	runtime-error
Abstract class instantiation	typing	static	runtime-error
Read access to target model	typing	static	warning-behaviour
Lazy rule with filter	typing	static	warning-behaviour
Target meta-model conformance			
No binding for compulsory target feature	analysis	static	error-target
Binding resolved by rule with invalid target	analysis	sometimes-solver	error-target
Collection assigned to mono-valued binding	analysis	static	error-target
Incompatible primitive value for primitive binding	analysis	static	error-target
Model element assigned to primitive binding	analysis	static	error-target
Primitive value assigned to object binding	analysis	static	error-target
Invalid assignment in imperative binding	typing	static	runtime-error
Transformation rules			
No rule to resolve binding	analysis	static	warning-behaviour
Binding possibly unresolved	analysis	always-solver	warning-behaviour
No rule to resolve a resolveTemp operation	typing	static	warning-behaviour
ResolveTemp possibly unresolved	analysis	always-solver	warning-behaviour
Undefined output pattern in resolveTemp operation	typing	static	runtime-error
Rule conflict	analysis (separate)	sometimes-solver	runtime-error

Typing problems 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 lacks the feature, raises an error.

OCL *navigation* expressions, which are pervasive in ATL transformations, can also contain errors. These include misuses of built-in OCL operations, such as invalid collection operations and operators. There can also be style issues, like mistaking the dot (for objects) and arrow (for collections) operators, as well as performance issues, like flattening a non-nested collection. Here, other ATL/OCL performance issues could be considered [28].

Some *integrity constraints* regarding the semantics of ATL are not enforced by the ATL IDE, and therefore can be violated by transformation implementations. Even if these violations will not produce runtime errors, they may cause unexpected results. For instance, since the target model of an ATL transformation should be write-only, 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.

Another range of problems concerns the *conformance to the target meta-model* of the output models produced by a transformation. These problems pass unnoticed during the transformation execution, becoming apparent only 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 detect some problems of this type. For instance, we report as an error any binding of feature with incompatible type. A target compulsory feature (i.e., with positive lower cardinality and no default value) which is left unbound is also an error, as this will produce an incorrect target model. Bindings resolved by rules with an invalid target type are a potential problem, which may require confirmation by finding a witness model.

Finally, problems due to dependencies between *transformation rules* include missing rules for the types used in bindings and conflicting rules that define non-exclusive guards[8].

Next, we describe in detail four of the most important and challenging types of errors.

4.3.1 Unresolved bindings

ATL enables loose coupling of matched rules by avoiding explicit rule calls. For this purpose, bindings are implicitly resolved when their right part contains source objects matched by other rules. This resolution returns the target objects in which the source objects were transformed. In case there is no rule to resolve a given source object, the ATL engine outputs a message indicating that the types are not compatible. This is a performance bottleneck for large models, and moreover, the message is useless as the line with the problematic statement is not reported. Although this kind of problem does not produce a runtime error, it is a smell of unexpected behaviour, as certain object configurations are not being handled by any rule.

The running example contains three errors of this kind. The binding in line 7 is statically guaranteed to be an error, since no rule has type Service in its from part ("No rule to resolve binding" in Table 1). In contrast, the binding in line 15 is more complex to analyse, as it can be resolved by three different rules with non-trivial filters (rules MessageType, FaultMessageType and ClassR in lines 19, 26 and 33). Hence, we mark this binding as potentially problematic, and resort to a model finder to determine if the right part of the binding can contain objects not covered by these three rules. In this case, the finder confirms that the binding will be unresolved for messages defined by XsSimpleTypeDef objects ("Binding possibly unresolved"). Finally, the binding in line 73 is also a potential problem because the resolving rule (TriggeringEvent in line 77) has a filter. However, in this case, the model finder does not find a witness model and therefore the problem is discarded.

4.3.2 Binding resolved by rule with invalid target

The binding resolution mechanism of ATL does not take into account the target type of the resolving rule. Hence, a binding might be resolved by a rule whose target type is not compatible with the type of the feature being assigned to. This will produce a target model that is not conformant to the target meta-model (i.e., there will be slots holding objects of wrong type)[9]. This is a subtle but important problem that should be detected statically and as early as possible to avoid having to inspect the conformance of each generated target model.

The binding in line 81 of the running example has a problem of this kind, which is graphically depicted in Figure 10. In particular, the binding type ← i.elem (label 1) could be resolved by rule ClassR (label 2); however, the output type of this rule is Class, which is not compatible with the type of the feature which is EventType (label 1). Hence, when the binding is resolved by rule ClassR, it produces an invalid target model because some MessageEventExpression objects will have a Class object in the type slot, which is incorrect according to the target meta-model.

4.3.3 Feature found in subtype

This is a variant of the *feature not found* problem. It is signalled when there is an expression obj.feature, the inferred type of obj is T, and feature is not declared in T but in one or more of its subtypes.

Detecting this error accurately depends on a proper analysis of the oclIsKindOf and oclIsTypeOf operations[10], as ATL does not support explicit downcastings (i.e., the oclAsType operation is not supported). For this reason, we keep track of any usage of oclIsKindOf and oclIsTypeOf in conditional expressions, rule filters and select iterators. Moreover, we build a symbol table per scope that relates expressions to checked types. When accessing a feature or operation, the receptor expression is sought in the symbol table, and if it is found, the type ensured by oclIsKindOf/oclIsTypeOf is used. To deal with logical operators and the else branch of if statements, we use the Union type. For instance, the expression i.typeDef in line 35 is initially given type XsSimpleTypeDef in the symbol table, but the presence of the not operator changes this type to the union of the sibling subtypes, that is, to Union(XsComplexTypeDef) which is simplified to just XsComplexTypeDef as there is only one subtype. This mechanism allows discarding statically some potential problems without recurring to the model finder.

The running example includes a problem of this kind in line 40. As discussed previously, the expression i.typeDef.content.term.particles is invalid because feature particles is not defined in XsTerm, but in its subtype XsModelGroup. On the contrary, the subexpression i.typeDef is valid due to the enclosing oclIsTypeOf check

8. In ATL, a source object can only be processed by one rule, and a runtime error occurs if the object is matched by a second rule.

9. The behaviour described here has been tested for ATL 3.X; subsequent versions of ATL may behave differently.

10. obj.oclIsKindOf(C) returns true if the object obj is an instance of C or any of its subclasses; obj.oclIsTypeOf(C) returns true if the object obj is an instance of C.

performed in the previous line. In the same expression, feature term is not defined by XsComplexTypeContent, but by its subtype XsParticle; however, this will not cause a runtime error because the only subtype of XsComplexTypeContent is XsParticle.

Line 53 shows another example. If the analyser reports an error in expression i.content.typeDef, it will be a false positive because the transformation logic guarantees that the type of i.content is XsElementDeclaration (check done in line 41). As a heuristic, we confirm the problem by model finding whenever the path condition to the problematic statement includes oclIsKindOf/oclIsTypeOf checks (see Section 5). This may still yield false positives if the developer uses expressions like p.content.oclType().name = 'XsElementDeclaration' to check types. However, this is seldom used in practice.

4.3.4 Rule conflict

If several rules match the same source objects, they become in conflict and will cause a runtime error. Hence, if several rules have compatible input types, they should define exclusive filters to ensure that the same source object is only matched by one of the rules. Detecting rule conflicts may be difficult, especially if rule filters are complex or meta-models have deep inheritance hierarchies.

Two rules are potentially in conflict if their input type is the same, one is subtype of the other (in which case we identify the less general type as problematic), or they have at least one subtype in common. In such cases, we use the model finder to generate a witness model that contains an object of the problematic type and fulfils the filter of both rules. If such a model exists, the conflict becomes confirmed, as both rules would be applicable to the same object.

In the running example, rules MessageType and FaultMessageType are in conflict because they have the same input type and the rule filters are non-exclusive. Thus, an XsElementDeclaration object that is pointed by both an Operation input (or output) and an Operation fault will be matched by both rules, causing a runtime error. The solution to this problem can be either fixing the rule filters or adding a transformation pre-condition.

5 Building the Path Condition

As we have seen, some of the problems detected during 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 1 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 causing the execution of 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 (i.e., cardinality of associations, compositions and OCL constraints).

The next subsections explain these steps in detail.

5.1 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. Table 2 shows the types of nodes that can appear in the error path. Additionally, there is a problem node for each type of problem, which gathers specific error conditions (see Section 5.2).

TABLE 2
Types of nodes in the error path.

Node type	Description
Matched rule	Execution of a matched rule, which typically starts the execution flow.
Abstract matched rule	Path branching that occurs when the engine selects one of the inheriting subrules according to the dynamic type of the source object.
Loop	Iteration over a collection using an OCL iterator.
If	Condition and branch that leads to the error.
Let	Variable definition and its scope.
Call	Invocation to a helper or lazy rule.
Helper invocation	Set of paths starting from a helper.
Static rule invocation	Set of paths starting from a lazy or called rule.
Subexpr	Valid part of a problematic expression.

Listing 1 outlines the algorithm that computes the error path to a problem detected statically. Its entry point is the function computePath, which receives the problem object created by the static analyser and the element of the abstract syntax model that exhibits the problem. It navigates the control flow backwards, branching the path if several flows are possible. This is done in the function pathToControlFlow, defined in line 6. This function traverses the containment tree of the AST to find a control flow statement, for which it creates a new node in the error path. For this purpose, the current path node is passed to the appropriate pathTo function, which is in charge of creating the new path node as a child of the current node, and then calls pathToControlFlow recursively for the created node. For simplicity, we only show the handling of conditional statements and helpers.

Function pathToIfExp handles conditional expressions (line 14). It checks whether the current path comes from the *if* or *else* branches to mark the created path node accordingly, and call pathToControlFlow recursively to compute the children of the created node (lines 25–27). Alternatively, if the path was flowing from the condition, we skip the conditional expression.

```
   Input: object identifying the detected problem (problem)
   Input: AST element of the faulty statement (elem)
   Output: root node of the error path (node)
1  def computePath (problem, elem):
2      node = createProblemSpecificNode(problem, elem)
3      pathToControlFlow(elem, node)
4      return node
5  end
6  def pathToControlFlow (elem, node):
7      parent = getParentControlFlowStmt(elem)
8      switch parent do
9          case IfExp do pathToIfExp(parent, elem, node);
10         case Helper do pathToHelper(parent, node);
11         // ... similarly StaticRule, MatchedRule, LoopExp, Let
12     end
13 end
14 def pathToIfExp (ifExp, child, node):
15     branch = nil
16     if ifExp.thenExpr = child then
17         branch = true
18     else if ifExp.elseExpr = child then
19         branch = false
20     else
21         // the path comes from the condition
22         pathToControlFlow(ifExp, node)
23         return
24     end
25     ifNode = new IfNode(ifExp, branch)
26     node.addChildren(ifNode)
27     pathToControlFlow(ifExp, ifNode)
28 end
29 def pathToHelper (helper, node):
30     hNode = new HelperInvocationNode(helper)
31     node.addChildren(hNode)
32     switch helper do
33         case StaticHelper do
34             foreach pcall in helper.stCalledBy do
35                 pathToCall(pcall, hNode)
36             end
37         end
38         case ContextHelper do
39             foreach pcall in helper.dynCalledBy do
40                 pathToCall(pcall, hNode)
41             end
42         end
43     end
44 end
45 def pathToCall (pcall, node):
46     cNode = new CallExpNode(pcall)
47     node.addChildren(cNode)
48     pathToControlFlow(pcall, cNode)
49 end
```

Algorithm 1: Computation of the error path

Function pathToHelper deals with helpers (line 29). In this case, the path is split by creating a new call node for each invocation to the helper (pathToCall, line 45). Interestingly, the handling of static and context helpers is different, as static helpers have explicit call sites, while context helpers require considering all possible polymorphic calls. This information is provided by the TDG, via the StaticHelper and ContextHelper classes, and the stCalledBy and dynCalledBy references. The other nodes in the graph are handled similarly.

Example. Figure 11 shows the error path for the problematic expression i.content.typeDef (line 53 in the running example). We start by creating a problem-specific node, *feature found in subtype* in this case (label 1). This kind of error distinguishes the invalid part of the expression (i.e., the call to typeDef) from the correct part (i.e., the receptor expression i.content). The correct part is added to a *Subexpr* node, to ensure that it will be considered by the model finder (label 2). To exercise this expression, its owning lazy rule AttributeR must be explicitly invoked; therefore, we look in the TDG for any call expression that invokes the rule. This information is stored in the TDG's reference stCalledBy. Since the rule can be called from several sites, we create a *Lazy rule invocation* node that aggregates all possible rule calls (label 3). In the figure, we only depict the call thisModule.AttributeR(e) (label 4). This call is performed within the elems→collect(...) expression, and hence, we create a *Loop* node to represent the iterated collection (label 5). For convenience, we create control flow nodes for *Let* assignments (label 6). Next, since expressions in labels 3–6 are within the true branch of a conditional expression, we create an *If* node indicating the branch from which the path is flowing (label 7). Finally, the execution starts in the ClassR matched rule (label 8).

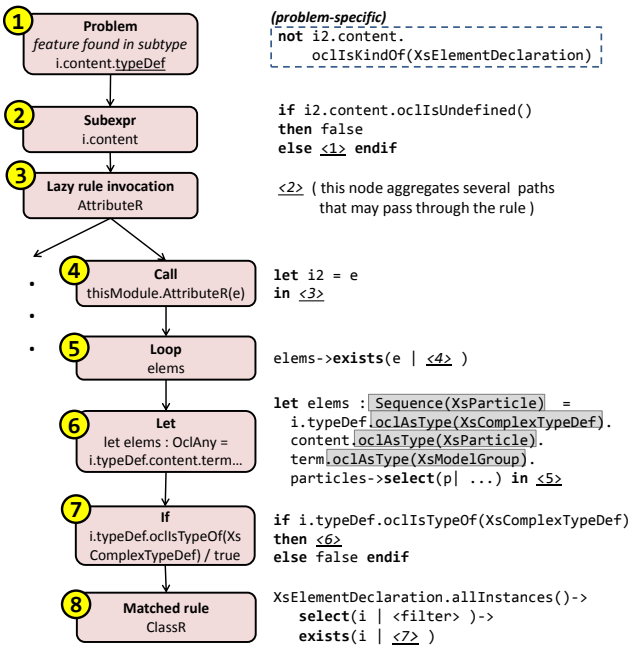

Fig. 11. Error path and OCL path condition for the problem in line 53 of the running example.

It is worth noting that our algorithm builds error paths which are trees with a *problem-specific node* as root, and *execution nodes* as leaves (i.e., typically nodes corresponding to matched rules). This facilitates the traversal of the error path, e.g., to generate OCL path conditions, as we explain in the next subsection.

5.2 Building the OCL path condition

Given the error path, we derive an OCL expression describing the models that cause the execution of the problematic statement. The strategy to construct 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. As noted before, the error path is a tree where every node has a single parent. If an error path has several leaf nodes, then, we take the disjunction of the conditions generated from each leaf. Another consideration is that there may be error paths that do not lead to an execution node, for instance, in the case of dead rules or helpers. We keep track of these cases and do not generate an incorrect path condition for them.

Table 3 shows the OCL fragments generated from each path node type. The place where the code produced by the parent node is inserted is marked as *depNode*.

The OCL fragments generated from each node of the error path do not depend on the error type, except for the root node, from which a problem-specific OCL condition is generated. Table 4 shows the conditions generated for problems with precision *sometimes-solver* and *always-solver*. We also support the generation of witness models for the rest of problems, but the extra condition is simply true as it is enough to reach the problematic statement. A problem type is considered *always-solver* if it always requires being confirmed using a model finder, whereas it is *sometimes-solver* if there are situations where the problem can be confirmed or discarded without using the model finder. For instance, a rule conflict problem is *sometimes-solver* because we can statically confirm that two rules with the same source type are in conflict if the rules have no guards.

Example. Figure 11 shows the OCL path condition built starting from the *Matched rule* node (label 8). The generated OCL fragment enforces the existence of objects that can be matched by the rule, that is, XsElementDeclaration objects that fulfil the rule filter. The rule filter invokes two helpers (isInOut and isFault) which we attach to the path condition to enable their use by the model finder. Next, the condition for the *If* node (label 7) is nested within the previous fragment, and the constraint imposed by the subsequent *Let* node (label 6) is inlined in the true branch of the if. The OCL fragment generated from the *Loop* node (label 5) ensures that there is at least one element in the collection that satisfies the rest of conditions in the path. To translate the call to the lazy rule (label 4), we bind the formal parameters of the rule to the actual parameters using nested let expressions that create new variable scopes (one in this case). The *Lazy rule invocation* node (label 3) does not produce any OCL as it just aggregates the paths passing through it. Then, the OCL for the *Subexpr* node (label 2) asserts that the correct part of the faulty expression is not undefined, to prevent a trivial answer from the model finder. Finally, the last step adds a problem-specific condition to the OCL path condition (label 1). In this case, it forces the existence of an object that lacks the feature typeDef, in order to cause a runtime error when executing the transformation. This will happen for any object not being an instance of XsElementDeclaration (i.e., the type for which we have statically checked that the feature exists).

Listing 2 shows the resulting OCL path condition.

```
1  XsElementDeclaration.allInstances()→
2    select(i | -- node 8
3      not i.typeDef.oclIsTypeOf(XsSimpleTypeDef) and
4      not ( i.isInOut() and i.isFault() )→
5    exists(i |
6      if i.typeDef.oclIsTypeOf(XsComplexTypeDef) -- node 7
7      then
8        let elems : Sequence(XsParticle) = -- node 6
9          i.typeDef.oclAsType(XsComplexTypeDef).
10           content.oclAsType(XsParticle).
11           term.oclAsType(XsModelGroup).
12           particles→select(p|
13             p.content.oclIsKindOf(XsElementDeclaration) )
14       in elems→exists(e | -- node 5
15         let i2 = e -- node 4
16         in if i2.content.oclIsUndefined() -- nodes 2, 3
17         then false
18         else not i2.content.oclIsKindOf(XsElementDeclaration) --node1
19         endif )
20     else false endif )
```

Listing 2. Generated OCL path condition for the problem in line 53 of the running example.

It is worth noting that, sometimes, our procedure may yield path conditions which are not OCL compliant. This is the case when the path condition includes ATL-specific constructs not provided by OCL, like calls to helpers. In Section 7.2, we identify the situations where this may happen, and provide rewriting rules that overcome this problem for some of them. As an example, lines 9-11 of Listing 2 had to be rewritten to add explicit castings via oclAsType, in order to obtain an OCL compliant expression.

5.3 Generating transformation pre-conditions

A transformation may have not been designed to work with every possible instance of the input metamodel, but only with a subset of them. The set of acceptable models may be undocumented, and the transformation execution may fail for models which are not in this set. When our static analysis reports an error, the developer can still discard the error in case it is a known limitation of the transformation (i.e., the transformation is not intended to work with the class of models that produce the error).

For example, line 15 (i.schema.elemDcl) contains an error *feature access over possibly undefined receptor* since the cardinality of schema is 0..1, meaning that there may be ElementType objects without a schema. However, the developer probably assumed that, for this transformation, any well-formed model requires ElementType objects to have a schema. This assumption can be documented with the following pre-condition:

```
1  ElementType.allInstances()→
2    forAll(i | not i.schema.oclIsUndefined())
```

To facilitate the creation of such pre-conditions, we propose a method that, given a typing error,

TABLE 3
Translation of ATL elements into OCL conditions: Control flow directives.

Element	ATL	OCL condition	Description
Matched rule	**rule** r { **from** t : T (filter) ... }	T.allInstances()→ select(t | filter)→ exists(t | *depNode*)	The model should contain at least one object t with compatible type (T) satisfying the rule filter. If the rule has several input elements, we generate nested allInstances expressions.

Matched rule (abstract)	**abstract rule** r (filter) { }
	rule r1 extends r { }
	rule r2 extends r { }
		if filter **then**
		depNode
		else
		false
		endif
			The execution of any subrule will lead to the execution of the abstract rule, provided its filter is satisfied. Although the filter of subrules should be more restrictive than the one of the abstract rule, the filter is checked by this code just in case programmers missed this requirement.

If expression	**if** condition **then** branchToError **else** theOtherBranch **endif**	**if** condition **then** *depNode* **else** false **endif**	The case in which the false branch leads to error just swaps the then/else parts.
Iterator expression (loop)	expr→collect(it | exprWithError) →followingOperations	expr→exists(it | *depNode*)	The operator exists ensures that the collection contains some problematic element. Any following operation is ignored.
Sub-expression (mono-valued property)	expr.property.invalidAtt.otherNav	**if not** expr.property.oclIsUndefined() **then** *depNode* **else** false **endif**	The condition demands the existence of property; thus, accessing the invalid attribute invalidAtt triggers an error (the problem-specific node is *true*).
Sub-expression (multi-valued property)	expr.property→invalidOp()	expr.property→exists(p | *depNode*)	The operator exists ensures that the collection contains some problematic element; thus, calling the invalid operation invalidOp triggers an error (the problem-specific node is *true*).
Call to helper with context	**helper context** T **def** : aHelper(p1 : T2) : T3 = self.exprWithError ... exprOfTypeT.aHelper(exprParam)	**let** genSelf = exprOfTypeT **in let** p1 = exprParam **in** *depNode*	The body of the helper is inlined, replacing self with a new variable that contains the receptor object (e.g., genSelf). Parameter passing is similar.

Call to helper with context (polymorphic call)	**helper context** Tsub
	def : aHelper(p1 : T2) : T3 =
	self.exprWithError
	...
	exprOfTypeT.aHelper(exprParam)
		if genSelf.oclIsKindOf(Tsub) **then**
		let genSelf_rebound = genSelf.
		oclAsType(Tsub)
		in *depNode*
		else
		false
		endif
			The call is polymorphic if the type of the receptor (exprOfTypeT) is a supertype of the context type of the helper in which the error is located (Tsub). genSelf needs to be rebound with a proper casting.

Call to helper without context, or lazy rule, or called rule	**helper def** : aHelper(p : T) : T2 = exprWithError ... thisModule.aHelper(exprParam)	**let** p = exprParam **in** *depNode*	Similar to context helpers, but with unbound context. Lazy rules and called rules are treated as global helpers, in which the "from" part of lazy rules are parameters.

synthesizes an OCL constraint such that any model that satisfies the constraint will not exercise the erroneous statement. This constraint will be used as a pre-condition that should be satisfied by any input model of the transformation. During the analysis, we attach all existing pre-conditions to the generated OCL path condition, to enforce that any generated witness model will fulfil them. A consequence of supporting pre-conditions is that any detected problem categorized as *sometimes-solver* may need to be promoted to potential problem if there are pre-conditions affecting its behaviour. Hence, the previously shown pre-condition will turn the problem in line 15 into a potential problem as it cannot be statically ensured without using the model finder.

Our method to generate pre-conditions is a variation of the method to generate OCL path conditions. We perform a bottom-up traversal of the error path, generating a piece of the pre-condition according to the type of each node. The generation rules are similar to the ones in Table 3, but instead of exists iterators, we generate forAll iterators since the pre-condition needs to ensure that any element "flowing" through the path satisfies the problem-specific constraint. The problem-specific constraints for pre-conditions are just the negation of those in Table 4.

Listing 3 shows the pre-condition generated for the problem *binding resolved with invalid target* in line 81. The problem happens because some XsElementDeclaration objects are transformed into Class by rule ClassR. Hence, the pre-condition must ensure that any XsElementDeclaration reached through the path (lines 1–3) does not fulfil the input pattern of rule ClassR (lines 5–10).

```
1  Input.allInstances()→
2    select(i | i.oclIsTypeOf(Input))→
3    forAll(i | let e : XsElementDeclaration = i.element in
4      -- negated input pattern of the guilty rule ClassR
5      not if ( e.oclIsKindOf(XsElementDeclaration) ) then
6              not e.typeDef.oclIsTypeOf(XsSimpleTypeDef) and
7              not (e.isInOut() and e.isFault())
8          else
9              false
10         endif)
```

Listing 3. Generated transformation pre-condition

In practice, our pre-conditions are written in the transformation header as comments prefixed with @pre. We internally parse this code and merge it with the rest of the abstract syntax model, so that it can be processed by our analyser (e.g., to report syntax or

TABLE 4
Translation of ATL elements into OCL conditions: Problem-specific conditions.

Problem (see Table 1)	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.
Feature access over possibly undefined receptor	<expr : T>.feature	expr.oclIsUndefined()	expr must be undefined.
Feature access over possibly undefined receptor via empty collection	aCollection→first().feature	aCollection→isEmpty()	Access to a single element of a collection via operations like first, last or at. aCollection must be empty. We consider collections coming from multivalued features and "allInstances()".

Binding resolved by rule with invalid target	feature ← <expr : T>
	...
	rule r { **from** t : T1 (guard(t)) ... }
	// with T1<:T
		// If expr is mono−valued:
		guard(expr)
		// If expr is multi−valued:
		expr→exists(t | guard(t))
			To force that expr contains objects accepted by rule r, expr must fulfil the rule guard. If T1 is subtype of T, the extra check expr.oclIsKindOf(T1) is added.

Binding possibly unresolved	feature ← <expr : T> ... **rule** r1 { **from** t : T1 (guard1(t)) ... } **rule** rn { **from** t : Tn (guardn(t)) ... } // with T1<:T, ...,Tn<: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 multivalued is similar.
Rule conflict	**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 closest descendant of T1 and T2. If T1=T2, we take T=T1.

typing errors within the pre-condition itself).

6 MODEL FINDING

The procedure described in Section 5.2 yields a collection of OCL path conditions for an error (i.e., one for each path to the error). Next, we show how to use these conditions to generate a witness model that exercises the error (Section 6.1), and how to interpret the result of the search (Section 6.2).

6.1 Generation of witness models

To obtain a witness model for an error, we take the disjunction of the OCL path conditions to the error, and declare the resulting expression as an invariant of an artificial class called ThisModule added to the input meta-model. This class 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 a class. Then, we use a model finder (in particular the USE Validator [10]) to check whether there is a model that satisfies all invariants, for which we force one object of type ThisModule, which is later discarded from the returned model.

Most model finders rely on bounded search, exploring models up to a certain size, typically given as a range for the number of objects of each class. Hence, they employ the *"small scope hypothesis"* [29], [30] and assume that most constraints are satisfied by models of limited size. In our case, we perform searches of increasing scope (i.e., we first look for a model with at most one object of each class, then with at most two, and so on) until a model within the given scope is found, or a predefined upper bound is reached. This allows obtaining smaller witnesses, where the problematic model configuration is easier to identify. Implementation-wise, we set the upper bound to 5, but this value can be configured easily. This strategy leads to faster execution times than just setting a fixed upper bound.

To further reduce the search space, we use a pruned version of the input meta-model. We consider two different strategies with increasingly larger pruned input meta-model: *error path* and *mandatory*.

Error path strategy. This strategy considers the *path effective meta-model* (MM_{path}), which is the minimum meta-model required to trigger the error at runtime. This meta-model contains the types and features used in the OCL path condition, which are extracted from the TDG when the error path is computed. In some cases, it also needs to be extended with classes that do not appear in the transformation but are needed to generate the witness model. There are two such cases. First, if MM_{path} contains abstract types without children, an arbitrary child type among those in the complete meta-model is added. This is done because, otherwise, the generated witness would not contain instances of the abstract type. As an example, this happens if the error path traverses a rule with an abstract type in its input pattern, but no subclass is explicitly used in the rest of the path.

Second, for some problems, we need to consider types not mentioned in the error path. For example, the problem in line 81 arises when the expression i.elem yields an XsElementDeclaration object which satisfies the filter of rule ClassR (i.e., an XsElementDeclaration object connected to an XsComplexTypeDef object). However, the error path does not include XsComplexTypeDef because it is not explicitly mentioned. In these cases, our approach is to search for oclIsKindOf(T) occurrences in the error path and, if possible, add a sibling type of T that does not belong to the error path yet.

This meta-model pruning strategy reduces the

search scope for the model finder. As an example, the error path meta-model for the path condition in Listing 2 (which corresponds to the problematic statement i.content.typeDef in line 53 of Listing 1) excludes classes like Description, Interface, Service, ElementType and XsSchema as they are not used in the path condition. Similarly, features like Operation.name are excluded as well. In this case, the model finder does not find a witness satisfying this path condition because the check performed in line 41 ensures that the type of i.content can only be XsElementDeclaration, which defines the feature typeDef, and thus, no error will arise.

Just for illustration, suppose we comment the check in line 41. Then, we obtain the witness model shown in Figure 12. This has an XsParticle object whose content is of type XsModelGroup, which lacks feature typeDef. Thus, the transformation will fail for this input model when executing line 53.

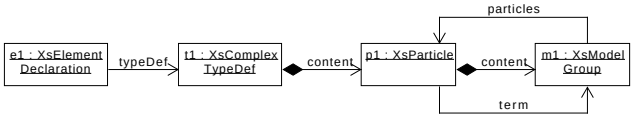

Fig. 12. Witness model for problem in line 53, using the *error path meta-model* strategy.

The presented strategy does not guarantee that the generated witness conforms to the original meta-model, as the pruned meta-model version used for finding the witness may lack compulsory features not required by the error path. Moreover, executing the transformation with the witness may cause unexpected errors because the witness may be incomplete.

Mandatory strategy. This strategy ensures that the generated witness model conforms to the original meta-model. For this purpose, it uses a pruned version of the input meta-model which contains the elements of MM_{path} plus the mandatory types and features of the complete input meta-model. The generated meta-model is called the *error meta-model*.

More in detail, let MM_{path} be the meta-model containing all types and features used in the OCL path condition as explained above, and MM_{input} the complete input meta-model. Then, for each class in MM_{path}, we add the opposite of the references it declares, as well as all mandatory attributes and references that the same class defines in MM_{input}. If the added feature is not owned by the class, but it is an inherited feature, our algorithm adds to MM_{path} the necessary ancestors, and adds the feature in the appropriate ancestor. Likewise, any added ancestor needs to include its mandatory features, and so on.

Using this strategy, we obtain the witness model in Figure 13 for the problem in line 53 (assuming we have removed the check in line 41). This is a valid instance of the WSDL meta-model, but its computation is more demanding for the model finder than the computation performed by the *error path* strategy. We analyse and compare the efficiency of both strategies in Section 8.3.

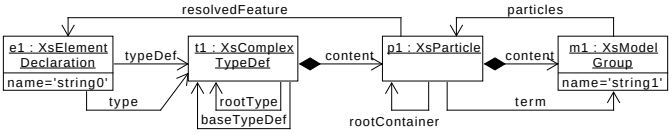

Fig. 13. Witness model for problem in line 53, using the *mandatory meta-model* strategy.

6.2 Interpreting the witness generation results

We use a two-phase model finding strategy to discard potential problems as fast as possible using the smallest search scope. First, we use the *error path* strategy. If this strategy does not find a witness model, then the problem is marked as *discarded* since the transformation logic prevents the error at runtime. Moreover, in this case, the *mandatory* strategy will not find a witness either, as this latter strategy adds more elements and constraints to the search. A caveat concerning our bounded search is that it may lead to false negatives, as it discards any problem for which no witness is found, but this may be due to using a too small search bound. In case the model finder yields a timeout, we mark the problem as *unconfirmed* (i.e., its state is unknown).

On the contrary, if the *error path* strategy finds a witness model, then the problem becomes *confirmed* (provided the complete input meta-model is instantiable). In such a case, it may be necessary to use the *mandatory* strategy to obtain a valid instance of the input meta-model satisfying the path condition, since the first strategy may exclude some mandatory meta-model features. In practice, we use this second step as a debugging mechanism to obtain example models. A possible situation in which only the first strategy, but not the second, returns a witness is when the meta-model is erroneous and cannot be instantiated at all. To discard this possibility, we provide the user with the option to verify the strong satisfiability of the transformation meta-models [31], i.e., whether there is some meta-model instance that contains at least one instance of every class. As Section 8.3 will show, we have not found cases where the meta-model is instantiable and the *error path* strategy finds a witness but the *mandatory* strategy does not, while the former strategy leads to faster solving times.

7 Technical Aspects of the Analyser

Our method is supported by an Eclipse plugin called anATLyzer, which can also be used as a plain Java API to implement other kinds of analysis of ATL transformations. Its source code and its update site are available at http://www.miso.es/tools/anATLyzer.html.

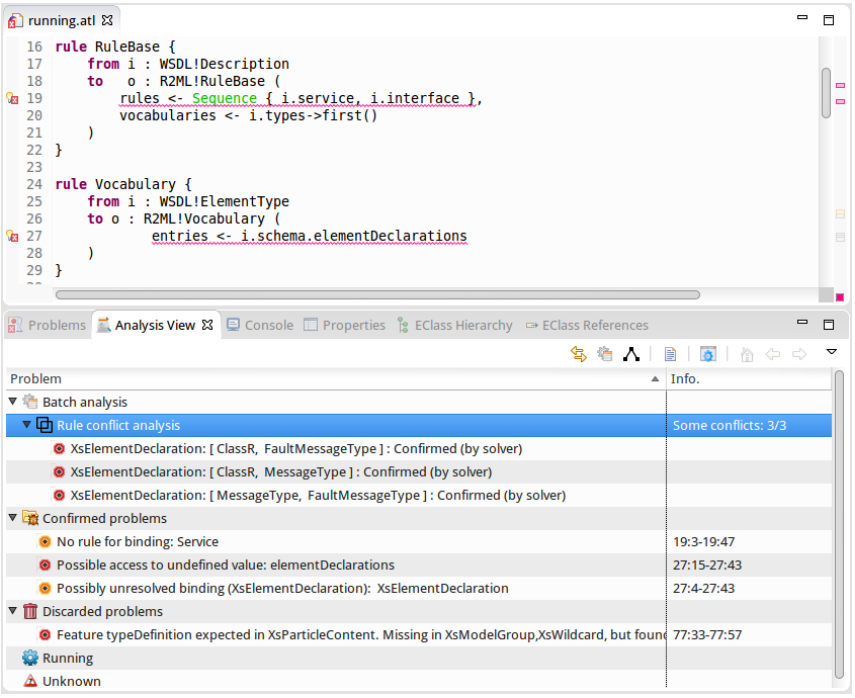

Fig. 14. Enhanced ATL editor.

Figure 14 shows a screenshot of ANATLYZER. It is seamlessly integrated with the ATL editor, so that whenever the edited transformation is saved, its errors and warnings are reported as editor markers (upper view of Figure 14) and in the regular Eclipse problem view. We have extended the original version of the analyser [12] with features intended to improve its usability, such as an extensible architecture to add new kinds of analysis, a view to inspect problems more easily (lower view of Figure 14), the possibility to silence the report of uninteresting errors, the handling of transformation pre-conditions, a better alignment with the USE Validator, and the option of running the witness generator in the background. The remaining of this section describes these features and other technical aspects on our tool implementation.

7.1 Architecture

Figure 15 shows the main components of ANATLYZER. The *Static Analyser* is a standalone API that processes the AST of an ATL transformation, returning a copy conformant to the extended ATL meta-model shown in Figure 9, annotated with type information. Additionally, it creates a *problem model* with details of all detected errors and warnings. The *Path Condition Generator* computes the problem *path condition*, which is represented as a set of visitable nodes, following the Visitor design pattern. The actual generation of the OCL code to feed the model finder is performed by a traversal of this path condition. Regarding the integration with the ATL Editor, our ANATLYZER *Eclipse plug-in* extends the original *ATL Editor plug-in* with a builder that invokes the analyser upon file saving, marking the detected errors in the ATL editor and updating our *Analysis View*. Finally, the *Analysis Index* identifies the already analysed transformations and notifies changes to interested parties.

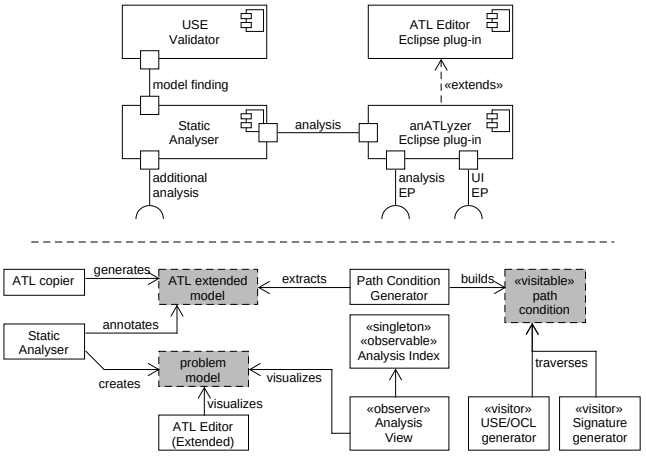

Fig. 15. Architecture of ANATLYZER. The upper diagram shows its main components, and the lower diagram a conceptual model of the relations between its parts. Gray boxes represent models.

The analyser is extensible with new kinds of analysis, which can be configured both in stand-alone mode or using an Eclipse extension point. An example of additional analysis is the support for UML profiles, for which we are able to point out some errors when dealing with stereotypes. There is also an Eclipse extension point to configure actions for the *Analysis*

View.

This architecture does not only support our Eclipse plug-in, but it can also be used by third parties as an analysis library to improve current support of ATL transformations. Moreover, it is an enabler of empirical evaluations for ATL transformations, which up to now have been limited due to the little information provided by the standard ATL compiler.

7.2 Bridging ATL and USE/OCL

The OCL path condition generated by the procedure described in Section 5 may include expressions that are not compliant with standard OCL. For example, OCL does not support global helpers, and therefore, an ATL call such as thisModule.isInOutPattern(e) is invalid. Another source of incompatibilities is the fact that OCL expressions in ATL are mostly untyped, while they must be explicitly and correctly typed to allow their processing by OCL-compliant model finders. The works that apply straightforward translations of ATL into OCL for model finding [32], [33], [34], [35] only work for simple transformations, and do not deal with real-world scenarios like the complete ATL Zoo.

Next, we present the rewriting rules that we have implemented to yield correct OCL for the USE Validator model finder [10], though they can be applied to other model finders with the corresponding adaptations. Table 5 summarizes the most important ones, classified in two categories: those due to incompatibilities between ATL and standard OCL, and limitations of USE that require special handling. In both cases, the rewriting is performed at the abstract syntax level, using the typing information computed in the first step of the analysis and propagated to the path condition.

In particular, the following rewriting rules bridge incompatibilities between ATL and OCL:

Primitive helper inlining. Unlike OCL, ATL supports the use of primitive types as context for helpers, e.g., adding the operation Integer.succ = self + 1. The solution to this problem is to inline the calls to this kind of helpers.

Binding flattening. In ATL, bindings whose right part is a collection of collections are automatically flattened. Thus, we add an explicit call to the flatten operation, to avoid typing errors in the problem-specific part of the path condition.

Misused call operator. OCL supports two types of calls: dot-notation for plain objects and arrow-notation for collections. However, ATL does not enforce this notation and it is possible to interchange them, in which case, the expression needs to be rewritten to use the proper call.

Invalid variable declaration. ATL does not check if the type of a variable declaration, or the return type of a helper, is correct with respect to the initializing expression. To avoid typing problems, we replace the declared type with the type inferred by our analyzer.

Implicit casting. Statically typed languages usually provide an explicit downcasting operation to inform the analyser about the expected type when a feature defined in a subtype must be accessed. In OCL, this is performed using the oclAsType operation; however, ATL lacks this operation because it is a dynamic language.

Thus, we take advantage of the type annotations in the AST to insert downcasting operations in those places that would raise a *feature found in subtype* error otherwise. That is, we convert the implicit castings performed via oclIsKindOf checks (see Section 4.3) to explicit castings with oclAsType. Figure 11 showed some necessary castings for the example (marked in grey), which are also shown in lines 9–11 of Listing 2.

Access to parent of an object. ATL provides the refImmediateComposite built-in operation to access the container of an object. We emulate this operation by generating an OCL operation for each metamodel type, which computes the container of the type instances.

Non-standard operations. ATL supports some operations that are not available in OCL. Some can be removed safely (e.g., debug) or replaced by equivalent ones. In the rest of cases, we signal that the path condition cannot be evaluated.

Alignment of two-valued and three-valued logic. USE/OCL makes use of a three-valued logic, so that Boolean values may be *true*, *false* or *undefined*. However, ATL uses EMF [36] as the underlying modelling framework, which uses a two-valued logic and therefore a Boolean attribute cannot be undefined. Similar issues occur with primitive types like Integer and Real. In order to solve this mismatch, for each primitive attribute of these types we generate a constraint like OwnerClass.allInstances()→forAll(o | not o.theAttribute.oclIsUndefined()).

The following rules tackle some limitations of USE:

Global helper. ATL supports the definition of static operations in the context of a transformation module. However, in USE, operations must be defined in the context of a class[11]. To address this mismatch, each global helper used in the path condition is defined in the context of a new auxiliary class called ThisModule. Then, we create a singleton object representing the transformation by wrapping the original path condition with ThisModule.allInstances()→exists(thisModule| <path>). Finally, we replace every call over the thisModule ATL global variable with the iterator variable. This requires adding a new parameter to every context

11. The OCL specification supports this via the *static* keyword, but only for Complete OCL.

TABLE 5
Path condition rewriting rules to bridge ATL and USE/OCL.

Rewriting	ATL	OCL-compliant expression	Description
OCL mismatches			
Primitive helper inlining	**helper context** Integer **def**: succ(): Integer = self + 1; ... 0.succ() ∗ 2	(0 + 1) ∗ 2	The receptor of 0.succ is an Integer. The helper succ must be inlined, replacing self and parameters with the actual values.
Binding flattening	**rule** aRule { **from** i : T1 **to** o : T2 (feature ← Sequence {i.mf1, i.mf2})}	Sequence {i.mf1, i.mf2}→flatten()	If mf1/mf2 are multivalued, the right part of the binding is a nested collection. This must be flattened to allow a correct implementation of the problem-specific part of binding problems.
Misused call operator	anObject→operation() aCollection.operation()	anObject.operation() aCollection→operation()	The arrow operator used on objects is changed by the dot operator. The dot operator used on collections is changed by the arrow operator.
Invalid variable declaration	**let** var : OclAny = expr−yielding−T	**let** var : T = expr−yielding−T	The declared type is replaced by the inferred type.
Implicit casting	−− *condition−based casting:* **if** obj.oclIsKindOf(T) **then** obj.propOfT ... −− *select−based casting:* aCol→select(obj | obj.oclIsKindOf(T)) →collect(obj.propOfT)	−− *condition−based casting:* **if** obj.oclIsKindOf(T) **then** obj.oclAsType(T).propOfT ... −− *select−based casting:* aCol→select(obj | obj.oclIsKindOf(T)) →collect(obj.oclAsType(T)) →collect(obj.propOfT)	Any access to a feature defined in a subtype is casted.
Access to parent of an object	−− *anObject is of type T* anObject.refImmediateComposite()	**context** T **def**: refImmediateComposite() = ...	We define a refImmediateComposite operation for each meta-model type.
USE mismatches			
Global helper	**helper def**: aGlobalHelper() : T = ... **helper context** T **def**: aCtxHelper() : T = thisModule.aGlobalHelper;	**context** ThisModule **def**: aGlobalHelper() : T = ... **context** T **def**: aCtxHelper(tm : ThisModule) : T = tm.aGlobalHelper(); ThisModule.allInstances() →exists(tm |... aCtxHelper(tm))	The new class ThisModule is added to the meta-model, declaring all global helpers. A singleton object of the class is passed as parameter.
Meta-model flattening	+ RootPkg ClassA − SubPkg ClassB	+ RootPkg ClassA SubPkg_ClassB	The meta-model is flattened. The renamed meta-model types are also renamed in the path condition.
Unsupported sequence operation	aSequence→first()	aSequence→any(x | true)	

helper that invokes thisModule, passing the iterator variable as an actual parameter in each call.

Meta-model flattening. A limitation of USE is that it does not support nested packages. However, large meta-models (like those in our running example) are organized in this way. For this reason, we flatten the meta-models that contain nested packages before computing the error meta-model. This is a recursive procedure where each classifier name is added the name of the container package as a prefix, moving the classifier to the ancestor package, until the root package is reached.

Unsupported sequence operation. The USE Validator cannot handle Sequences, but it automatically converts them into Sets. Thus, specific operations of Sequence, like first or iterate, are not supported. Since the first operation is used frequently, we emulate it by col→any(x | true).

Name clashes. The set of reserved keywords in ATL and USE/OCL differ, and it may happen that the name of a meta-model element or a helper collides with a reserved keyword in USE. Thus, we keep a catalogue of such keywords to rename meta-model elements and helpers accordingly.

Different branch types in conditional expression. USE requires the type of each branch in a conditional expression to be compatible. ATL does not guarantee this, and in general, it is not possible to rewrite the expression to enforce it. Hence, we detect this situation, and report that the path condition cannot be evaluated.

For the remaining cases that we are not able to handle, our tool does not crash, but it classifies the problem from which the path condition was built as *unknown* (i.e., the problem is neither confirmed nor discarded).

7.3 Error dependencies

Transformations may contain problems which are not independent. We say that the resolution of a problem p_1 depends on the resolution of another problem p_2, if p_2 occurs within the error path of p_1. We compute these dependencies and recommend the user to correct first the problems that do not depend on any other. In this way, dependent problems can be resolved more accurately and robustly once the errors on their path have been removed, or they may even become fixed if they are false positives signalled due to the dependency with another problem.

To identify the error dependencies in a transformation, we compute the error path of each error individually, and construct an *error forest* that contains one tree for each error that does not depend on other errors. Then, the dependent errors are added as children of the corresponding tree recursively.

Figure 16 shows one of the trees of error dependencies computed for the running example. The upper node does not depend on any problem. The error in the second node depends on the previous one as it occurs in the same navigation expression term.particles, and the path to particles includes the path to term. The error in the leaf node occurs in a lazy rule invoked by rule ClassR, and such an invocation always requires executing the expressions in the previous errors.

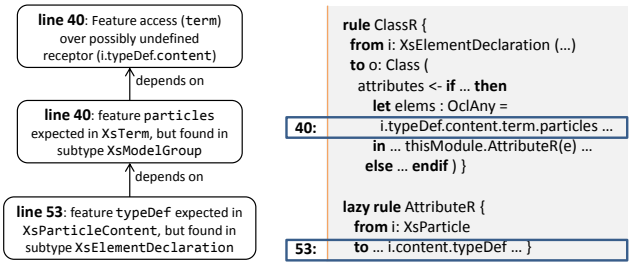

Fig. 16. Excerpt of error dependency forest for the running example.

7.4 Continuous analysis tool

ANATLYZER is integrated with the ATL IDE. Thus, it requires good response times to analyse the transformation as it is being developed and report problems without interrupting the editing process. However, in contrast to other analysis tools, ours requires launching a model finder. This may be time-consuming if many potential problems need to be confirmed as the transformation is being edited.

In our first version of ANATLYZER [12], the generation of witnesses for a given problem had to be launched manually to avoid delays in the UI thread. This affected negatively the user experience and was tedious. Hence, inspired by the notion of *continuous analysis tool* [13], we now run the model finder in the background and report errors of type *always/sometimes solver* once they are confirmed (i.e., errors being evaluated in the background appear under the "running" label, see Figure 14). However, this does not solve the technical issue of how to keep track of already evaluated errors when the transformation changes, in order to avoid executing the model finder unnecessarily. We cannot use line numbers as they easily change when the code is edited, while annotating each faulty statement pollutes the code if there are many errors.

Our solution to this issue is the generation of a different signature for each error, which is used to determine if a problem detected in one pass of the analyser is the same as another problem detected in the following pass. The signature of a problem contains the elements strictly needed to uniquely identify the problem, and becomes "invalidated" when some of these elements change. It is computed by performing a bottom-up traversal of the error path, starting from the execution nodes, and adding a piece of signature in each step. Table 6 shows the signature of each node kind in the error path. The sig function returns the signature of an OCL expression by a special string serialization that minimizes the number of characters. Optional syntax elements are marked with ?, in which case, nothing is generated if they are not defined. Note that the signature of a (non-abstract) matched rule does not include its name or output pattern, as they are irrelevant for witness generation.

This strategy permits modifying the code or the layout, without having to re-confirm all problems with the model finder. If a piece of code within an error path changes, the generated signature will differ and the problem will be re-evaluated.

Listing 4 shows part of the signature for the problem in line 81 of the running example. Line 1 corresponds to the signature of the TriggeringEvent matched rule. Lines 2–3, which are derived from the problem-specific part of the error path, include the signatures of the guilty matched rules (we only show rule ClassR).

```
1 MessageEventExpression;i.oclIsTypeOf(WSDL!Input)|
2   InvalidTarget;WSDL!XsElementDeclaration;
3   and[not[i.typeDef.oclIsTypeOf(WSDL!XsSimpleTypeDef)]...]
```

Listing 4. Excerpt of signature for problem in line 81

Implementation-wise, we maintain an index with the detected problems and their signature. Problems that require confirmation using the model finder are assigned the status *witness required*. The background process evaluates each problem with this status, and updates the status to either *confirmed* or *discarded*. The next time the analyser is run, a new set of problems is generated. We use the stored signatures to compare the old problems with the new ones, and when there is a match, we assign to the new problem the status of the old one. Last, we discard the old problems and keep the new ones.

7.5 Analysis configuration

The behaviour of the analyser can be fine-tuned for a transformation by means of a configuration file with extension .atlc.

Listing 5 shows an example of configuration file. Line 1 activates the continuous analysis mode introduced in the previous subsection. We also support batch mode, typically used for exploring existing transformations and testing. Line 2 sets a timeout to abort the model finder execution if it takes too much time for a problem. The show-marker directive (lines 4–6) customizes the errors reported by the editor according to their status. By default, confirmed errors are always shown, while the typical configuration

TABLE 6
Computation of problem signatures.

Node	Element	Signature
Matched rule	Matched rule (R)	sig(R.inputTypes) + sig(R.filter?) + R.superRule?
Matched rule (abstract)	Abstract matched rule (R)	R.name + sig(R.inputTypes) + sig(R.filter?)
Call	Nav/OperationCallExp (C)	sig(C.receptor) + C.name + sig(C.args?)
Helper	Helper (H)	sig(H.context) + H.name + sig(H.argTypes?)
Imperative rule	Lazy/Called Rule (R)	R.name + sig(R.inputTypes)
If (true branch)	IfExp (I)	'T' + sig(I.condition)
Loop	IterateExp (I)	sig(I.receptor) + I.name
Subexpression	OclExpression (O)	sig(O.correctPart)
Let	LetExp (L)	L.varName + sig(L.initExpr)
Problem-specific	Problem (P)	P.typeName + specific signature

hides discarded errors. Developers may also choose to display errors that have not been evaluated yet (witness-required) or that the system has failed to evaluate (unknown). A finer-grained control is still possible with the ignore directive (line 8), to indicate that certain families of errors must not be reported, such as style warnings or specific types of errors like "feature found in subtype".

```
1 witness-finder continuous
2 timeout 20 secs
3
4 show-marker witness-required
5 show-marker unknown
6 # show-marker discarded
7
8 ignore style, feature-found-in-subtype
```

Listing 5. Example of configuration file

8 Evaluation

In this section, we evaluate the following aspects of our method and tool: accuracy of the reported problems (Section 8.1), usefulness (Section 8.2) and performance (Section 8.3). Next, we discuss possible threats to the validity of our evaluation (Section 8.4) and draw some conclusions (Section 8.5). Finally, we analyse the applicability of our technique to other transformation languages (Section 8.6). The artefacts to reproduce the experiments and the raw results are available in http://miso.es/anatlyzer_exp_tse.html.

8.1 Accuracy of analysis results

To assess the accuracy of our method, we use the *precision* and *recall* metrics originally defined in the information retrieval community [37]. Precision gives a measure of correctness or quality, and in our context, it will be computed as the fraction of real errors among all problems signalled by anATLyzer. Recall is a measure of completeness, and in our context, it will be computed as the fraction of errors detected by anATLyzer among all existing errors in a transformation.

Figure 17 outlines the process to compute these metrics. The idea is performing random testing on a set of mutants of a transformation, and comparing the results obtained by random testing with those reported by our analyser.

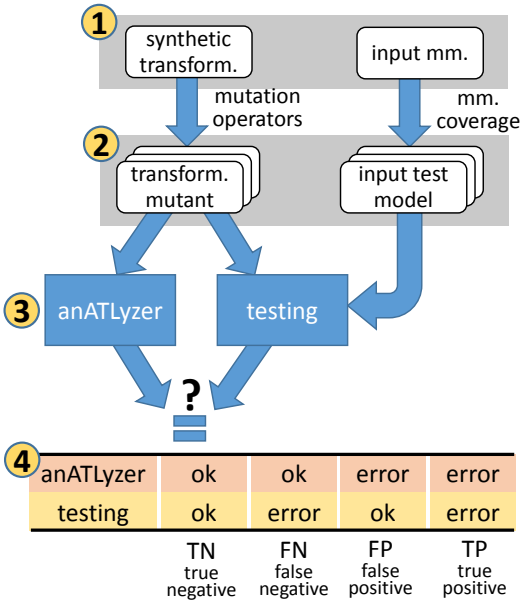

Fig. 17. Main steps in the evaluation of precision/recall.

Thus, in order to perform the evaluation, we first built a synthetic ATL transformation with as much ATL features as possible (i.e., rules of different kind, rule inheritance, helpers, navigation expressions, iterators, etc.) and with no errors. Table 7 shows the main features covered in the transformation, whose size is 133 lines of code (removing blank lines).

TABLE 7
Main features used by the synthetic transformation.

Features	#	Features	
Context helpers	7	If-else	✓
Module helpers	1	Collection iterators	✓
Matched rules	6	Bindings	✓
Abstract rules	1	resolveTemp	✓
Rule filters	6	allInstances	✓
Lazy rules	1	refImmediateComposite	✓
Called rules	3	oclIsKindOf	✓

From this transformation, we generated a set of transformation mutants, each one of them obtained by applying a single mutation operator to the original transformation. Table 8 lists the considered mutation operators. Note that each operator may be applied

on different locations, yielding a different mutant in each case. Moreover, not all mutants necessarily contained typing errors (e.g., deleting a binding for an optional feature may introduce no typing error); in this way, our test set contained both faulty and correct transformation mutants.

TABLE 8
Mutation operators for ATL transformations.

Type	Targets
Creation	binding source/target pattern element rule inheritance relation
Deletion	rule, helper binding source/target pattern element rule filter rule inheritance relation operation context formal/actual parameter in operation or called rule argument in operation invocation parameter in operation or called rule definition variable definition
Type modification	type of source/target pattern element helper context type, helper return type type of variable or collection parameter type of operation or called rule definition type parameter (e.g., oclIsKindOf(Type))
Feature name modification	navigation expression target of binding
Operation modification	predefined operator (e.g., and) or operation (e.g., size) collection operation (e.g., includes) iterator (e.g., exists, collect) operation/attribute helper invocation

Our evaluation uses a synthetic transformation because we did not find any existing transformation covering all features of ATL and without errors. Moreover, the fact that each mutant is a small variant of a correct transformation makes it less possible to have inter-dependent errors in mutants, which may distort the evaluation results. The evaluation in the next section considers real transformations with several errors in each one of them.

In addition to the transformation mutants, we automatically generated a set of input test models ensuring coverage of all classes and relationships in the input meta-model [6]. These models were used for testing the mutants. In this way, a transformation mutant is deemed correct if, for each input model in the generated test set, its execution does neither raise a runtime error nor yield an ill-typed target model; otherwise, the mutant is proved to be erroneous.

As a last step, we compared the result obtained with testing and with ANATLYZER for each mutant. Figure 17 shows the four possible outcomes of the comparison: *true negative*, *true positive*, *false negative* (ANATLYZER fails to report an existing error), and *false positive* (ANATLYZER reports an error incorrectly). Based on these results, we computed the precision, recall, accuracy and f-measure using formula 1, 2, 3 and 4, respectively.

$$precision = \frac{\#TP}{\#TP + \#FP} \quad (1)$$

$$recall = \frac{\#TP}{\#TP + \#FN} \quad (2)$$

$$accuracy = \frac{\#TP + \#TN}{\#TP + \#TN + \#FP + \#FN} \quad (3)$$

$$f - measure = 2 \times \frac{precision \times recall}{precision + recall} \quad (4)$$

Table 9 summarizes the obtained results, running the experiment with 541 transformation mutants, which were executed against 1089 test models. Column *Original* shows the resulting metrics. The precision of ANATLYZER is relatively high (0.82), and the recall is very high (0.98), meaning that ANATLYZER covers very well the ATL language. The low amount of false negatives is explained by our use of path conditions and model finding to widen the kind of problems detected.

TABLE 9
Accuracy of ANATLYZER.

	Original		Manually processed	
Evaluated transformations	541		483	
True positives	337	(62.29%)	337	(62.29%)
True negatives	125	(25.88%)	125	(25.88%)
False positives	73	(13.49%)	15	(3.11%)
False negatives	6	(1.11%)	6	(1.24%)
Precision	0.82		0.96	
Recall	0.98		0.98	
Accuracy	0.85		0.96	
F-measure	0.89		0.97	

However, some problems detected by our method (like behaviour and style warnings) do not imply a runtime error or a target meta-model disconformity, which are the kind of errors that random testing can detect. Thus, we filtered out the mutants exhibiting those errors from the evaluation, obtaining the metrics shown in the column *Manually processed*. By excluding these cases, the precision increases up to 0.96.

Regarding the discrepancies between the errors reported by testing and ANATLYZER, one false negative is due to infinite recursion in a helper (i.e., a stack overflow error), which is exposed by testing but it is beyond the capabilities of ANATLYZER. Another source of false positives is the presence of problems in dead code, which are signalled by ANATLYZER but cannot be exercised at runtime. Other false positives are related to problems in the rule inheritance hierarchy, which do not cause a runtime error. In the same line, we are not currently able to signal all rule hierarchy problems, which leads to some false negatives.

8.2 Usefulness

To evaluate how useful is our method to uncover errors in real transformations, in [12], we presented a preliminary study that analysed 19 existing transformations coming from the ATL Use Cases repository[12]. Though ANATLYZER was able to detect issues in all of them but one, the number of analysed transformations was very small. Thus, this section expands the evaluation by considering the complete ATL Zoo, which

12. http://www.eclipse.org/atl/usecases/

contains 101 unique transformations[13]. These transformations are mostly connected to research projects, range from very simple to large and complex ones, and have been contributed by different kinds of developers including experienced ones (e.g., the creator of ATL) and beginners (e.g., students). Thus, we use it as a representative sample of the variety of ATL constructs and problems that may occur in practice.

Table 10 shows a summary of the errors detected by ANATLYZER organized by severity and kind, considering only the errors confirmed either statically or by the model finder. The table shows the total number of detected problems (#*Occ.*), the percentage of each kind over this total (%), the number of transformations in which they appear (#*Trafos*), the maximum number of errors of each kind found in a single transformation (*Max*), and the average number of errors of each kind per transformation (*Avg*).

TABLE 10
Summary of errors organised by severity and type. The last row shows the aggregated values for all analysed transformations.

Severity	**#Occ.**	**%**	**#Trafos**	**Max**	**Avg**
error-load	21	0.4	9	9	0.2
error-target	2157	44.9	68	454	21.6
runtime-error	1201	25.0	68	349	12.0
warning-behaviour	801	16.7	54	430	8.0
warning-style	342	7.1	66	21	3.4
warning-perf	284	5.9	46	32	2.8
Kind	**#Occ.**	**%**	**#Trafos**	**Max**	**Avg**
navigation	1142	23.8	79	345	11.4
source typing	1125	23.4	65	429	11.3
target mm conformance	2158	44.9	68	454	21.6
transformation integrity	23	0.5	14	6	0.2
transformation rules	358	7.4	50	62	3.6
All	4806	100%	101	808	48.1

In total, ANATLYZER discovered 4806 problems. Regarding their severity, around 70% can be considered very severe since they can cause runtime errors or generate invalid models (*error-load*, *error-target* and *runtime-error*), and 16% are smells of wrong transformation behaviour (*warning-behaviour*). With respect to the kind of error, target conformance errors dominate, in particular errors of type *No binding for compulsory target feature* (see below). Navigation and source typing problems are also pervasive (e.g., there are navigation issues in 79 out of 101 transformations), which is explained by the extensive use of OCL in ATL transformations and the lack of checks in the regular ATL IDE.

Table 11 shows the 15 most prevalent problems detected by ANATLYZER, discarding style and performance issues. We show the total number of occurrences per problem (#*Occ.*), the percentage over this total number (%), the number of transformations that contain some problem of the given kind (#*Trafos*), and the maximum number of the given kind of problem found in a single transformation (*Max*). *Avg* is the average of occurrences over all transformations in the Zoo, whereas *Avg (if some)* is the average over the transformations with at least one problem of the given kind.

The most recurring problem is *No binding for compulsory target feature*, which produces output models that do not conform to their meta-model. This may break any tool or transformation using the generated output model. Problems #2, #8 and #11 indicate discrepancies between the declared types and the types inferred by ANATLYZER, which are important to consider because incorrect declared types may confound the developer. There are also many problems related to feature access (#3, #4, #9, #10, #13, #14), which reflects the need for IDE facilities helping in detecting incorrect feature usages according to the meta-models. Regarding rule issues, there are 148 rule conflicts (#6) and 283 binding problems (#5, #7) which affect around 1/4 of the transformations in the Zoo. In general, we have seen that if a transformation has certain kind of problem, it will likely contain other problems of the same kind (column *Avg (if some)*). This may indicate that developers were not aware of the particular kind of problem, or that the involved meta-models tended to induce them. Thus, ANATLYZER can be useful not only to spot problems, but also to make developers aware of the transformation language semantics.

We have also analysed the use of the model finder within ANATLYZER. For this purpose, we collected all detected *sometimes-solver* and *always-solver* problems, and evaluated to what extent ANATLYZER provides an answer for them (i.e., either confirms or discards the problem). Table 12 summarizes the results. First, we show how many problems were confirmed statically (*ST*) or required the model finder (*MF*). In total, 8493 problems required model finding, most of them rule conflicts (7161) and binding related problems (746 + 405 + 70 = 1221). Then, we show how many answers provided the model finder, either to confirm (*Conf.*) or discard (*Disc.*) a problem. ANATLYZER gave an answer for 84% of the problems. Most problems were discarded (78%), but the number of confirmed problems was high (485 problems). The columns under *Not answered* contain the number of potential problems that ANATLYZER did not answer, and therefore, it is unknown whether they are actual problems or not (16% of the problems). The reason for this lack of answer was because either USE does not support some features appearing in the OCL path condition (column *USE lim.*, like the sortedBy operation) or due to internal errors produced by a variety of causes like incompatibilities between ATL/OCL and USE/OCL that we cannot solve (column *Other*, 3% of the cases). We have observed that, many times, the ultimate cause of such internal issues is the presence of other problems within the path condition, and thus, we were feeding the model finder with incorrect code

13. We used a snapshot of the ATL Zoo (http://www.eclipse.org/atl/atlTransformations/) downloaded on April 2015.

TABLE 11
Detail of the more significant problems in the ATL Zoo.

	Problem	#Occ.	%	#Trafos	Max	Avg	Avg (if some)
1	No binding for compulsory target feature	2009	48.80%	61	454	20.09	32.93
2	Invalid actual parameter type	572	11.90%	22	426	5.72	26.00
3	Feature access over possibly undefined receptor	539	11.22%	37	344	5.39	14.56
4	Feature found in subtype	180	3.75%	22	47	1.80	8.18
5	Binding possibly unresolved	178	3.70%	38	26	1.78	4.68
6	Rule conflict	148	3.08%	20	59	1.48	7.40
7	Binding resolved by rule with invalid target	105	2.18%	21	26	1.05	5.00
8	Incoherent helper return type	95	1.98%	38	10	0.95	2.50
9	Operation found in subtype	62	1.29%	8	26	0.62	7.75
10	Feature not found in union type	60	1.24%	11	24	0.60	5.45
11	Incoherent variable declaration	47	0.98%	22	5	0.47	2.14
12	Feature access over possibly undefined receptor via empty collection	46	0.96%	18	8	0.46	2.56
13	Feature not found	44	0.92%	15	12	0.44	2.94
14	Operation not found	43	0.89%	17	16	0.43	2.53
15	No rule to resolve binding	23	0.48%	10	5	0.23	2.30

TABLE 12
Summary of errors which required model finding.

	# Problems			Model finder answers			Not answered		
Problem	**#Occ.**	**#ST**	**#MF**	**Total**	**Conf.**	**Disc.**	**Total**	**USE lim.**	**Other**
Rule conflict	7164	3	7161	6415	145	6270	746	676	70
Binding possibly unresolved	746	0	746	402	178	224	344	212	132
Binding resolved by rule with invalid target	405	0	405	210	105	105	195	137	7
ResolveTemp possibly unresolved	70	0	70	36	9	27	34	29	58
Feature access over possibly undefined receptor via empty coll.	89	0	89	58	46	12	31	21	2
Feature found in subtype	192	180	12	0	0	0	12	10	10
Feature access over possibly undefined receptor	547	537	10	2	2	0	8	1	5
Total	9213	720	8493	7123	485	6638	1370	1086	284
Average (over problems requiring the model finder)	-	-	8481	84%	6%	78%	16%	13%	3%

that could not be evaluated. As a solution, the user could prioritize the order in which errors are solved, as explained in Section 7.3.

Altogether, this experiment demonstrates the usefulness of our analysis technique and the need for tools like ANATLYZER. So far, model transformations have been developed without analysis facilities, resulting in transformations with poor quality. In fact, no transformation in the ATL Zoo is free of problems, with an average number of 48 problems per transformation, and a median of 25 problems. Moreover, the fact that we have been able to process the complete Zoo is a milestone for our analyser and provides an insight about its completeness. Since there is not a complete ATL specification, some subtle aspects of the ATL engine semantics were assessed by inspecting its source code and writing small test cases, which complicated the implementation task.

8.3 Performance

In this section, we evaluate the performance of our tool to determine if it is adequate for its integration in an IDE. The benchmarks were run in a laptop with chipset Intel i7 and running a JDK 1.8 configured with options `-Xms1024m` and `-Xmx2048m`.

First, we have measured the time employed by each component of the analyser when evaluating all transformations in the ATL Zoo. Table 13 shows the resulting average and median times. Most components have a very small performance footprint. In particular, type analysis is very fast. The generation of the problem tree can be heavyweight if there are many dependent problems, hence the relatively large difference between its average and median. As expected, the performance of the model finding is the main factor in the global execution time of the analysis. Still, times in Table 13 are per transformation, which may imply many calls to the solver. The low value of the median indicates good performance in most cases.

TABLE 13
Average recorded times (in seconds) of the different components of the analyser.

Component	Average	Median
Parser	0.17	0.03
Type analysis	0.02	0.01
Path generation	∼0.0	∼0.0
Tree generation	0.15	∼0.0
Model finding	21.5	0.28
Total time	15.0	0.19

We have conducted another experiment to measure the ability of our approach to reduce the model finding times by using a pruned version of the input metamodel. To this end, we analysed the complete ATL Zoo using three strategies to compute the meta-model fed to the model finder, recording the model finding time for each potential problem found. We set a time

out of 15 seconds per problem to be able to finalize the experiment. The strategies are the ones explained in Section 6.1: the *error path* strategy uses the smallest possible meta-model and it is the default in ANATLYZER; the *mandatory* strategy extends the error path meta-model with all mandatory features in the complete meta-model, in order to ensure that any generated witness conforms to the complete meta-model; and as a baseline, the *full* strategy uses the complete meta-model without pruning to perform the search.

Table 14 shows the average and median of the solving times per error. As expected, the average model finding time is shorter in the *error path* strategy, obtaining a speedup of 3.5. The median of the finding times shows that this strategy leads to smaller times than using the full meta-model. Moreover, note that the real averages are actually higher, as we set a time out of 15 seconds. If we look at the number of time outs, there are only 36 for the *error path* strategy (5% of the potential problems will not receive an answer from the model finder), 83 for the *mandatory* strategy, and 289 for the full meta-model. This confirms the hypothesis that our method enables a practical usage of the model finder by reducing the meta-model size used in the search. The last three rows in the table measure the actual reduction by comparing the number of meta-model classes and features used by the benchmarked transformations. The meta-model size is greatly reduced for the *error path* and *mandatory* strategies, being the main difference the reduction of mandatory features, which leads to enhanced execution times for the *error path* strategy.

TABLE 14
Comparison of model finding times (in seconds) for different meta-model pruning strategies.

	Error path	Mandatory	Full
Average finding time	2.16	3.16	7.66
Speedup in finding time	3.5x	2.4x	-
Median finding time	0.11	0.23	10.20
Time outs (>15 secs)	36 (5%)	83 (11%)	289 (40%)
Deleted classes	70%	66%	-
Deleted features	97%	90%	-
Deleted mandatory features	97%	76%	-

8.4 Threats to validity

An external threat to the validity of the results concerning the accuracy of our method is the fact that we have evaluated mutants generated from a synthetic transformation, and not from a set of real transformations. However, we believe this is reasonable in this case, because it allowed us to start from a correct transformation covering most features of ATL. Moreover, the generated mutants were likely to contain at most one error, hence having very low chances to contain error dependencies that may distort the evaluation results. Instead, compiling a large set of real transformations with either 0 or 1 errors would have been challenging.

An internal threat to the validity of this evaluation is that mutations could be biased towards the generation of errors that we are able to detect. To limit this issue, the mutation operators were developed independently from the analyser. Another threat is that random testing may yield false negatives (the transformation has errors but the input test set has no model that allows reproducing the error). To minimise this problem, we manually checked all cases where testing discovered no error but the analyser reported an error, to assess that they were real false negatives. As regards to the obtained recall, it is a measure of completeness with respect to transformation problems that cause runtime errors or non-conformant target models, but not with respect to semantic errors, which would require from testing techniques using some form of oracle.

A more subtle threat in this evaluation is that we do not perform a fine-grained comparison of the actual kind of error reported by the analyser and the one raised by random testing, in order to check that both errors are actually the same. Thus, ANATLYZER might report an incorrect type of error, which would remain unnoticed in the evaluation (e.g., it signals the error *operation not found*, when the actual problem is *operation found in subtype*). Thus, the evaluation only considers whether an error is reported or not, but not its classification. We have checked that, sometimes, the errors reported by both methods are different, but the root cause is always the same. Performing such a fine-grained evaluation cannot be automated.

Concerning the evaluation of the usefulness of our tool, the main threat to the validity of the results is the particular selection of analysed transformations, since most of them are connected to research projects and may not be fully representative of industrial practice. Nonetheless, the number of transformations is high, they have varying levels of complexity, and they were developed by people with different proficiency in ATL. On the other hand, even if our evaluation shows that ANATLYZER is useful to detect errors in existing transformations, we would need to perform another experiment to measure to what extent it helps reducing the number of errors when developing new transformations.

Finally, there is no formal semantics for ATL, and most 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 in some cases. To mitigate this issue, 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. Moreover, the experiment presented in Section 8.1 shows that the problems predicted by the analyser

are frequently confirmed by brute-force testing, as the transformations fail at runtime. This is an indicator that our implementation aligns with the behaviour of the ATL engine.

8.5 Discussion

Next, we discuss on the outcomes of the conducted experiments, identifying strengths and limitations. Generally, the results indicate that our approach is well suited for the practical discovery of problems of diverse nature in ATL transformations.

With respect to its adequacy as a complement to the interactive construction of model transformations, the execution time of the analyser is low when there is no model finding, while the model finding time is still acceptable thanks to our meta-model pruning strategies. Moreover, we have implemented a novel mechanism to avoid running the model finder unnecessarily when the source file changes. Thus, our method seems to be suitable to be integrated in an IDE with the aim of facilitating its practical usage. Nevertheless, an empirical evaluation of the usability of our tool would be required to fully support this claim. This is left as future work.

Regarding accuracy, our evaluation shows that our analyser has high precision and recall. There are several reasons for this. On the one hand, the use of a model finder allows being less conservative in the problem detection phase than regular type checkers. On the other hand, our analyser makes a *closed-world* assumption, that is, it assumes that it is analysing the complete source code that will be executed. This means that no runtime extension to the transformation (like superimposing additional modules [38]) or to the meta-model are allowed. Finally, ATL is a domain-specific language with constructs geared to solve transformation-specific problems, and the more focused a language is, the more complete its analysis can be. This claim is acknowledged by DSLtrans [39], a turing-incomplete transformation language that enables full reasoning about confluence and termination. In our case, infinite recursion is a source of false negatives since ATL is turing-complete.

With respect to the ability of ANATLYZER to deal with real transformations, the ATL Zoo provides a valuable testbed, as its transformations range from purely declarative (i.e., only matched rules) to purely imperative (i.e., only called rules and imperative blocks), exhibiting many different programming styles. In this sense, our success rate when using the model finder is quite high, since we are running the model finder in an unconstrained scenario.

Some problems detected in the ATL Zoo actually uncover scenarios that the transformations were not designed to handle, instead of programmer mistakes. A typical example is an undocumented limitation of a transformation revealed by a *binding possibly unresolved* problem, signalling that certain object configuration is not handled by any rule. Another example is an under-constrained meta-model which allows building semantically incorrect models. For instance, some PetriNet meta-models used in the ATL Zoo permit connecting two places to an arc, which is semantically incorrect. In these cases, our method to generate pre-conditions helps in fixing the problem by documenting the transformation.

Altogether, the analysis of the ATL Zoo shows that transformations, most of them thought to be stable, may contain errors of diverse nature. Some may be due to the dynamic nature of ATL, but others are inherent to rule-based languages (e.g., non-initialized features, wrong rule resolution, etc.). Our method has uncovered a surprisingly high number of problems in the Zoo, proving that this kind of techniques is needed to improve the quality of model transformations.

Some of the problems we analyse could be avoided by construction, modifying ATL's syntax or semantics. In particular, the dynamic typing nature of ATL makes it more flexible, but also more error prone. A statically typed language would detect common errors like *Feature found in subtype*; however, it should also provide means to detect uninitialized features in target objects, which is the first cause for errors in ATL (see Table 11). The choice of a textual or graphical language syntax may impact on this kind of error. Graphical languages (like triple graph grammars, TGGs [40]) equipped with editors which provide object creation templates for all required fields, may help in preventing errors related to uninitialized target objects fields. Finally, the choice of implementing an implicit binding mechanism in ATL provides a natural way to access created target objects by accessing source ones, but it is also error prone (see *Binding possible unresolved* error in Table 11). Instead, languages with an explicit management of the traces (like TGGs) tend to be more verbose, but are safe from binding resolution errors.

Regarding the limitations of our approach, Table 15 summarizes the main ones. First, our technique discovers "syntactic" errors (i.e., rule and typing errors), but cannot be used to find "semantic" errors (i.e., output models that do not fulfil the developer expectations). Such kind or errors requires a testing approach using oracles.

TABLE 15
Summary of main issues and limitations.

1	Detection of "syntactic" errors, but not "semantic" errors
2	Limited precision in some scenarios, like usage of dummy expressions and reflection.
3	Some detected problems do not manifest at runtime (e.g., incorrect type declarations, errors in dead code)
4	Limitations for some analyses (e.g., infinite recursion)
5	Limitations of the model finder (e.g., unsupported sequence operations)

Second, it is possible to deceive the analyser intro-

ducing dummy expressions. For example, a binding like name ← Set {} will produce a false positive for the error *Collection assigned to mono-valued binding* if name is mono-valued and optional, and a false negative if name is multi-valued and compulsory. Another way to circumvent the checks of the analyser is using reflection. For instance, if we use the expression refSet(obj, feature, value) to reflectively set the mandatory feature feature of obj to value, the analyser will incorrectly report that feature has not been initialized.

Third, some problems detected by ANATLYZER do not cause runtime errors. For example, this is the case when there are discrepancies between the types declared in a transformation and those inferred by the analyser. This is indeed the main source of false positives in our evaluation, as we compare with the results of random testing. Nevertheless, it is important to report these problems, which in any case, could be easily repaired using quick fixes [41], [42].

Fourth, testing can detect some errors which are beyond the capabilities of ANATLYZER. A prototypical example is infinite recursion. Thus, testing is a technique complementary to ours. As part of our future work, we aim at guiding the generation of test models using the information gathered statically.

Finally, as we explained in Section 7.2, we generate OCL code for its processing by the USE Validator. Whereas we are able to translate most ATL constructs into USE, this latter has some limitations which we cannot overcome, such as the lack of support for recursive functions or sequence operations like iterate (initial work in this direction is available in [43]). Path conditions that contain these elements cannot be processed and no witness can be generated for them. Another limitation is that our implementation does not support the translation of maps and tuples.

8.6 Applicability beyond ATL

Next, we discuss the applicability of our techniques to other mainstream model-to-model transformation languages.

- **ATL-like languages**. Similar to ATL, languages like ETL [44] and RubyTL [45] rely on the notions of matched rule and binding, and are dynamically typed. Our approach can be easily adapted to them, upon some minor changes to tackle slight differences in their semantics. However, ETL and RubyTL have less constrained imperative constructs than ATL, which may make the implementation more difficult, especially the synthesis of OCL code compliant with the standard.
- **Imperative languages**. Kermeta [21] is an imperative language that can be used to implement model transformations. Its last version, K3, is built on top of Xtend. It is statically typed, and inherits from Xtend features like "type guards" (a limited form of implicit downcasting) and null-safe feature calls (an operator ?. which returns null if the receptor object is null). Our approach could be adapted to detect typing errors that do not use these features. However, such an adaptation would be difficult because Kermeta lacks transformation-specific constructs; therefore, many elements of our approach are not applicable, like the TDG, rule conflict detection and many typing problems.
  QVT-Operational (QVTo) [11] is an imperative language based on OCL. It is statically typed, and hence, many typing problems discussed in this work are natively handled by QVTo. However, it lacks some of our type checkings, like the detection of accesses to undefined values, invalid downcastings, or target-conformance problems. Although QVTo has an explicit rule call mechanism, there are especial kinds of rules (e.g., disjunct rules and guarded rules) which could be analysed statically using our approach based on model finding. Moreover, QVTo has support for pre-conditions, but they are not checked.
- **Declarative languages**. Tefkat [46] is a logic-based declarative language. It does not rely on OCL for model navigation, but on a custom language to define object patterns. Rules communicate values via intermediate data structures called tracking classes. Hence, a graph similar to our TDG could be constructed to express the communication between rules, detect problems related to target conformance, and perform model finding to identify implicit relations between rules.
  QVT-Relations (QVTr) [11] is statically typed and based on OCL. Similar to ATL helpers, it supports side-effect free query operations called functions. It provides two kinds of rules: Top-rules, which are executed for each match of their source pattern, and non-top rules, which are used by other rules. Rule resolution is explicit by naming the rule that should be satisfied (in when clauses) or invoked (in where clauses). Thus, constructing the TDG would be easy. As QVTr is based on OCL, type checkings regarding null pointers and downcastings are implementable as in ANATLYZER. Rule conflict analysis is less interesting in this case because it only applies to top rules, and moreover, no runtime error occurs if the same objects are matched by two rules.
- **Graph-based languages**. TGGs are a formal approach based on declarative, bi-directional, triple graph rules that model the synchronized evolution of a source graph, a target graph, and a correspondence (i.e., traces) graph. From these bi-directional rules, forward and backward transformation rules can be automatically generated. TGG rules explicitly manage the trace model, i.e., they need to explicitly state the traces to create and the objects pointed by the trace, and there

is no implicit rule scheduling mechanism, like ATL bindings. TGG rules are made of graphs, and hence, some TGG tools do not support OCL to express navigations (like eMoflon), while others (like the TGG interpreter) use it for application conditions and attribute constraints [47]. Operations and queries are normally not supported either. From our analyses, the most useful for TGGs is rule conflict detection to ensure that the results are deterministic, and finding rule dependencies with the TDG. For OCL-enabled TGG tools, our navigation analyses are immediately applicable.

In summary, our technique is fully applicable to transformation languages based on rules and relying on OCL or a similar navigation language. Having rules where the source object matching and target object creation are explicit permits to reliably construct the TDG and path conditions. This is particularly useful to reason about implicit rule resolution in ATL, or to analyse explicit rule invocations with some adaptations. Moreover, the fact that the source model is read-only in ATL, and that OCL queries and navigations are side-effect free, facilitates the analysis and the compilation of path conditions into OCL for model finding.

ATL-like languages cover all mentioned features, and thus, similar results can be expected. QVTo and QVTr do not have implicit rule resolution, but the TDG and the model finder could be used to analyse rules. Tefkat is not based on OCL, but still, it can benefit by our technique since it has implicit rule resolution. Regarding TGGs, it is possible to incorporate some basic rule analysis based on model finding, and if the particular TGG approach supports OCL, then more of our analyses could be incorporated to them. The case of imperative languages is the less favourable for our technique, since there is no notion of rule on which harness the transformation-specific analysis.

9 Related work

Next, we review related approaches for transformation verification, comparing with our work and stressing on our contributions. We focus on transformation testing, transformation analysis via constraint solving, transformation slicing, and static analysis of transformations. Detailed discussions on these topics can be found in [2], [48]. The section concludes revising works on static analysis of general-purpose dynamic programming languages.

9.1 Transformation testing

The main challenges in transformation testing [15], [49] are the automated generation of input test models and the availability of a suitable oracle. Most proposals employ black-box approaches, and some rely on model finders to generate test models [8]. For example, in [50], *partial models* are used to reflect test intentions, and get completed into full-fledged models using model finders. In [51], *Tracts* (a kind of model transformation contract) are used to specify transformation properties, and the ASSL language is used to generate input test models. The defined contracts are then used as oracles for testing. Other approaches rely on coverage of the meta-model [6] or the transformation requirements [8] as criteria for automated model generation.

Few works focus on white-box testing. In [52], the authors use the effective meta-model of each transformation rule to derive test cases, taking into account the meta-model constraints. The white-box testing approach in [53] generates a set of input models ensuring certain coverage of an ATL transformation. 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 models. Instead, we drive the generation of witnesses by the problems found by the static analysis.

Our work can be classified as white-box, as we generate input models (witnesses) driven by static analysis. These models are maximally effective as they allow reproducing a particular error. However, our technique should not be thought as a replacement for the previous testing methods, but complementary, directed to typing and rule errors. Instead, black box approaches like [8], [51] are directed to find semantical errors of the transformation (i.e., disconformities of the transformation implementation w.r.t. the specification), which our technique cannot handle.

9.2 Transformation analysis based on constraint solving

Many works use transformation models [54] 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. Transformation models can be built manually, or derived from transformations in languages like ATL or QVT.

In [32], [33], ATL transformations are translated into transformation models, and model finders are used to check whether the transformation can produce a model violating the output meta-model constraints. In [55], 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.

Thus, our main contribution in this area is a novel technique based on static analysis that extracts the

slice of the transformation that corresponds to the path to a particular error, builds an OCL expression stating the conditions for a model to reach the error, and uses model finders to generate one such model.

9.3 Program slicing and path conditions

Program slicing is a technique to determine the parts of a program that affects a given statement [56], [57]. Few works have adapted this idea to model transformations. One exception is [58], 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.

Program slices can be used to identify the dependencies of some fault, while path conditions characterize the input test data leading to the faulty statement. In [9], 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 from another one. To our knowledge, the use of path conditions and witnesses in model transformation is novel.

Path conditions are used by symbolic executors to describe each possible execution path of a program. In the model transformation setting, the DSLtrans transformation language uses this technique to fully verify transformation contracts [59]. This is possible because DSLtrans is turing-incomplete and implements a layering system to reduce the search space. In DSLtrans, path conditions are produced in a forward manner to cover all possible execution paths. Instead, we generate them backwards from the error location to the transformation entrypoints, as we are only interested in one execution path. In [60], a translation from ATL to DSLtrans enables the full verification of semantic transformation contracts for ATL. Instead, we do not focus on semantic issues but on typing and rule errors.

9.4 Static analysis of model transformations

Even though static analysis has been used in graph transformation to detect, e.g., rule conflicts and dependencies [3], its use in transformation languages closer to programming languages, like ATL or QVT, is still an exception. The literature reports on three main applications of static analysis: maintainability of existing transformations [61], [62], generation of test input models, and static detection of errors. We focus on the last two applications.

Regarding test generation, in [63], 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 [64], the authors introduce a Java Façade for ATL that 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 transformations is presented in [65]. VIATRA2 transformations are made of declarative patterns with untyped parameters and the goal is to ensure a correct typing of the parameters with respect to the meta-models. This is checked using constraint solving. In our case, the type-checking is on ATL, which heavily relies on OCL, and hence type-checking becomes more complex. Moreover, we perform a second analysis step to find further problems and generate witness models able to signal such errors. In [66], the authors present a static fault localization technique to identify the rules responsible of transformation contracts violations. Problematic contracts are discovered via testing, and the guilty rules are identified by comparing the meta-model footprint of the contracts and the transformation rules. This technique is complementary to ours, although our implementation could be used to improve the precision of their footprints.

9.5 Static analysis of dynamic programming languages

Several techniques have been proposed in the last years to improve type safety of dynamic languages. For instance, dRuby [67] is an extension of Ruby in which type information is written as comments. The system is able to infer types from expressions, and checks them against handwritten signatures. Similar to our work, it supports union and intersection types. In [68], *nested refinements* are introduced as a logic for duck typed languages (e.g., Javascript, Ruby, etc.). Gradual typing [69] permits a language to be both statically and dynamically typed. Our approach is related to these typing techniques as our static analysis could generate the type information to create a new, typed version of the code (e.g., using DRuby annotations).

In [70], theorem proving is used to analyse Ruby on Rails applications and guarantee that database update operations do not break any data model invariant. However, the relevant updating code is extracted dynamically via code instrumentation. Some of our techniques could help in this regard, in particular, those to guarantee that the model generated by a transformation fulfils the constraints of its meta-model.

10 CONCLUSIONS AND FUTURE WORK

We have presented a novel technique for uncovering errors in model transformations and its application to ATL. The technique uses static analysis and type checking for finding problematic statements, and constraint solving to generate witness models confirming

and explaining the errors found. We have built a tool, which in average shows good performance and accurate results. Moreover, we have evaluated the tool on 101 public ATL transformations. We found that all of them have some issue, and uncovered a surprisingly high number of problems in most of them. This shows that tools like ours are a necessity for a proper engineering of model transformations.

Recently, we have extended our work with the possibility to associate quick fixes for the errors detected by our analyser [41], [42]. This provides further value to our techniques. In the future, we plan to classify the most common errors made by developers. This classification can be used to suggest features of ATL that are particularly error prone. We will also study possible correlations of transformation errors with meta-model features and metrics, to understand the meta-model characteristics which are more likely to yield transformation errors.

ACKNOWLEDGMENTS

This work was supported by the Spanish MINECO (TIN2014-52129-R and TIN2015-73968-JIN), and the R&D programme of the Madrid Region (S2013/ICE-3006). We are grateful to the referees for their accurate comments.

REFERENCES

[1] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, 2003.

[2] L. A. Rahim and J. Whittle, "A survey of approaches for verifying model transformations," *Software and System Modeling*, vol. 14, no. 2, pp. 1003–1028, 2015.

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

[4] OMG, "OCL 2.4 specification," http://www.omg.org/spec/OCL/2.4/.

[5] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1, pp. 31–39, 2008.

[6] F. Fleurey, B. Baudry, P.-A. Muller, and Y. L. Traon, "Qualifying input test data for model transformations," *Software and System Modeling*, vol. 8, no. 2, pp. 185–203, 2009.

[7] S. Sen, B. Baudry, and J.-M. Mottu, "Automatic model generation strategies for model transformation testing," in *ICMT'09*, ser. LNCS, vol. 5563. Springer, 2009, pp. 148–164.

[8] E. Guerra and M. Soeken, "Specification-driven model transformation testing," *Software and System Modeling*, vol. 14, no. 2, pp. 623–644, 2015.

[9] G. Snelting, T. Robschink, and J. Krinke, "Efficient path conditions in dependence graphs for software safety analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 4, pp. 410–457, 2006.

[10] M. Kuhlmann, L. Hamann, and M. Gogolla, "Extensive validation of OCL models by integrating SAT solving into USE," in *TOOLS (49)*, ser. LNCS, vol. 6705. Springer, 2011, pp. 290–306.

[11] QVT, http://www.omg.org/spec/QVT/.

[12] J. S. Cuadrado, E. Guerra, and J. de Lara, "Uncovering errors in atl model transformations using static analysis and constraint solving," in *ISSRE'14*. IEEE Computer Society, 2014, pp. 1–11.

[13] K. Muslu, Y. Brun, M. Ernst, and D. Notkin, "Reducing feedback delay of software development tools via continuous analyses," *IEEE Trans. Software Eng.*, vol. PP, no. 99, pp. 1–1, 2015.

[14] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Syst. J.*, vol. 45, pp. 621–645, July 2006. [Online]. Available: http://dx.doi.org/10.1147/sj.453.0621

[15] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu, "Barriers to systematic model transformation testing," *Commun. ACM*, vol. 53, no. 6, pp. 139–143, 2010.

[16] F. Jouault and J. Bézivin, *KM3: A DSL for Metamodel Specification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 171–185. [Online]. Available: http://dx.doi.org/10.1007/11768869_14

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[18] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[19] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *TACAS'07*, ser. LNCS, vol. 4424. Springer, 2007, pp. 632–647.

[20] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *TACAS'08*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[21] J.-M. Jézéquel, O. Barais, and F. Fleurey, "Model driven language engineering with kermeta," in *GTTSE'09*, ser. LNCS, vol. 6491. Springer, 2011, pp. 201–221.

[22] W3C, "WSDL 2.0 specification (primer)," http://www.w3.org/TR/wsdl20-primer/, 2007.

[23] REWERSE Working Group I1, "R2ML web page," http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=R2ML, 2006.

[24] M. Ribaric, D. Gasevic, M. Milanovic, A. Giurca, S. Lukichev, and G. Wagner, "Model-driven engineering of rules for web services," in *GTTSE'07*, ser. LNCS, vol. 5235. Springer, 2008, pp. 377–395.

[25] U. Dayal, "Ten years of activity in active database systems: What have we accomplished?" in *ARTDB'95*, ser. Workshops in Computing. Springer London, 1996, pp. 3–22. [Online]. Available: http://dx.doi.org/10.1007/978-1-4471-3080-2_1

[26] O. Agesen, "The cartesian product algorithm: Simple and precise type inference of parametric polymorphism," in *ECOOP'95*. Springer-Verlag, 1995, pp. 2–26.

[27] L. Larsen and M. J. Harrold, "Slicing object-oriented software," in *ICSE'96*. IEEE, 1996, pp. 495–505.

[28] J. S. Cuadrado, F. Jouault, J. G. Molina, and J. Bézivin, "Optimization patterns for ocl-based model transformations," in *Models in Software Engineering*. Springer, 2009, pp. 273–284.

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

[30] D. Jackson and C. Damon, "Elements of style: Analyzing a software design feature with a counterexample detector," *IEEE Trans. Software Eng.*, vol. 22, no. 7, pp. 484–495, 1996.

[31] J. Cabot, R. Clarisó, and D. Riera, "On the verification of UML/OCL class diagrams using constraint programming," *Journal of Systems and Software*, vol. 93, pp. 1–23, 2014.

[32] F. Büttner, M. Egea, and J. Cabot, "On verifying ATL transformations using 'off-the-shelf' SMT solvers," in *MoDELS'12*, ser. LNCS, vol. 7590. Springer, 2012, pp. 432–448.

[33] F. Büttner, M. Egea, J. Cabot, and M. Gogolla, "Verification of ATL transformations using transformation models and model finders," in *ICFEM'12*, ser. LNCS, vol. 7635. Springer, 2012, pp. 198–213.

[34] D. Calegari, C. Luna, N. Szasz, and Á. Tasistro, "A type-theoretic framework for certified model transformations," in *Formal Methods: Foundations and Applications*. Springer, 2011, pp. 112–127.

[35] E. Richa, E. Borde, and L. Pautet, "Translating atl model transformations to algebraic graph transformations," in *ICMT'15*, vol. 9152. Springer, 2015, p. 183.

[36] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[37] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[38] D. Wagelaar, R. V. D. Straeten, and D. Deridder, "Module superimposition: a composition technique for rule-based

model transformation languages," *Software and System Modeling*, vol. 9, no. 3, pp. 285–309, 2010.

[39] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa, "Dsltrans: A turing incomplete transformation language," in *SLE'10*. Springer, 2010, pp. 296–305.

[40] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science, WG'94*, ser. LNCS, vol. 903. Springer, 1994, pp. 151–163.

[41] J. S. Cuadrado, E. Guerra, and J. de Lara, "Quick fixing ATL model transformations," in *MoDELS'15*. IEEE, 2015, pp. 146–155.

[42] ——, "Quick fixing ATL transformations with speculative analysis," *Software and System Modeling*, vol. In press, 2016.

[43] J. S. Cuadrado, "Recursion and iteration support in use validator with anatlyzer," in *Workshop Proceedings of the 15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations*, 2015, p. 73.

[44] D. S. Kolovos, R. F. Paige, and F. Polack, "The Epsilon Transformation Language," in *ICMT'08*, ser. LNCS, vol. 5063. Springer, 2008, pp. 46–60.

[45] J. S. Cuadrado, J. G. Molina, and M. M. Tortosa, "RubyTL: A Practical, Extensible Transformation Language," in *ECMDA-FA'06*, ser. LNCS, vol. 4066. Springer, 2006, pp. 158–172.

[46] M. Lawley and J. Steel, "Practical declarative model transformation with tefkat," in *MoDELS'05*. Springer, 2005, pp. 139–150.

[47] S. Hildebrandt, L. Lambers, H. Giese, J. Rieke, J. Greenyer, W. Schäfer, M. Lauder, A. Anjorin, and A. Schürr, "A survey of triple graph grammar tools," *ECEASST*, vol. 57, 2013. [Online]. Available: http://journal.ub.tu-berlin.de/eceasst/article/view/865

[48] M. Amrani, B. Combemale, L. Lúcio, G. Selim, J. Dingel, Y. Le Traon, H. Vangheluwe, and J. R. Cordy, "Formal verification techniques for model transformations: A tridimensional classification," *The Journal of Object Technology*, vol. 14, no. 3, pp. 1–1, 2015.

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

[50] S. Sen, J.-M. Mottu, M. Tisi, and J. Cabot, "Using models of partial knowledge to test model transformations," in *ICMT'12*, ser. LNCS, vol. 7307. Springer, 2012, pp. 24–39.

[51] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann, "Formal specification and testing of model transformations," in *SFM'12*, ser. LNCS, vol. 7320. Springer, 2012, pp. 399–437.

[52] J. M. Küster and M. Abd-El-Razik, "Validation of model transformations - first experiences using a white box approach," in *MoDELS Workshops*, ser. LNCS, vol. 4364. Springer, 2006, pp. 193–204.

[53] C. A. González and J. Cabot, "ATLTest: A white-box test generation approach for atl transformations," in *MoDELS'12*, ser. LNCS, vol. 7590. Springer, 2012, pp. 449–464.

[54] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow, "Model transformations? Transformation models!" in *MoDELS'06*, ser. LNCS, vol. 4199. Springer, 2006, pp. 440–453.

[55] 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*, vol. 83, no. 2, pp. 283–302, 2010.

[56] F. Tip, "A survey of program slicing techniques," *J. Prog. Lang.*, vol. 3, no. 3, 1995.

[57] M. Weiser, "Program slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, 1984.

[58] Z. Ujhelyi, Á. Horváth, and D. Varró, "Dynamic backward slicing of model transformations," in *ICST'12*. IEEE, 2012, pp. 1–10.

[59] G. M. K. Selim, "Formal verification of graph-based model transformations," Ph.D. dissertation, Queen's University, 2015.

[60] B. J. Oakes, J. Troya, L. Lúcio, and M. Wimmer, "Full contract verification for atl using symbolic execution," *Software & Systems Modeling*, pp. 1–35, 2016.

[61] A. Rentschler and P. Sterner, "Interactive dependency graphs for model transformation analysis," in *Demos/Posters/StudentResearch@MoDELS*, ser. CEUR Workshop Proceedings, vol. 1115. CEUR-WS.org, 2013, pp. 36–40.

[62] M. van Amstel and M. G. J. van den Brand, "Model transformation analysis: Staying ahead of the maintenance nightmare," in *ICMT'11*, ser. LNCS, vol. 6707. Springer, 2011.

[63] J.-M. Mottu, S. Sen, M. Tisi, and J. Cabot, "Static analysis of model transformations for effective test generation," in *ISSRE'12*. IEEE, 2012, pp. 291–300.

[64] A. Vieira and F. Ramalho, "A static analyzer for model transformations," in *MtATL'11*, 2011.

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

[66] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo, "Static fault localization in model transformations," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 490–506, 2015.

[67] M. Seki, *The dRuby Book*. The Pragmatic Programmers, 2005.

[68] R. Chugh, P. M. Rondon, and R. Jhala, "Nested refinements: a logic for duck typing," in *ACM SIGPLAN Notices*, vol. 47(1). ACM, 2012, pp. 231–244.

[69] J. Siek and W. Taha, "Gradual typing for objects," in *ECOOP'07*. Springer, 2007, pp. 2–27.

[70] I. Bocić and T. Bultan, "Inductive verification of data model invariants for web applications," in *ICSE'14*. ACM, 2014, pp. 620–631.