Quick fixing ATL model transformations

Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara
Universidad Autónoma de Madrid

***Abstract*—The correctness of model transformations is key to obtain reliable MDE solutions. However, current transformation tools provide limited support to statically detect and correct errors. This way, the identification of errors and their correction are mostly manual activities. Our aim is to improve this situation.**

Based on a static analyser for ATL model transformations which we have previously built, we present a method and a system to propose quick fixes for transformation errors. The analyser is based on a combination of program analysis and constraint solving, and our quick fix generation technique makes use of the analyser features to provide a range of fixes, notably some non-trivial, transformation-specific ones. Our approach integrates seamlessly with the ATL editor. We provide an evaluation based on an existing faulty transformation, and automatically generated transformation mutants, showing overall good results.

***Index Terms*—Model Transformation, Transformation Static Analysis, ATL, Quick fixes, Verification and Testing.**

I. Introduction

Model transformation is key in Model Driven Engineering (MDE), as it enables automated model manipulations. Hence, methods to detect and correct transformations errors, as well as to speed up their construction are of great interest [17].

Many transformation languages and tools have been proposed along the years, and some like ATL [5] or ETL [6] are widely used in the MDE community. However, few (if any) have achieved the same level of maturity as supporting tools for general-purpose programming languages like Java. In this respect, missing features include static analysers that detect advanced typing and rule errors, as well as quick fix generators able to propose corrections to these problems.

The static guarantees that transformation languages provide varies. For instance, most QVT implementations statically type the transformation against the meta-models, but other languages such as ATL or ETL are dynamically typed. In that case, transformations are prone to typing errors that can only be discovered by thorough testing if no static analysis is applied. In all cases, facilities to fix typing errors may help to improve productivity and transformation quality. Other elusive errors that are important to detect and fix correctly include rule conflicts, target meta-model conformance, and whether rule guards cover all possible cases [3].

In previous work [3], we built a static analyser for ATL transformations, named anATLyzer[1], which is able to detect a wide number of typing errors (about 40 different types). The analyser is integrated with the standard ATL editor, so that errors can be detected interactively while the user is constructing the transformation. Using the analyser, we discovered that even transformations considered in a mature stage, like those in the ATL Use Cases[2], contain errors.

In this work, we have extended the analyser with the possibility to propose and apply quick fixes for the detected errors. Quick fixes can be used for autocompletion to speed up transformation development, or as a means to solve errors. Depending on the kind of error, quick fixes can suggest changes in the transformation (e.g., adding filters to rules or collections, refine the type of a variable), in the meta-model (e.g., set a feature cardinality to optional), or add transformation pre-conditions that prevent the transformation execution for problematic models.

We have evaluated our approach from two perspectives. First, we have tested the degree in which we can fix a faulty transformation developed by a third party (one from the ATL Use Cases). Second, we have tested the completeness of our quick fix catalogue by applying it to automatically generated transformation mutants. To the best of our knowledge, this is the first work proposing a catalogue of quick fixes for model transformations which can be used in practical tools.

Paper organization. Sec. II introduces types of quick fixes, and a running example. Sec. III explains our method for static analysis. Sec. IV presents our catalogue of quick fixes, classified according to a feature diagram. Sec. V describes our implementation, and Sec. VI its evaluation. Sec. VII discusses related research, and Sec. VIII ends with the conclusions.

II. Overview

Recommenders are increasingly being used to assist in different software engineering tasks [18]. In particular, code recommenders assist programmers with coding activities, like API usage or quick fixes. The actual recommendation may come from a mix of sources, like the static analysis of the program being developed, its execution, or the programmer himself [16]. In this work, we focus on quick fixes, where information is gathered via static analysis.

We define a *quick fix* as an automatable solution to a problem detected *statically*. Typically, a quick fix provides a rapid means to correct a problem reported by the IDE as the program (a transformation in our case) is developed. We found no explicit classification of quick fixes in the literature, but the following classification has suited our needs:

1) *Repair*. These quick fixes generate a fix that removes the targeted problem, typically adding or modifying expressions in certain locations, and without any additional input

[1] Available at http://www.miso.es/tools/anATLyzer.html

[2] http://www.eclipse.org/atl/usecases/, some of these transformations originated from industrial projects

from the developer. An example is a quick fix adding a condition to ensure that a null pointer exception cannot occur. Sometimes the application of this kind of quick fixes may introduce errors in other locations.

2) *Template*. This type of quick fix generates a piece of code solving a problem, but there may be missing information that is only initialized with default values, and the developer must add the logic to complete the generated code. For example, a program may refer to a non-existent helper, and the quick fix creates a template for it, which the user needs to fill with appropriate code.
3) *Heuristic*. This corresponds to a suggestion, e.g., proposing a valid name for a collection operation based on string similarity [2]. Unlike the first type of quick fix, these suggestions are among several possibilities, and their application normally implies just some replacement.

In practice, quick fixes are used in two ways: to correct errors or for code autocompletion. In the former scenario, the developer is reported a problem, and he applies one of the available quick fixes to solve it. For this purpose, repair and heuristic quick fixes are useful. In the latter scenario, the developer may even make the error on purpose (e.g., invoking a non-existing lazy rule) and he uses the proposed quick fix as a means to generate a template. This is the most common use of template quick fixes.

Quick fixes can also be classified according to the artefact they fix. In the context of model transformations, quick fixes may target the transformation implementation (the most common case), the involved meta-models, or the transformation specification by adding a transformation pre-condition. The latter two possibilities (fixing the transformation *contract*) are sometimes preferred over changing the implementation, as discussed in [15] in the context of object-oriented programs.

A. Running example

Next, we introduce a running example that will be used to present our quick fix generation techniques. Listing 1 shows an excerpt of the PNML2PetriNet transformation, from the *Grafcet to PetriNet* scenario in the ATL Zoo[3], slightly modified to illustrate a richer set of quick fixes. Fig. 1 shows the source and target meta-models of the transformation.

```
1  rule PetriNet {
2    from n : PNML!NetElement
3    using {
4      arcsSet : Set(PNML!NetContentElement) =
5                n.contents->select(e | e.oclIsKindOf(PNML!Arc));
6    }
7    to p : PetriNet!PetriNet (
8      elements <- n.contents,
9      arcs <- arcsSet
10   )
11 }
12
13 helper def : selectLabel(labels : Sequence(PNML!Label)) : String =
14   if labels->isEmpty() then 'no-name'
15   else labels->first().text endif;
16
17 rule Place {
18   from n : PNML!Place
19   to p : PetriNet!Place (
20     name <-thisModule.selectLabel(n.name.labels)
21   )
22 }
23
24 rule Transition {
25   from n : PNML!Transition
26   to  p : PetriNet!Transition
27 }
28
29 rule PlaceToTransition {
30   from n : PNML!Arc (
31     n.source.oclIsKindOf(PNML!Place) and
32     n.target.oclIsKindOf(PNML!Transition)
33   )
34   to p : PetriNet!PlaceToTransition (
35     name <- thisModule.selectLabel(n.name.labels),
36     "from" <- n.source,
37     "to" <- n.target
38   )
39 }
40
41 rule TransitionToPlace {
42   from n : PNML!Arc (
43     -- The developer forgets to add n.source.oclIsKindOf(PNML!Transition)
44     n.target.oclIsKindOf(PNML!Place)
45   )
46   to p : PetriNet!TransitionToPlace (
47     name <- thisModule.getLabel(n.name.labels),
48     "from" <- n.source, -- Problem here, n.source could be a Place
49     "to" <- n.target
50   )
51 }
```

Listing 1. Excerpt of the PNML2PetriNet ATL transformation.

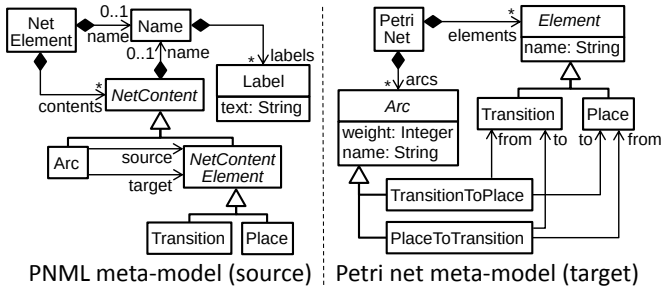

Fig. 1. Source and target meta-models of the transformation.

Our analyser reports the following problems, for which we show one illustrative quick fix. Other quick fixes are possible, as we will show in the following sections.

- **Declared type mismatch**. Our static analyser infers the type Sequence(Arc) for the expression in line 5, which is incompatible with the declared type Set(NetContentElement).
  Quick fix: change declared type by inferred type.
- **Possible unresolved binding** (lines 8–9). Some models may contain objects that appear in the right part of these bindings, but are not matched by any rule. In particular, both bindings will be unresolved for PNML documents that contain arcs linking places to places, or transitions to transitions. Although this is an incorrect Petri net, the PNML meta-model does not forbid these cases.
  Quick fix: add pre-condition to the transformation.
- **Invalid target for resolved binding** (lines 8 and 48). This problem arises when the rule matched by the right part of a binding generates an object of type incompatible with the feature where it is assigned. In line 8, the error is due to a missing filter in the right part of the binding.

[3]http://www.eclipse.org/atl/atlTransformations/

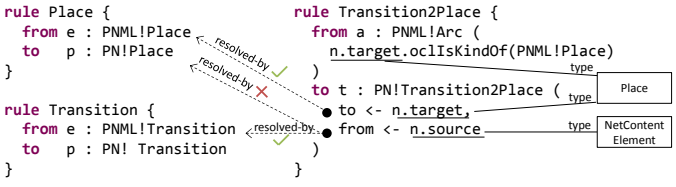

Fig. 2. Example of static analysis. Solid lines represent inferred type annotations. Dashed lines represent "resolved-by" binding-rule dependencies.

Quick fix: filter out Arc *objects from the binding. This also removes the "possible unresolved binding" problem.*

The binding of line 48 can be resolved by rule Place, since property n.source may hold either a place or a transition. *Quick fix: ensure that n.source holds a transition, for instance, checking this condition in the rule filter.*

- **Access to undefined value** (lines 20, 35 and 47). The name property is optional, so in case it holds an undefined value, it will cause a runtime exception. *Quick fix: change the cardinality in the meta-model.*
- **Compulsory feature not initialized** (line 26). Rule Transition creates a Transition object, but it does not initialize its mandatory attribute name. *Quick fix: generate a default value.*
- **Operation not found** (line 47). getLabel does not exist. *Quick fix: change the called operation to* selectLabel.

III. Transformation Analysis

Our system uses static analysis to identify problems that need to be fixed and gather the information required to implement the quick fixes. Next, we describe the main parts of our analyser and classify the problems it is able to detect.

A. Static analysis of ATL model transformations

Our static analyser proceeds in three steps [3]. First, it type-checks the transformation, annotating each node of the abstract syntax with its type. Then, it creates the *transformation dependence graph* (TDG), which makes control and data flow explicit, including information about rule resolution and rule dependencies. The TDG enables rule analysis, e.g. to determine unresolved bindings. However, some of the identified problems may not happen in practice, e.g., if the program logic prevents the error. In those cases, the analyser uses a model finder (USE Validator [7] in our current implementation) to confirm or discard the problem.

Fig. 2 shows an example of the analysis process. Initially, the type of the expression n.target in the filter of rule Transition2Place is NetContentElement, but the analyser detects the occurrence of oclIsKindOf and annotates the expression with the more refined type Place. In the second binding of this rule, the expression n.source has type NetContentElement. Using the TDG, the analyser infers that this type can be resolved by rules Place and Transition. However, resolving the binding by rule Place yields an incorrect target model, because reference from in class Transition2Place can only hold Transition objects, while rule Place creates Place objects.

To confirm whether the binding could actually be resolved by rule Place, the analyser uses the TDG to build an OCL path condition from the entry points of the transformation to the possible error location. This condition collects the features that an input model needs to have to make the transformation fail. Then, the analyser uses a model finder to search a model conformant to the input meta-model and satisfying the computed OCL path condition. If the finder finds a model, the error is confirmed. In the example, the entry point is rule Transition2Place, and we want to enforce binding from to be resolved by rule Place. Therefore, the needed OCL path condition is: Arc.allInstances()→exists(n | n.target.oclIsKindOf(Place) and n.source.oclIsKindOf(Place)), for which the model finder indeed finds a model, meaning that the error can occur in practice.

B. A taxonomy of errors in ATL model transformations

Fig. 3 summarizes kinds of problems detected by our analyser. These are classified into rule problems (which are the most specific to model transformations), style and optimization warnings, and object-oriented and OCL typing problems.

In the rest of the paper, we focus on quick fixes for transformation-specific errors (*Rule conflict*, *Unresolved binding*, *Rule resolution with invalid target*, *Feature initialization*) and on errors that typically appear in ATL transformations although they are not exclusive of ATL (*Invalid receptor*, *Declaration mismatch*, *Operation not found*).

IV. Generating Quick Fixes

Each kind of problem detected by the analyser has one or more associated quick fixes. Each quick fix comprises an optional application condition and an action. The application condition allows discarding the quick fix if the problem occurs in a context where it does not make sense or that the quick fix cannot handle. The action implements a strategy to fix the problem, which can be modifying the transformation, modifying the meta-model, or generating a transformation pre-condition. In all cases, the quick fix implementation can use the information gathered during the static analysis.

Fig. 4 shows a feature diagram with the available fixing strategies. Each one includes a label that is used to refer to the fix in a compact way. Possible transformation modifications include generating new expressions (a, c), adapting an existing expression to a new context (b), or modifying operation/feature calls (d). Rule-related problems are typically fixed by creating or removing rules (e, f), modifying their filters (g), creating or removing bindings (k), or modifying the right part of a binding (l). Other fixes may involve the creation of a new helper or a lazy/called rule[4] (h, i), or changing a reference to a type (m).

A. Catalogue of quick fixes

Fig. 4 classifies the quick fixes provided by our system, while Fig. 3 shows which quick fixes become applicable for each kind of error (the fixes are identified using the labels in Fig. 4). Table I summarizes some errors and their associated quick fixes. Due to space constraints, the table only shows some quick fixes applicable to error types 0–14 from Fig. 3. The full catalogue is available at http://miso.es/qfx.

[4]In practice, it is more natural to consider lazy/called rules as operations.

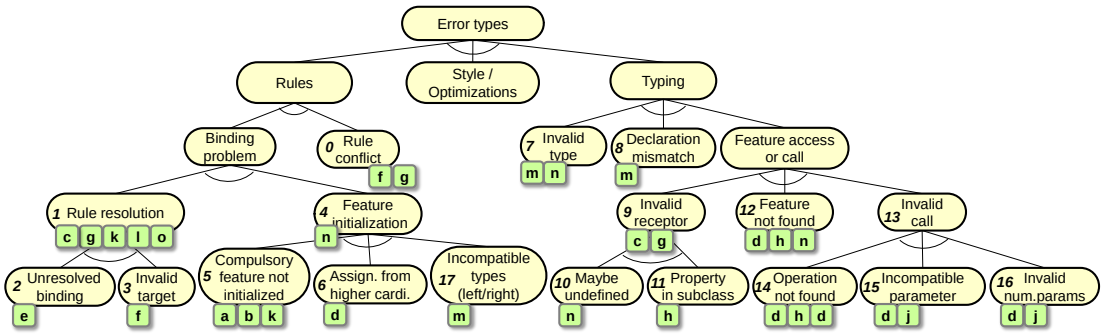

Fig. 3. Classification of typing/rule errors in ATL transformations. Labels *a–m* correspond to fixing strategies in Fig. 4. Numbers 1–14 are used in Table I.

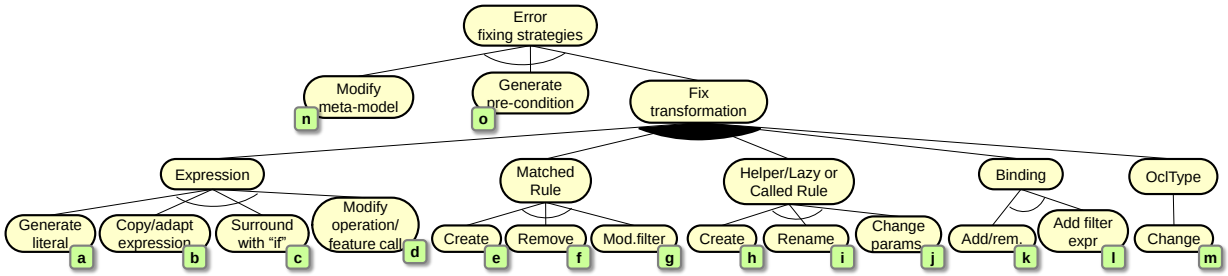

Fig. 4. Classification of error fixing strategies. Labels *a–m* are used in Fig. 3 to refer to the associated fixing strategy.

In Fig. 3 and Table I, we group fix strategies common to several error types in the common ancestors. For example, *Rule resolution* errors (E1) can be of type *possible unresolved binding* (E2) and *invalid target for resolved binding* (E3). Both error types share 5 fixing strategies (c, g, k, l, o), while each has a specific fixing strategy (e and f, respectively).

In the following subsections, we focus on the quick fixes more specific to transformations.

B. Rule resolution (E1, E2, E3)

Given a binding of the form feature ← expr, the ATL engine looks up in the trace model the source elements resulting from evaluating expr, and assigns their corresponding target elements to feature. This is called *binding resolution*. Two main problems may occur in this process, *possible unresolved binding* (E2) and *invalid target for resolved binding* (E3), which may cause performance penalties and invalid target models respectively. Both problems have the same fix strategies, except creating a rule to make the binding resolvable (Q2.1) and deleting the rule that is creating the invalid target element (Q3.1). Moreover, all quick fixes receive the following input: the type T_f of the feature, the expression $expr$ in the right part of the binding, the type T_{expr} of $expr$, and the set of rules R resolving T_{expr} and involved in the problem.

Q1.3: Add filter to binding expression. This strategy filters $expr$ to avoid the resolution of the problematic elements.

For the *possible unresolved binding* problem, the filter will select only the elements that will be certainly resolved by some rule. In this case, R is the set of rules able to resolve the binding. For example, in the problematic line 9 of Listing 1, we have $T_f = Sequence(Arc)$, $expr = arcsSet$, $T_{expr} = Sequence(Arc)$, and $R = \{TransitionToPlace, PlaceToTransition\}$.

Then, the quick fix proceeds as follows:

1) Group R by input type, yielding sequence G_r. This sequence is ordered by subtype relationships, where more concrete types go first. If a matched rule has more than one input type, it will not appear in R because ATL does not consider it as a candidate to resolve bindings.
2) Create a filter expression, $filter$, as follows. Take the head of G_r, and create an if expression whose condition checks the type given to the group, and the then branch is the *or*-concatenation of the rule filter expressions. The else branch applies the same procedure to the rest of G_r. When there are no more groups, the last else branch returns false.
3) If T_{expr} is a collection, modify expr to expr→select(v | filter(v)).
4) If T_{expr} is a single value, create let v = expr in if filter(v) then v else <default value> endif.

The quick fix for the problem in line 9 of Listing 1 is:

```
1 arcs <- arcsSet->select(v | if v.oclIsKindOf(PNML!Arc) then
2     v.target.oclIsKindOf(PNML!Place) or
3     (v.source.oclIsKindOf(PNML!Place) and
4      v.target.oclIsKindOf(PNML!Transition))
5   else false endif
```

This is so as both TransitionToPlace and PlaceToTransition are grouped by their common input type PNML!Arc. Then, lines 2-4 of the generated quick fix have the *or* of both rules filters, while the else branch returns false. Notice that, while this quick fix removes the problem, the developer is in charge of ensuring that it is semantically correct.

TABLE I
SOME QUICK FIXES. R: REPAIR, T: TEMPLATE, H: HEURISTIC.

Errors (E) and Quick fixes (Q)	Fix	Type
Rule conflict (E0)		
Q0.1 Modify guilty rules filter	g	R
Rule resolution (E1)		
Q1.1 Modify filter of container rule	g	R
Q1.2 Remove problematic binding	k	R
Q1.3 Add filter to binding expression	cl	R
Q1.4 Generate transformation pre-condition	o	R
Possible unresolved binding (E2)		
Q2.1 Create new rule	e	T
Invalid target for resolved binding (E3)		
Q3.1 Remove guilty rule	f	R
Feature initialization (E4)		
Q4.1 Modify feature (cardinality/type) in the mm.	n	R
Compulsory feature not initialized (E5)		
Q5.1 Initialize with default literal (e.g., empty string)	a	R
Q5.2 Copy and adapt an existing expression	b	H
Q5.3 Suggest mapping to a similar source feature	k	H
Assignment from higher cardinality (E6)		
Q6.1 Add ->first() to the collection	d	R
Invalid type (E7)		
Q7.1 Suggest a type of the meta-model	m	H
Q7.2 Add type to the meta-model	n	R
Declaration mismatch (E8)		
Q8.1 Change declared type for inferred type	m	R
Invalid receptor (E9)		
Q9.1 Surround problem with "if"	c	R
Q9.2 Modify filter of container rule	g	R
Possible access to undefined property (E10)		
Q10.1 Change the feature lower bound to 1	n	R
Access to a property defined in a subclass (E11)		
Q11.1 Create helper	h	T
Feature/operation not found (E12, E14)		
Q12.1 Suggest an existing feature/operation	d	H
Q12.2 Create a feature/operation helper	h	T
Q12.3 Create feature in the meta-model	n	T
Q12.4 Change feature call to operation call (and vice.)	d	R

For the *invalid target for resolved binding* error, the quick fix filters out the elements resolvable by rules that produce incompatible target objects. In this case, R is the set of *guilty* rules. The identification of guilty rules is similar to the mechanism proposed in [3]. Briefly, we generate a path condition for each rule that may potentially cause the problem, and use a model finder to produce a witness model satisfying each path condition. The rules for which a witness model is found are marked as guilty and added to R. The generated quick fix is similar to *possible unresolved binding*, except that the filter condition is negated.

Q1.1 Modify filter of container rule. This strategy avoids executing the container rule of a problematic binding for the objects that cause the problem. For this purpose, the rule filter is added (*and*-concatenated) an expression similar to the one of the previous strategy.

For example, in the *invalid target for resolved binding* problem of line 48, we have $T_f = Transition$, $expr = n.source$, $T_{expr} = NetContentElement$, and $R = \{Place\}$. To obtain set R, the static analyser first computes the set of possible guilty rules, with only one rule in this case, {Place}. Then, the model finder is used to discriminate which rules actually cause the problem, selecting Place. Applying the quick fix adds an additional condition to the filter of rule TransitionToPlace:

```
1 from n : PNML!Arc (
2     n.target.oclIsKindOf(PNML!Place) and
3     not n.source.oclIsKindOf(PNML!Place)
4 )
```

Actually, the previous algorithm would generate the expression let v = n.source in if v.oclIsKindOf(PNML!Place) then false else true endif, but we have an optimization for the cases when only one rule is involved.

Q1.2 Remove problematic binding. This quick fix is applicable when the lower cardinality of the feature is 0 in the meta-model.

Q1.4 Generate transformation pre-condition. Sometimes, the problem is not in the transformation itself, which is correct according to the developer assumptions concerning the source models. In such cases, applying this quick fix generates a transformation pre-condition that makes those assumptions explicit. This pre-condition serves as documentation, and in addition, it will be used to feed the model finder in subsequent invocations in order to discard problems the pre-condition rules out. Technically, pre-conditions are implemented as comments in the transformation header, prefixed with "@pre".

The generation process of pre-conditions uses a strategy similar to the generation of path conditions explained in Section III-A, but in this case, the path to the error is negated. The pre-condition generated for the problem in line 9 of Listing 1 is the following (provided the problem in line 48 has been fixed, as shown before):

```
1 not NetElement.allInstances()->exists(n |
2   n.contents->select(e | e.oclIsKindOf(Arc))->exists(a |
3     not (a.source.oclIsKindOf(Place) and a.target.oclIsKindOf(Transition)) or
4     not (a.source.oclIsKindOf(Transition) and a.target.oclIsKindOf(Place))))
5 or NetElement.allInstances()->isEmpty()
```

Although this pre-condition is not optimal in the sense of being as general as possible, it correctly states the fact that arcs in a Petri net must connect places to transitions or transitions to places, and any other configuration of arcs is invalid.

Q2.1 Create new rule (only for *possible unresolved binding*). This quick fix adds a new rule which complements the existing ones so that the binding never gets unresolved. The input pattern of the new rule uses the type of the binding expression, while the output pattern uses the type of the assigned feature.

If the resolving rules have filter conditions, such filters must be considered in the filter of the new rule, in order to avoid a rule conflict (i.e., two rules matching the same element).

Finally, if the type of the assigned feature is abstract, it cannot be used as output pattern. For our experiments, we heuristically select a non-abstract subclass that preferably is not used in any other rule. In practice, the system could allow selecting the appropriate type manually.

As an example, the quick fix for the *possible unresolved binding* problem in line 8 generates the following rule:

```
1 rule Arc2TransitionToPlace {
2   from a : PNML!Arc ( not
3     (a.source.oclIsKindOf(PNML!Place) and
4     (a.target.oclIsKindOf(PNML!Transition) or
5       a.target.oclIsKindOf(PNML!Place))) )
6   to t : PetriNet!TransitionToPlace ...
```

Q3.1 Remove guilty rule (only for *invalid target for resolved binding*). This quick fix removes the guilty rules, identified as explained above. Although this may result in subsequent unresolved bindings in other places, we do not check this situation in the application condition of the quick fix as it is too time consuming to be integrated in the IDE.

C. Rule conflicts

In ATL, if two or more rules match the same element, a rule conflict is raised and the transformation fails at runtime. Our static analysis is able to report this situation, identifying which sets of rules are in conflict.

Q0.1 Modify guilty rules filter. This quick fix is technically similar to Q1.1. In this case, given a set of guilty rules, we extend the filter of each rule by *and*-concatenating the negation of the other rules' filters. In this way, all rule filters are disjoint, ensuring that no rule conflict can arise.

D. Invalid receptor (E9, E10, E11)

This kind of problem appears when the receptor object of a feature access or operation call can be invalid. We distinguish two cases: access to an undefined value (i.e., a "null pointer exception"), and a special kind of *feature not found* problem in which the accessed feature is defined in a subclass of the receptor object's class, but it is missing in other subclasses.

Q9.1 Surround problem with "if". This quick fix surrounds the problematic expression with a conditional. Its if branch checks that the receptor object is not undefined (for E10) or its type has the accessed feature (for E11), while the else branch contains an appropriate default value according to the type of the expression (e.g., the empty string for a String).

The expression thisModule.selectLabel(n.name.labels) in lines 20 and 35 of Listing 1 may cause a runtime exception if n.name (the receptor object) is undefined. This quick fix will generate:

```
1 if not n.name.oclIsUndefined() then thisModule.selectLabel(n.name.labels)
2 else '' endif -- The type of the binding is 'String'
```

Q9.2 Modify filter of container rule. The underlying idea of this strategy is similar to Q1.1. If the problem appears within a binding, it is possible to avoid the problem by preventing the rule execution. In this case, the filter is modified to select a subclass that defines the feature.

This is a typical idiom in ATL, in which there are several, similar rules dealing with different variations (e.g., a rule to deal with objects for which certain property is undefined, and another rule dealing with the definite case).

Q10.1 Change the feature lower bound to 1 (only *possible access to undefined property*). The lower cardinality of the feature is set to 1 in the target meta-model. This may cause *compulsory feature not initialized* problems in other transformations using the same target meta-model, though these problems can be easily detected using our analyser.

E. Feature initialization (E4, E5, E6)

We focus on *compulsory feature not initialized* since it is the most common error in ATL transformations [3].

Q4.1 Modify feature cardinality in the meta-model. The lower cardinality of the feature is set to optional in the target meta-model. This may cause *possible access to undefined value* in other transformations using the same meta-model, or in the current transformation if it is endogenous.

Q5.1 Initialize with default literal. This is the simplest strategy, as we just generate a default value according to the feature type. For primitive types we generate the usual default values (e.g., an empty string, 0 for integers, etc.). For objects, we try to assign a target object of the compatible type that is the scope of the rule (i.e., other output pattern elements).

Q5.2 Copy and adapt an existing expression. This strategy relies on the same hypothesis as the GenProg system [10]: *"a program that makes a mistake in one location often handles a similar situation correctly in another"*.

In this way, we seek for bindings in other rules that assign the same feature, and for each candidate binding we check if the variables used in the right part of the binding are compatible with the variables accessible from the current rule. A variable in the current rule is compatible with a variable used in the candidate if the feature calls over the candidate variable can be performed over the rule variable. Currently, we select just one binding, copying and adapting it to the rule. However, all different bindings could also be shown to the user.

For example, in line 26, there is no binding for the name feature. However, such feature is assigned in lines 20, 35, and 47. Thus, the bindings name ← thisModule.selectLabel(n.name.labels) and name ← thisModule.getLabel(n.name.labels) could be proposed. Unfortunately, they are discarded as our analyser has detected problems in both bindings. Nevertheless, if we apply quick fixes Q9.1 or Q9.2, the problems in lines 20 and 35 become fixed, thus enabling this quick fix.

Q5.3 Suggest mapping to a similar source feature. This quick fix tries to find a feature, accessible from the rule input type, whose name is similar to the assigned feature and its type is compatible with the feature.

F. Declaration mismatch (E8)

This problem is frequent in ATL transformations developed without any static analysis tool like ours, due to ATL not checking neither statically nor dynamically variable, parameter and return type declarations. While this problem does not prevent the correct execution of a transformation, it is a maintenance problem because developers normally have expectations according to the declared types.

Q8.1 Replace declared type with inferred type. This quick fix replaces the declared type with the type inferred by the static analyser. In the example, the assignment in lines 4–5 is fixed to:

```
1 arcsSet : Sequence(Arc) = n.contents->select(e | e.oclIsKindOf(PNML!Arc))
```

In some cases, the type inferred by the analyser is richer than any type that can be expressed with the ATL type system. For example, when the branches of an if expression have types that are not related by inheritance. In such cases, we use the most general type OclAny.

G. Operation not found

ATL supports two types of helpers, *attribute helpers* and *operation helpers*. An operation helper may take parameters, while an attribute helper acts as a derived attribute, whose result is memoized. On the other hand, a helper can be defined in the context of a class, called *context helper*, acting as a regular polymorphic method, or without context, called *module helper*, acting as a global function. In addition, for practical purposes, we consider lazy and called rules to be module helpers since they are invoked like module helper operations with one or more parameters.

Q12.1 Suggest an existing feature/operation. We use the Levenshtein string distance [2] to look up candidate features (including both helper and meta-model features) and operations. In the case of operations, we take into account the types of the parameters to improve filtering. We also consider built-in functions, such as collection operations.

Q12.2 Create new context helper (*only calls over an object*). A new context helper is created, whose context is the inferred class for the receptor object of the call, and its parameters will be created according to the types of the actual arguments. The body of the helper is initialized to a default value according to its return type.

Q12.2 Create new module helper (*only calls over thisModule*). Given a call $thisModule.op(par_1, \ldots, par_n)$, the quick fix proposes adding a new helper, lazy or called rule. The heuristic to select each option is the following:

- **Lazy rule**. The problematic call is within a collect's body and belongs to the right part of a binding. Besides, the feature initialized by the binding must be a reference (i.e., a non-primitive type), the number of arguments of the call must be one, and the argument must be a single object.
- **Called rule**. Same as before, but either more than one parameter is passed, or just one parameter of primitive type (including collections).
- **Module helper**. Otherwise.

This heuristic reflects the most usual invocation patterns in ATL. In the example, the quick fix suggests creating a module helper for the call thisModule.getLabel(n.name.labels).

H. Summary

A developer working interactively with our system would be able to completely fix the running example using the following sequence of quick fixes: *Q8.1* to fix the declaration mismatch (lines 4–5), *Q1.1* to modify the rule filter and fix the problem in line 48, *Q1.3* to add a filter to the binding in line 8 so that arcs are filtered out (this solves problems E02 and E03 at once), *Q1.4* to add a pre-condition to prevent the invalid arcs causing the problem in line 9, *Q9.1* to avoid accessing an undefined value in lines 20, 35 and 47, *Q5.2* to initialize the name property in line 26, and *Q12.1* to replace the invalid call in line 47 by the one suggested.

V. Implementation

Our proposal is backed by an implementation atop anATLyzer, our static analyser for ATL, which is integrated with the Eclipse/ATL IDE. In particular, the quick fixes are available through the standard facilities provided by Eclipse, complemented with a dedicated view to easily inspect and fix detected problems (see Fig. 5).

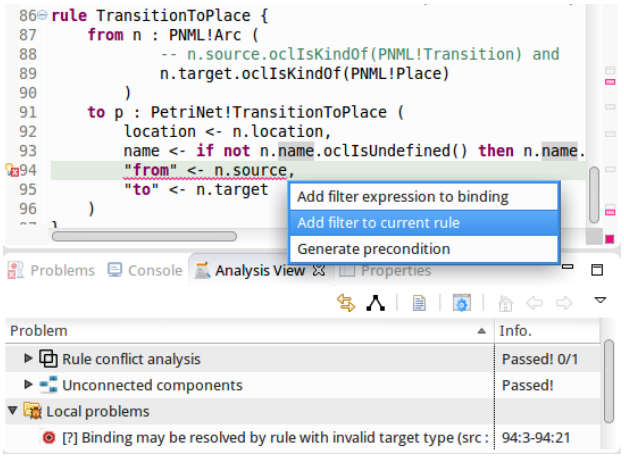

Fig. 5. Screenshot of the tool

A screencast of the tool, the complete results of our experiments, the source code and an Eclipse Update site are available at http://miso.es/qfx.

Implementation-wise, our quick fixes do not work at the text level, as standard Eclipse quick fixes do, but they modify the ATL abstract syntax using a dedicated API that we have built. This decision was motivated by the need of automatically applying quick fixes in our experiments, but it also opens the possibility of implementing advanced quick fix strategies like speculative analysis [13]. This posed several challenges, such as the need to build a variant of the ATL meta-model suitable to be easily manipulated using Java code, and the generation of the new code, which is performed by means of an incremental ATL serializer built as part of the framework.

Finally, our catalogue of quick fixes is extensible by means of an extension point and a set of pre-defined abstract quick fixes, along with the API to modify the ATL abstract syntax.

VI. Evaluation

This section reports on the evaluation of our system. We evaluate its completeness by generating mutants of a transformation in order to create a wide range of problems, and its usefulness by manually quick fixing a transformation written by a third-party. In both cases we analyse the amount of applicable quick fixes and their impact after their application, that is, the difference between the number of detected problems before and after the application of a quick fix. The evaluation is performed using the following automation script:

```
1  Input: transformation T, list of problems Lp
2  Step 1: Confirm or discard problems using the model finder
3          If discarded, remove problem from Lp
4  Step 2: Foreach P in Lp
5              Retrieve the set of available quick fixes for P
6              Foreach quick fix, Q
7                  Copy T into Tq
8                  Apply Q on Tq
9                  Run the static analyser on Tq, obtaining
10                     the problems, Lpq, remaining after Q
11                 Confirm or discard problems from Lpq, as in step 1
12                 Compare the size of Lp and Lpq
```

TABLE II
MUTATION OPERATORS FOR ATL TRANSFORMATIONS.

Type	Targets
Deletion	rule, helper binding source/target pattern element rule filter rule inheritance relation operation context formal/actual parameter in operation or called rule 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), Type.allInstances())
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

A. Evaluating completeness

In order to evaluate the completeness of our quick fix catalogue (i.e., does it cover a range of different problems?) we have performed an experiment based on using mutations to generate possibly faulty transformations, and then measure how many quick fixes are applicable and their impact in terms of the number of fixed and newly generated problems.

For the experiment, we have selected the original *PNML2PetriNet* transformation, fixing its errors manually with our quick fixes, so that it is error free. Then, we have applied the mutation operators shown in Table II, one at a time. For each mutation, the generated transformation is likely to have only one problem, or if it has more they are likely independent.

In total, we obtained 304 mutated transformations, out of which 56 were discarded because the analyser did not detect any problem (due to false negatives or because the mutation did not introduce errors). Table III summarizes the results. For each problem detected by the analyser, we show the number of occurrences in all the mutated transformations (*#Occ.*), the number of applied quick fixes (*#Qfx.*), the average number of quick fixes per transformation (*Avg*) and the minimum/maximum number of quick fix proposals in the transformations. Columns *Fix.* and *Gen.* show the number of fixed and newly generated problems after applying the quick fix.

The number of quick fixes ranges normally between 2 and 3 for most of the problems, which indicates that our catalogue covers an acceptable range of situations. We checked the main causes of failure by manual inspection.

Rule conflicts (E00) do not arise frequently. In all cases our strategy fixes the problem, but it tends to generate additional *possible unresolved bindings* because the modified rules filter cover less cases. For binding resolution problems (E02 and E03), most quick fixes do not generate additional problems. This is due to the use of the model finder to confirm the problem, which provides detailed information to generate accurate fixes. On the contrary, creating a new rule (Q2.1) produces many errors because of uninitialized compulsory features. Deleting guilty rules (Q3.1) sometimes yields a *possible unresolved binding*.

Regarding invalid receptor problems, several *null pointer exceptions* are generated (E10). Fixes based on changing the meta-model (Q10.1) and fixes based on adding a conditional (Q9.1) work well. However, adding a rule filter (Q9.2) is introducing several binding resolution issues.

Regarding quick fixes related to operation and feature calls, we have implemented the most usual ones found in IDEs for object-oriented languages. Quick fixes that create helpers or rules (Q12.2) tend to generate additional problems because we only generate a generic body or because of missing bindings, and thus they are useful as autocompletion facilities. Similarly, we could not evaluate the application of Q12.3 automatically, because it requires user intervention to set the new feature type. Finally, Q12.1 is not very accurate because there are few helpers to choose in the transformation, and our system tends to suggest ATL built-in operations, for which we have not considered yet whether their types fit well in the context of the replaced expression.

TABLE III
ERRORS DETECTED IN THE MUTATED TRANSFORMATIONS AND FIXES

Problem	#Occ.	#Qfx	Avg	Min	Max	Fix.	Gen.
E00	7	7	1.0	1	1	6	7
Q0.0	-	7	-	-	-	6	7
E02	55	140	2.5	1	4	111	150
Q1.2	-	27	-	-	-	28	0
Q1.3	-	29	-	-	-	17	0
Q1.4	-	29	-	-	-	38	0
Q2.1	-	55	-	-	-	28	150
E03	36	114	3.2	2	5	119	19
Q1.1	-	24	-	-	-	18	8
Q1.2	-	9	-	-	-	10	0
Q1.3	-	36	-	-	-	30	0
Q1.4	-	36	-	-	-	52	0
Q3.1	-	9	-	-	-	9	11
E05	56	148	2.6	2	3	156	0
Q4.1	-	56	-	-	-	56	0
Q5.1	-	56	-	-	-	64	0
Q5.2	-	36	-	-	-	36	0
E06	9	9	1.0	1	1	9	0
Q6.1	-	9	-	-	-	9	0
E07	2	4	2.0	2	2	0	0
Q7.1	-	2	-	-	-	0	0
Q7.2	-	2	-	-	-	0	0
E08	10	10	1.0	1	1	12	0
Q8.1	-	10	-	-	-	12	0
E10	28	84	3.0	3	3	57	44
Q10.1	-	28	-	-	-	28	0
Q9.1	-	28	-	-	-	24	0
Q9.2	-	28	-	-	-	5	44
E12/E14	92	162	1.8	0	3	57	7
Q12.1	-	67	-	-	-	18	3
Q12.2	-	64	-	-	-	39	4
Q12.3	-	31	-	-	-	0	0

B. Evaluating usefulness

Next, we analyse to what extent a real world transformation can be fixed in practice using our proposal. To this end, we have selected the CPL2SPL transformation from the ATL Zoo. It is a moderately complex transformation, consisting of 15 rules and 4 helpers, and it has already been used to analyse fault localization techniques [1].

Table IV summarizes the errors detected by the analyser and the proposed fixes. The listing below shows an excerpt of two rules to give an impression of the transformation.

TABLE IV
ERRORS DETECTED IN CPL2SPL AND APPLIED FIXES

Problem	#Occ.	#Qfx	Avg	Min	Max	Fix.	Gen.
E02	11	37	3.4	1	5	50	34
Q1.1	-	2	-	-	-	0	0
Q1.2	-	8	-	-	-	14	0
Q1.3	-	8	-	-	-	0	0
Q1.4	-	8	-	-	-	36	0
Q2.1	-	11	-	-	-	0	34
E05	49	98	2.0	2	2	2354	0
Q4.1	-	49	-	-	-	2305	0
Q5.1	-	49	-	-	-	49	0
E10	7	21	3.0	3	3	56	1
Q10.1	-	7	-	-	-	49	0
Q9.1	-	7	-	-	-	7	0
Q9.2	-	7	-	-	-	0	1

```
1 rule CPL2Program {
2  from s : CPL!CPL
3  to ..., d : SPL!Dialog (
4         methods <− Sequence {s.incoming, s.outgoing} ) }
5 rule SubAction2Function {
6   from s : CPL!SubAction
7   to t : SPL!LocalFunctionDeclaration (
8       statements <− s.contents.statement ) }
```

There are 48 problems due to the location target attribute not being initialised. This attribute is used by TCS to maintain traceability between text and model elements. The best option in this case is to accept quick fix Q4.1 to make the feature optional in the meta-model, since it is not mandatory in TCS, and there is no similar feature in CPL that can be mapped into. There is another uninitialized feature (branches), which is made optional in the meta-model since it is not compulsory according to the SPL specification and its concrete syntax.

Then, there are 3 bindings similar to the one in line 4, but there is no rule to resolve them. These are 3 clear cases for *new rule* (Q2.1). The other 8 *possible unresolved problems* are similar to the one in line 8. The statement reference may hold objects of a subclass not handled by the transformation, and therefore a pre-condition is generated (Q1.4).

Finally, there are 7 *possible access to undefined value*, due to the contents feature being optional (line 8). The most obvious quick fix is Q9.1 to surround the problem with an if. However, the default value for the else branch will not be adequate, as an object must be assigned to the statements feature. Therefore, we should create a lazy rule and invoke it to create a proper default object. We could fix the meta-model as well (Q10.1), but in this case it does not seem adequate according to the semantics of the CPL language. By inspecting a related transformation (XML2CPL) we realised that this transformation expects contents not to hold an undefined value. Hence, the most adequate action is to create a pre-condition.

In summary, we have been able to fix a transformation with 67 problems by applying 13 quick fixes (2 Q4.1, 3 Q2.1, 8 Q1.4), and with the manual completion of 3 new rules and the creation of one pre-condition.

C. Threats to validity

The main threat to internal validity is that our analyser may report some false positives (i.e., indicate an error incorrectly). In this case, the quick fix will be applied to a correct statement yielding an unknown result. It may also have false negatives (i.e., fail to report a true error) which may yield an incorrect counting of fixed errors. In our experience, the analyser has a low rate of false positives (a preliminary evaluation is presented in [3]). For false negatives, we do not have a proper evaluation yet. Finally, the model finder used by our implementation (USE) is based on the "small scope hypothesis", limiting the search of models to the given scope. To minimise the number of false negatives due to this, we have used reasonably wide searching scopes.

Related to the previous threat, our system relies on anATLyzer to statically spot faults and it uses the TDG to build the quick fixes. At the same time, the counting of newly generated problems by a quick fix is performed using anATLyzer as the oracle. Unfortunately, to our knowledge, there is no other analyser of ATL with which cross-validate the results.

Another threat to internal validity is that mutations could be biased towards the generation of errors for which we have available quick fixes. To limit this issue, the mutation operators were developed independently.

The main threat to external validity is that we have used only one transformation in each experiment. Another threat is that we only cover ATL. The features of other transformation languages may limit or impose additional constraints in some quick fixes. However, provided that a suitable static analysis phase is available, quick fixes related to OCL and model navigation are directly applicable to OCL-based transformation languages, such as QVT or ETL. Languages such as ETL and RubyTL could also benefit from rule related quick fixes.

D. Discussion

Quick fixes for rule resolution may generate large filter expressions because they aggregate the types and filters of several rules. For some cases, we have optimizations that generate more compact expressions, as in the running example, where we generate the filter not n.source.oclIsKind(Place) with Q1.1. However, a more natural filter is n.source.oclIsKindOf(Transition). In recommender systems terms, our method is not able to generalize, in the sense that the generated code just targets the detected error, and does not consider the future cases that may arise (e.g., adding a subclass of NetContentElement breaks the current fix). Another view of this issue is that our system tends to generate *alien code*, which is not adequate for a human-targeted repair system as ours [12]. Improving this issue is part of our future work.

In comparison with established quick fix frameworks, such as Eclipse JDT or Intelli/J, our proposal would be classified as a recommendation system for model transformations. However, some of the problems that the analyser detects are bugs (i.e., problems that manifest themselves at runtime provoking an incorrect behaviour of the transformation), though they are uncovered statically. This is particularly the case of rule conflicts, binding resolution and invalid receptor problems. Hence, part of our proposal can be seen as a lightweight form of automatic program repair.

Furthermore, as shown in Section VI-B, the detected problems and the proposed quick fixes prove to be useful to reason

about the correction of the meta-models, allowing fixing them when necessary. The generation of pre-conditions is useful to document which cases are not handled by a transformation. Moreover, our system uses these pre-conditions to provide accurate problem reports (i.e., it does not show a rule problem if the pre-condition guarantees that will not happen).

Regarding the performance of the system, it is responsive enough to be used as an editing facility. The use of the model finder does imply a bottleneck since the search is limited to the *error meta-model*, as discussed in [3].

Finally, our evaluation shows that some quick fixes are more likely to solve problems than others. This could be a starting point to rank quick fixes, and show the "best quick fixes" more prominently to the user.

VII. Related work

Strategies for proposing and ranking quick fixes have been studied in the programming community. For example, in [13], quick fixes are ranked according to the number of errors remaining after their application. However, there are few works dealing with this topic in the MDE literature.

Solutions for quick fix generation have also been applied to Domain-Specific Modelling Languages (DSMLs). For example, [4] uses design-space exploration to propose quick fixes for DSMLs. A quick fix is defined as a set of model operations that reduces the number of errors. The authors propose some guidelines for quick fix generation, like ranking quick fixes by their simplicity (offer first those with less model modifications). In our case, errors are detected by static analysis, and quick fixes implement pre-defined correction strategies and may introduce new problems.

In [8], [19], the authors present a taxonomy of common pitfalls in QVT-R transformations. Some of these errors are detected by executing the transformation using Petri nets. In our case, errors are detected statically and we provide a suite of quick fixes to amend them.

In [20], a catalogue of refactorings for model-to-model transformations is presented. We believe our method can be applicable to the automated refactoring of ATL transformations, but we leave this aspect for future work.

Automated program repair [9] aims at correcting faulty programs, where faults are detected by the dynamic testing of the programs. The work [11] is close to our proposal, but for an object-oriented language. Autofix [15] uses pre/post-conditions and invariants to synthetize repairs. While most program repair techniques are time consuming (as e.g., they evaluate possible patches by random testing), our goal is to integrate quick fix proposals in the development process, which requires lighter, quicker techniques that do not involve executing the transformation using a test suite.

VIII. Conclusions and Future Work

We have presented a method based on static analysis and constraint solving to generate quick fixes for ATL transformations and a catalogue of such quick fixes. The synthetic evaluation based on analysing and fixing mutated transformations has shown that our proposal covers a wide range of problems, and that the quick fixes actually fix most of the problems. The manually performed case study has shown the usefulness of our proposal. In addition, the implementation of the tool and the detailed results of the evaluation are available.

As future work, we plan to use speculative analysis [14] to improve the accuracy of the quick fixes. To improve the recommendation aspect of the system, we aim at creating a ranking of quick fixes that may be modified at run-time by taking into account previously selected quick-fixes by the user. We also plan to perform a controlled experiment with users to provide stronger evidence of the usefulness of the approach.

Acknowledgements. Work supported by the Spanish MINECO (TIN2011-24139 and TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), and the EU commission (FP7-ICT-2013-10, #611125).

References

[1] L. Burgueno, J. Troya, M. Wimmer, and A. Vallecillo. Static fault localization in model transformations. *IEEE TSE*, 41(5):490–506, 2015.

[2] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string metrics for matching names and records. In *KDD workshop on data cleaning and object consolidation*, volume 3, pages 73–78, 2003.

[3] J. S. Cuadrado, E. Guerra, and J. de Lara. Uncovering errors in ATL model transformations using static analysis and constraint solving. In *ISSRE*, pages 34–44. IEEE, 2014.

[4] Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró. Quick fix generation for DSMLs. In *IEEE VL/HCC*, pages 17–24. IEEE, 2011.

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

[6] D. S. Kolovos, R. F. Paige, and F. Polack. The epsilon transformation language. In *ICMT*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

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

[8] A. Kusel, W. Schwinger, M. Wimmer, and W. Retschitzegger. Common pitfalls of using QVT relations - graphical debugging as remedy. In *ICECCS*, pages 329–334. IEEE, 2009.

[9] C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.

[10] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE TSE*, 38(1):54–72, 2012.

[11] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA*, pages 133–146. ACM, 2012.

[12] M. Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *ICSE*, pages 234–242, 2014.

[13] K. Muslu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. In *OOPSLA*, pages 669–682. ACM, 2012.

[14] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices*, 47(10):669–682, 2012.

[15] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE TSE*, 40(5):427–449, 2014.

[16] S. Proksch, S. Amann, and M. Mezini. Towards standardized evaluation of developer-assistance tools. RSSE, pages 14–18. ACM, 2014.

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

[18] M. P. Robillard, R. J. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

[19] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Right or wrong? - verification of model transformations using colored petri nets. In *DSM*, 2009.

[20] M. Wimmer, S. Perez, F. Jouault, and J. Cabot. A catalogue of refactorings for model-to-model transformations. *JOT*, 11(2):1–40, 2012.