Quick fixing ATL transformations with speculative analysis

Jesús Sánchez Cuadrado[⋆], **Esther Guerra, Juan de Lara**

Universidad Autónoma de Madrid (Spain),
e-mail: {Jesus.Sanchez.Cuadrado, Esther.Guerra, Juan.deLara}@uam.es

Abstract Model transformations are central components of most model-based software projects. While ensuring their correctness is vital to guarantee the quality of the solution, current transformation tools provide limited support to statically detect and fix errors. In this way, the identification of errors and their correction are nowadays mostly manual activities which incur in high costs. The aim of this work is to improve this situation.

Recently, we developed a static analyser that combines program analysis and constraint solving to identify errors in ATL model transformations. In this paper, we present a novel method and system that uses our analyser to propose suitable quick fixes for ATL transformation errors, notably some non-trivial, transformation-specific ones. Our approach supports *speculative analysis* to help developers select the most appropriate fix by creating a dynamic ranking of fixes, reporting on the consequences of applying a quick fix, and providing a previsualization of each quick fix application.

The approach integrates seamlessly with the ATL editor. Moreover, we provide an evaluation based on existing faulty transformations built by a third party, and on automatically generated transformation mutants, which are then corrected with the quick fixes of our catalogue.

Key words Model transformation, ATL, Transformation static analysis, Quick fixes, Speculative analysis.

1 Introduction

Model transformation is one of the cornerstones of Model-Driven Engineering (MDE), as it enables the automation of model manipulations. Hence, methods to detect and correct transformation errors, as well as to speed up the construction of transformations, are of great interest for MDE practitioners [33].

Many transformation languages and tools have been proposed along the years, and some like ATL [15] or ETL [18] are widely used by the MDE community. However, they have not 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, quick fix generators able to propose corrections to these errors, and tools that help understanding the consequences of applying a correction.

The static guarantees that transformation languages provide vary. For instance, most QVT [32] implementations statically type the transformation against the source/target meta-models, but other languages such as ATL or ETL are dynamically typed. In the latter case, transformations are prone to typing errors, like accessing a feature that is defined on a subtype of the receptor object's type, mistakes in type declarations, or navigation through possibly null references. Other elusive errors that are important to detect and fix include rule conflicts (i.e., rules with overlapping applicability conditions, which cause errors in languages like ATL), unresolved or incorrectly resolved bindings, and conformance errors of the generated output models with respect to the target meta-model (e.g., uninitialized mandatory features) [35]. Nowadays, in most cases, these errors have to be discovered by manual testing, which is a costly activity with the risk to be incomplete. Instead, fault localization using static analysis is an automatic, lighter technique, while facilities to fix typing errors may help to improve the developer productivity and the transformation quality.

In previous work [35], we built a static analyser for ATL transformations, named anATLyzer[1], which is able to detect a wide number of typing and rule errors (about 45 different types). The analyser is integrated with the

Send offprint requests to:
[⋆] *Present address:* Computer Science Department, Universidad Autónoma de Madrid, 28049 Madrid (Spain)

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

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 extend the analyser with the possibility to propose and apply quick fixes for the detected errors. Quick fixes can be used for autocompletion in order to speed up transformation development, or as a means to correct existing errors. Depending on the kind of error, quick fixes may suggest changes in the transformation (e.g., adding filters to rules or collections, or refine the type of a variable), in the meta-model (e.g., setting a feature cardinality to optional), or add transformation pre-conditions that prevent the transformation execution for problematic models. In this way, quick fixes proposed for an error can be selected and applied interactively.

In case of errors that can be fixed in several ways, we provide a *static* ranking which shows first the quick fixes that we have found empirically to solve more errors and introduce less issues. In addition, to help developers make better decisions when several possibilities exist, and to understand the consequences of applying a fix, we use speculative analysis [3,28]. This term was coined by the programming languages community in analogy to *speculative execution*, e.g., for branch prediction and cache pre-fetching in program execution. It consists in analysing the possible future states of the program evolution (a transformation in our case), with the purpose of gathering information about remaining or introduced errors by a quick fix. This analysis is presented to the developer who can use it to perform more informed decisions when applying a quick fix.

We have evaluated several aspects of our approach. First, we have tested the completeness and validity of our quick fix catalogue by applying it to a large set of transformation mutants automatically synthesized from existing third-party transformations. The aim of this experiment is twofold: (i) to evaluate the degree in which there are quick fixes *applicable* for every error found, and (ii) to study how the quick fix *application* impacts the quality of the transformation. The latter is analysed by inspecting whether the quick fix actually solves the targeted error, does (not) produce additional issues, or solves other problems as a side effect. Second, we have empirically collected the efficacy of each quick fix (i.e., errors solved vs. issues introduced) to create a *static* ranking of quick fixes for every error. Then, we compare such a static ranking with a *dynamic* ranking produced by speculative analysis, taking as a basis the "optimal" quick fix selected by ATL experts. This experiment is performed over a set of faulty transformations developed by third parties.

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.

This paper extends our previous work [36] with the following contributions: we enlarge our catalogue of quick fixes, including variants and refinements of previously existing ones; we give a detailed account of all of these quick fixes and illustrate them with comprehensive examples; we support speculative analysis; we present a more precise experimental evaluation; we provide a static ranking of fixes which has been determined empirically; we provide a dynamic ranking of fixes created on-demand using speculative analysis; and we compare the dynamic and static rankings.

The rest of this paper is organized as follows. First, Section 2 introduces a classification and conceptualization of quick fixes and a running example. Then, Section 3 explains our method for static analysis. Section 4 presents our catalogue of quick fixes classified according to a feature diagram, while Section 5 analyses their impact and introduces our speculative analysis technique. Section 6 describes our implementation, and Section 7 its evaluation. Section 8 discusses related research, and finally, Section 9 ends with the conclusions and lines of future work.

2 Overview and Running Example

In this section, we provide an overview of quick fixes and introduce a running example that will be used in the rest of the paper to illustrate our catalogue of quick fixes.

2.1 Quick fixes: An initial classification

Recommenders are increasingly being used to assist in different software engineering tasks [34]. In particular, code recommenders assist programmers with coding activities, like API usage or the application of 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 [31]. In this work, we focus on quick fixes, where information is gathered via static analysis of the ATL transformation.

We define a *quick fix* as an automatable solution, and readily applicable, 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 categories have suited our needs:

1. *Repair.* These quick fixes remove the targeted problem, typically adding or modifying expressions in certain locations, and without any additional input from the developer. An example is a quick fix adding a condition to ensure that a navigation expression cannot

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

go through a null reference. In some cases, the application of this kind of quick fixes may introduce errors in other locations. For example, a quick fix changing the type of a helper's formal parameter[3] to make it compatible with the actual parameter of an existing helper call (e.g., from integer to string), may produce an error in other calls that were coherent with the original helper definition.

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 transformation 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 [6]. Unlike the first type of quick fix, these suggestions are provided heuristically among several possibilities, and their application normally implies just some replacement.

In practice, quick fixes are used in two ways: either to correct errors or for code autocompletion. In the former scenario, the developer is reported a problem and applying one of the available quick fixes solves the problem. In this case, repair and heuristic quick fixes are most useful. In the case of code autocompletion, the developer may even make the error on purpose (e.g., invoking a non-existing lazy rule) and the proposed quick fix application generates a template that the developer later completes manually. This is the most common use of template quick fixes.

2.2 Conceptual overview of our approach

Figure 1 shows the conceptual view of our system. We use this figure to provide an overview of our approach and as a guide to read the rest of the paper.

For each kind of problem detected by our static analyser, there are zero or more associated quick fixes. Section 4 introduces our catalogue of quick fixes, which can be extended easily via extension points (see Section 6). 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 classified according to the affected artefact. In the context of model transformation, quick fixes may target the transformation implementation (the most common case), the involved source/target metamodels, 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 [29] for object-oriented programs. A classification of

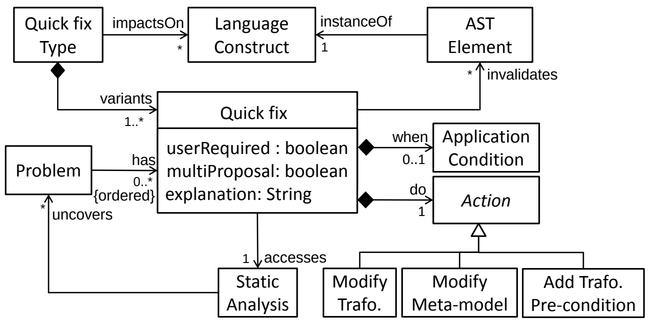


Fig. 1 Conceptual model of our proposal for quick fixes.

Quick fixes have access to the information gathered during the static analysis to implement the application condition and the action. Section 3 provides an overview of the static analysis process. Some quick fixes, like those of type template, may require the user intervention to complete the generated code. Sometimes, a single type of quick fix may provide several proposals, which is typically the case of heuristic fixes that suggest the best-rated solutions to the user. The user is provided with an explanation of the behaviour of the quick fix, which can be as simple as a single line or more elaborated. There are also fixes with variants, where each variant solves a problem in the same way, but generating code in a different manner. For example, a quick fix may generate an in-line expression, or alternatively, it may encapsulate the expression in a helper that is invoked.

The local changes performed by a quick fix may impact on other locations of the transformation. Our speculative analysis identifies at runtime the abstract syntax elements affected by a change, whereas the language constructs likely impacted by a fix application can be identified empirically (see Section 7.2). Section 5 introduces our speculative analysis technique, which analyses the consequences of applying available quick fixes for a given problem. We use this analysis to produce a dynamic ranking of fixes (hence the *ordered* annotation in Figure 1) according to the number of problems solved/remaining after the quick fix application. In addition, we support a lighter way to order quick fixes without resorting to speculative analysis. This consists on a default static order of applicable fixes, derived empirically from the automated fixing of automatically mutated transformations. This static ranking is presented in Section 5.3, and its comparison with the dynamic one is discussed in Section 7. The static ranking is intended to provide reasonable accuracy without delay time, whereas speculative analysis provides richer information that includes

[3] In ATL, a helper is an auxiliary query operation.

a previsualization of the quick fix result, but it requires some computation time.

2.3 Running example

To illustrate our quick fix generation techniques, we will use excerpts of a transformation from UML Activity Diagrams (AD) to Intalio BPMN[4], partially based on the mappings introduced in [5]. Figure 2 contains relevant snippets of the input and output meta-models for this transformation.

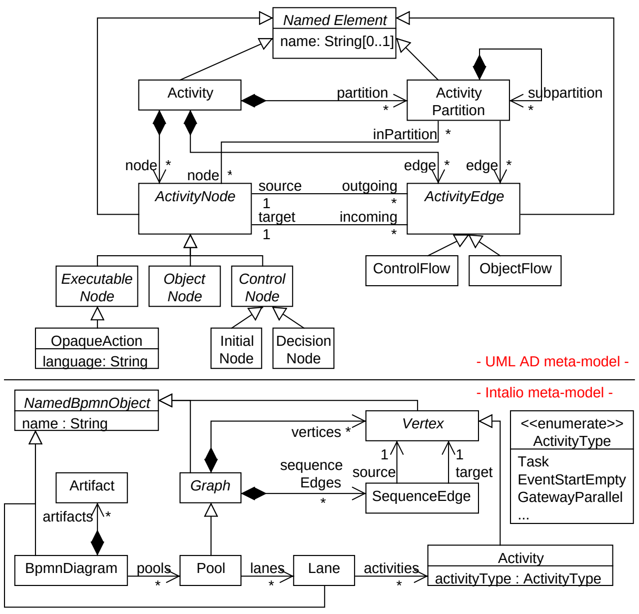


Fig. 2 Excerpts of AD (up) and Intalio (down) meta-models.

Listing 1 shows an excerpt of the transformation, consisting of three context helpers (lines 1-11) and five matched rules (lines 13-50). A context helper is an auxiliary operation defined in the context of a class, and can be invoked on instances of that class. A matched rule is executed for every input object that matches the input pattern specified in the from part of the rule and satisfies the filter (if specified). For example, in line 37, rule initialnode matches any object compatible with type UML!InitialNode and having an empty incoming reference. Each rule execution creates the objects indicated in the rule's output pattern (to part), together with trace links to the originating input objects. The features of the created objects are initialized according to the declared bindings, using the syntax *feature* ← *OclExpr*. For example, lines 39 and 40 initialize the features name and activityType of the created object a of type Intalio!Activity.

If the feature is a reference, a mechanism called binding resolution takes place by looking up the input objects resulting from *OclExpr* in the collection of trace links, in order to retrieve the corresponding target objects. In lines 47 and 48 of the example, references source and target of the created SequenceEdge are assigned an object of type UML!ControlFlow.source and UML!ControlFlow.target respectively. As these types belong to the source meta-model, the binding resolution mechanism takes place. The type of both UML!ControlFlow.source and UML!ControlFlow.target is UML!ActivityNode, which can be transformed by rules opaqueaction (line 29) and initialnode (line 36). Both rules create objects compatible with Intalio!Vertex, and hence, the resolution mechanism yields correct object types for the bindings in lines 47 and 48.

```
 1 helper context UML!Action def:
 2    toIntalioName : Intalio!Activity =
 3      self.name + '_' + self.oclType().name;
 4
 5 helper context UML!Activity def:
 6    allPartitions : Sequence(UML!Activity) =
 7      self.partition→collect(p | p.allPartitions)→flatten();
 8
 9 helper context UML!ActivityPartition def:
10    allPartitions : Sequence(UML!ActivityPartition) =
11      self.subpartition→collect(p | p.allPartition)→flatten();
12
13 rule activity2diagram {
14    from a : UML!Activity
15    to   d : Intalio!BpmnDiagram (
16      name ←  a.name,
17      pools ←  a.allPartitions
18    )
19 }
20
21 rule activitypartition2pool {
22    from a : UML!ActivityPartition
23    to p : Intalio!Pool,
24       l : Intalio!Lane (
25         activities ←  a.node→reject(e | e.oclIsKindOf(UML!ObjectNode))
26       )
27 }
28
29 rule opaqueaction {
30    from n : UML!OpaqueAction
31    to a : Intalio!Activity (
32      name ←  n.toIntalio
33    )
34 }
35
36 rule initialnode {
37    from n : UML!InitialNode ( n.incoming→isEmpty() )
38    to a : Intalio!Activity (
39      name ←  n.toIntalio,
40      activityType ←  #EventStartempty
41    )
42 }
43
44 rule edges {
45    from f : UML!ControlFlow
46    to e : Intalio!SequenceEdge (
47      source ←  f.source,
48      target ←  f.target
49    )
50 }
```

Listing 1 Excerpt of the transformation from UML AD to Intalio. Errors are shown underlined.

Listing 1 contains several errors (shown underlined), none of which are detected at compile time by the standard ATL IDE. Typically, these errors may remain unnoticed until the developer executes the transformation

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

with an input model making the transformation hit the problematic statement. Instead, our analyser statically detects and 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** (lines 2 and 6). Our static analyser infers the type String for the helper toIntalioName, which is incompatible with the declared type Activity. Similarly, the inferred type for the Activity.allPartitions helper (line 5) is Sequence(ActivityPartition), which differs from the declared type Sequence(Activity).
Quick fix: change declared type by inferred type. By applying this quick fix, the helper toIntalioName would be assigned the return type String, and Activity.allPartitions would be assigned Sequence(ActivityPartition), solving the problems.
– **Possible access to undefined value** (line 3). The name property is optional in class NamedElement, so in case it holds an undefined value, it will cause a runtime exception when applying the + operator.
Quick fix: change the cardinality in the meta-model. By applying this quick fix, the lower cardinality of attribute name would change from 0 to 1. This ensures that this attribute will never be undefined, solving the problem.
– **Compulsory feature not initialized** (lines 23 and 24). Rule activitypartition2pool creates objects of types Pool and Lane, but it does not initialize their mandatory attribute name.
Quick fix: generate a default value. By applying this quick fix to the error in line 23, the binding name ← '' would be added to the created Pool, providing a default value for name and solving the issue. Applying the quick fix to the error in line 24 would solve the problem for the created Lane.
– **Possible unresolved binding** (lines 25, 47 and 48). This issue is signalled when the right part of a binding may contain objects not matched by any rule. For example, the OCL expression a.node → reject(...) in line 25 may contain objects that are not considered by the transformation. In particular, objects that are instances of any subtype of ActivityNode except InitialNode and OpaqueAction, like DecisionNode in Figure 2, would not be transformed by any rule. The same problem applies to the bindings of lines 47 and 48. These errors are a smell of incompleteness that should be either fixed or documented.
Quick fix 1: add pre-condition to the transformation. For the binding in line 25, this quick fix generates an OCL pre-condition that discards models in which ActivityPartition objects contain objects different from InitialNode and OpaqueAction in its node reference.
Quick fix 2: add rule filter. For the bindings in lines 47 and 48, another option would be to generate a rule filter, disabling the rule execution for ControlFlow objects connecting ActivityNodes different from InitialNode and OpaqueAction.
– **Feature not found** (lines 32 and 39). The invoked feature toIntalio does not exist, either in the meta-model or as an attribute (context) helper.
Quick fix: change invocation to the toIntalioName attribute helper. This heuristic quick fix uses different string comparison criteria to find a suitable proposal. In this case, it uses the longest common substring criterion [6] to suggest suitable feature/helper names.
– **Enum not found** (line 40). Enum literal #StartEventempty cannot be found in the meta-model.
Quick fix: change to #StartEventEmpty. In this case, the most optimal proposal is found using the Levenshtein string distance criterion [6].

3 Transformation Analysis

Our system uses static analysis to identify problems and gather the information required to implement the quick fixes. This section describes the main parts of our analyser and classifies the problems it is able to detect. Further information can be found in [35].

3.1 Static analysis of ATL model transformations

Our static analyser proceeds in three steps, as shown in Figure 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), a kind of program dependence graph [9] which makes control and data flow explicit and includes information about rule resolution and rule dependencies. The TDG is the basis to analyse the behaviour of rules and bindings (e.g., to determine unresolved bindings) and to detect rule conflicts. 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 tries to find a witness model that makes the transformation execute the problematic statement, and hence confirming (or falsifying if it does not exist) the problem. Our current implementation relies on the USE Validator [19] model finder to perform this search.

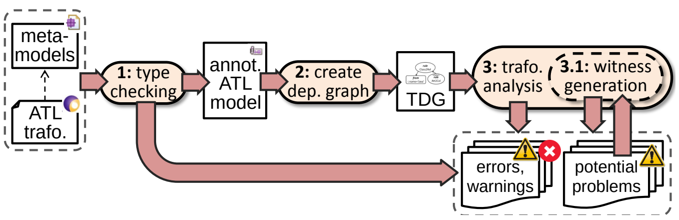
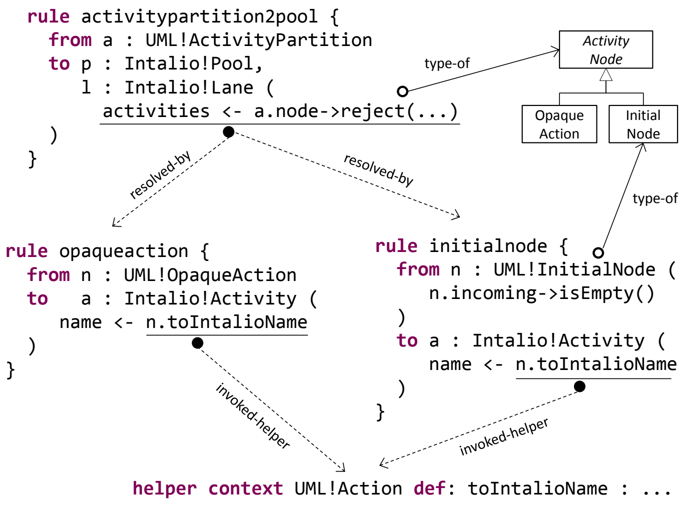


Fig. 3 Overview of static analysis process.

Figure 4 shows an example illustrating the analysis process. Each node of the OCL expressions involved in the transformation is annotated with its inferred type. For instance, the type of a.node → reject(...) is a collection of ActivityNode, and thus, the corresponding AST node is annotated with a reference to this metaclass (*type-of* in the figure). Given this information, we build the TDG to make the data and control dependencies between the transformation elements explicit. Figure 4 shows two relationships recorded by the TDG, *invoked-helper* and *resolved-by*. For the former, the calls to toIntalioName[5] are linked to the helpers that may resolve the calls at runtime (only one helper in this case). For the latter, given a binding, we compute all matched rules that may resolve it at runtime. In the example, the binding activities ← a.node → reject(...) may be resolved by either the opaqueaction or the initialnode rules, since their from parts are subtypes of ActivityNode, the type inferred for the right part of the binding.



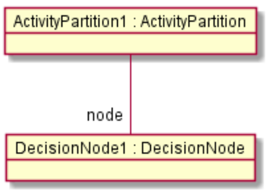
```
rule activitypartition2pool {
  from a : UML!ActivityPartition
  to p : Intalio!Pool,
     l : Intalio!Lane (
      activities <- a.node->reject(...)
    )
}

rule opaqueaction {
  from n : UML!OpaqueAction
  to   a : Intalio!Activity (
     name <- n.toIntalioName
  )
}

rule initialnode {
  from n : UML!InitialNode (
     n.incoming->isEmpty()
  )
  to a : Intalio!Activity (
     name <- n.toIntalioName
  )
}

helper context UML!Action def: toIntalioName : ...
```

Fig. 4 Example of static analysis. Solid lines (labelled as "type-of") represent inferred type annotations. Dashed lines represent "resolved-by" binding-rule dependencies and "invoked-helper" call-helper dependencies.

This information enables rule–binding analysis, for instance, to detect which bindings may be unresolved. In general, this type of problems cannot be fully confirmed by the type checker, but are marked as *potential problems*. In this example, we are interested in determining if the binding a.node → reject(...) may be unresolved for some instances of ActivityNode. We use the TDG to build an OCL path condition [35] 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 hit the given location and produce a failure. Then, the analyser uses a model finder to search a model conformant to the input metamodel 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 activitypartition2pool, which directly leads to the possibly faulty binding. To assert whether the right part of the binding may contain objects *not* resolved by any rule, we need the following OCL path condition, which looks for a model with an ActivityPartition containing some node that is neither an OpaqueAction nor an InitialNode without incoming edges:

```
1  ActivityPartition.allInstances()→exists(a |
2    a.node→reject(e | e.oclIsKindOf(ObjectNode))→exists(n |
3      not n.oclIsKindOf(OpaqueAction) and
4      not (if n.oclIsKindOf(InitialNode) then
5              n.incoming→isEmpty()
6           else
7             false
8           endif)))
```

When fed into the model finder, it produces the *witness model* in Figure 5, which confirms that the error can occur in practice. This model satisfies the OCL constraint as the ActivityPartition object contains a DecisionNode object in its node reference, and hence, it satisfies the condition in lines 3–8 of the OCL path condition.



ActivityPartition1 : ActivityPartition
node
DecisionNode1 : DecisionNode

Fig. 5 Witness model confirming the "possibly unresolved binding" error in line 25 of Listing 1.

3.2 A taxonomy of errors in ATL transformations

Our analyser is currently able to detect about 45 different types of errors. Figure 6 shows a feature model summarizing the most important kinds of problems detected, none of which is reported by the standard ATL IDE. The problems are classified into rule problems (which are the most specific to model transformations), style and optimization warnings, and object-oriented and OCL typing problems.

Rule errors may occur due to conflicts with other rules (label 0 in the figure), or due to binding problems. Rule conflicts arise if two different rules can match the same source object, causing a runtime exception. Binding problems may be related to rule resolution (label 1), either because the binding is unresolved (label 2) or because it is resolved with an invalid target object (label 3). The ATL resolution mechanism for bindings replaces the source objects by the target ones in which they were transformed. If there is no rule to transform the source objects, they are discarded but incurring in an execution penalty, and probably being the smell of a deeper issue. A related problem occurs when there is a rule to resolve the binding, but it produces target objects which are not compatible with the target feature.

[5] We have fixed Listing 1 for this figure.

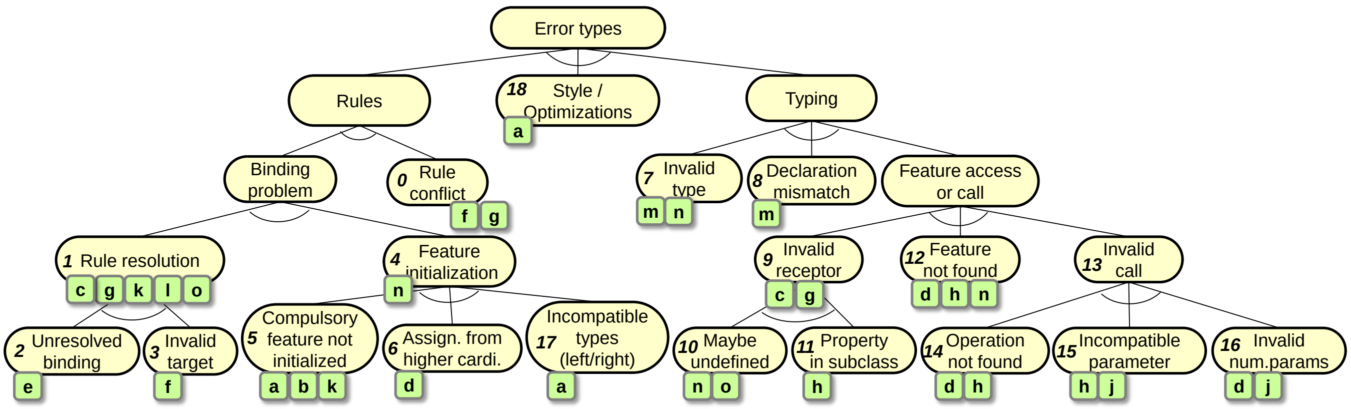


Fig. 6 Classification of typing/rule errors in ATL transformations. Labels *a–o* correspond to fixing strategies in Figure 7. Numbers *1–18* are used in Table 1.

Another source of binding problems is related to feature initialization (label 4), which may occur if a mandatory feature of a target object is not initialized (label 5), a feature is initialized from a collection with higher cardinality (label 6, e.g., a feature with maximum cardinality 1 is initialized from a collection with cardinality `*`), or a feature is assigned an incompatible value (label 17, e.g., a `String` feature is assigned an `Integer` value).

Style/optimization problems include iterating over empty collections, using "." instead of "→" to apply a collection operator (supported by ATL but not conforming to the OCL standard style), or invoking a `flatten` operation over a non-nested collection, among others [35].

Typing problems include referring to a non-existing type (label 7) like an undefined class name, type declaration mismatches (label 8) where the declared type for an expression does not correspond to the real type of the expression, or problems with feature accesses or operation/rule calls. The latter case can be due to an invalid receptor object (label 9), which may be undefined (label 10) causing a null pointer exception, or the accessed property may not belong to the receptor object but to a subtype (label 11). Other sources of errors include using incorrect feature names (label 12) or making invalid calls (label 13). The latter problem includes the use of incorrect operation names (label 14), incompatible parameter types (label 15) or an incorrect number of parameters (label 16).

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

4 A Catalogue of Quick Fixes for ATL

Each kind of problem detected by our static analyser has one or more associated quick fixes. Each quick fix follows a particular fixing strategy. The set of strategies that we have considered are summarized in Figure 7. The figure shows a feature diagram in which each strategy includes a label that is used to refer to the fix strategy in a compact way. These labels are used in Figure 6 to depict which quick fixing strategies become applicable for each kind of error.

In general, fixings may involve modifying the meta-model (label *n* in Figure 7), creating or modifying an OCL transformation pre-condition (label *o*) or modifying the transformation itself. Possible transformation modifications include generating new expressions (*a*), adapting an existing expression to a new context (*b*), restricting the applicability of expressions (*c*), or modifying operation/feature calls (*d*). Rule-related problems are typically fixed by creating or removing rules (*e*, *f*), modifying rule 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 lazy/-called rule[6] (*h*, *i*), or changing a reference to a type (*m*).

Table 1 contains the current list of quick fixes in our catalogue, and the errors to which they apply. In this table, and in Figure 6, we group all fix strategies common to several error types in their common ancestor. 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 five fixing strategies (*c, g, k, l, o*), while each one of them has a specific fixing strategy (*e* and *f*). Although the table does not show it, some of these quick fixes have variants, e.g., regarding how a generated expression is inserted in the transformation (in-lined or encapsulated in a helper).

We explain our quick fixes in the next subsections, with special focus on those more specific to transformations. We use the quick fix codes in the table to identify each quick fix, and indicate quick fix variants adding a suffix to their code.

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

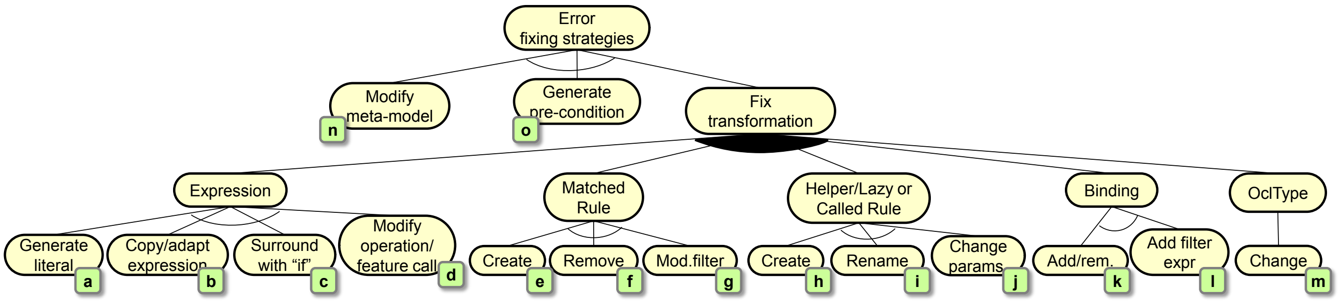


Fig. 7 Classification of error fixing strategies. Labels *a*–*m* are used in Figure 6 to refer to the associated fixing strategy.

Table 1 Catalogue of quick fixes. Labels *a*–*o* in column *Fix Str.* correspond to fixing strategies in Figure 7. The *Type* column uses R for Repair, T for Template and H for Heuristic.

Errors (E) and Quick fixes (Q)	**Fix Str.**	**Type**
Rule conflict (E0)		
Q0.1 Modify guilty rules filter	g	R
Q0.2 Remove one guilty rule	f	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	c, l	R
Q1.4 Generate transformation pre-condition	o	R
Q1.5 Generate most general 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
Q3.2 Choose a different target feature	k	H
Feature initialization (E4)		
Q4.1 Modify feature (cardinality/type) in MM	n	R
Compulsory feature not initialized (E5)		
Q5.1 Assign default value (e.g., empty string)	a	R
Q5.2 Copy and adapt 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 collection	d	R
Invalid type (E7)		
Q7.1 Suggest a type from meta-model	m	H
Q7.2 Add type to meta-model	n	R
Declaration mismatch (E8)		
Q8.1 Change declared type with inferred type	m	R
Invalid receptor (E9)		
Q9.1 Surround problem with "if"	c	R
Q9.2 Modify filter of container rule	g	R
Q9.3 Generate transformation pre-condition	o	R
Possible access to undefined property (E10)		
Q10.1 Change feature lower bound to 1	n	R
Access to property defined in subclass (E11)		
Q11.1 Create helper	h	T
Feature/operation not found (E12, E14)		
Q12.1 Suggest existing feature/operation	d	H
Q12.2 Create context/module helper	h	T
Q12.3 Create feature in the meta-model	n	T
Q12.4 Change feature call to operation call, and vice versa	d	R
Q12.5 Convert receptor to collection	d	R
Incompatible parameter (E15)		
Q15.1 Create new helper operation	h	T
Q15.2 Change type of formal parameters	j	R
Invalid number of parameters (E16)		
Q16.1 Add/remove actual parameters	d	H
Q16.2 Add/remove formal parameters	j	R
Q16.3 Choose other operation	d	H
Incompatible types (E17)		
Q17.1 Assign value with correct type to feature	a	R
Style warnings (E18)		
Q18.1 Correct invalid expression	a	R

4.1 Fixing rule resolution errors (E1, E2, E3)

Given a binding of the form feature ← expr, the binding resolution mechanism looks up in the trace model the source elements resulting from evaluating expr, and assigns their corresponding target elements to feature. Two main problems may occur in this process: *possible unresolved binding* (E2) and *invalid target for resolved binding* (E3). The former is a smell of incompleteness and may cause performance penalties because the ATL engine needs to check type compatibility and it will output error messages when it cannot assign the source element to the target feature. The latter problem may yield invalid target models. Both problems have five fix strategies in common (Q1.1 to Q1.5). In addition, *possible unresolved binding* can be fixed by creating a rule that makes the binding resolvable (Q2.1), and *invalid target for resolved binding* can be fixed by deleting the rule that is creating the invalid target element (Q3.1) or choosing a different target feature for the binding (Q3.2). In all cases, the quick fixes make use of the following input from the static analyser:

- T_f: type of the feature in the left part of the binding,
- *expr*: expression in the right part of the binding,
- T_{expr}: inferred type of *expr*,
- R: set of rules that resolve T_{expr} and are involved in the problem.

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

For the *possible unresolved binding* problem, the filter selects 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 25 of Listing 1, we have $T_f = Sequence(Intalio!Activity)$, $expr = a.node \rightarrow reject(e|e.oclIsKindOf(UML!ObjectNode))$, $T_{expr} = Sequence(UML!ActivityNode)$, and $R = \{opaqueaction, initialnode\}$.

Then, the quick fix proceeds as follows:

1. Group R by input type, yielding sequence G_r. This sequence contains sets of rules and is ordered by subtype relationships (with sibling types given arbitrary order), where groups of rules with more concrete types take precedence in the sequence. 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. (optimization) Simplify the conditionals in filter by omitting checkings when the input type of the rule is the same as the type of the right-hand side of the binding T_{expr}.
4. If T_{expr} is a collection, modify expr to expr→select(v | filter(v)).
5. If T_{expr} is a single value, create let v = expr in if filter(v) then v else <default value> endif.

Applying this quick fix to the problem in line 25 of Listing 1 creates the sequence of rules $G_r = \langle\{opaqueaction\}, \{initialnode\}\rangle$. This is so as the input types of the rules in R (UML!OpaqueAction and UML!InitialNode) are not related by inheritance and hence they are listed in arbitrary order in the sequence. The resulting binding is therefore:

```
1  activities ← a.node→reject(e | e.oclIsKindOf(UML!ObjectNode))→
2    select(v |
3      if v.oclIsKindOf(UML!OpaqueAction) then
4        true        −− implicit filter of opaqueaction
5      else
6        if v.oclIsKindOf(UML!InitialNode) then
7          v.incoming→isEmpty()      −− filter of initialnode
8        else
9          false
10       endif
11     endif
12   )
```

The condition in line 3 comes from the input type of rule opaqueaction, and line 4 contains true because the rule has no filter. The condition in line 6 is added due to the input type of rule initialnode, while line 7 is created due to the filter of the rule. Notice that, while this quick fix removes the problem, the developer is in charge of ensuring that it is semantically correct.

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 [35]. 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.

Q1.3b: Add filter to binding expression (variant). The previous quick fix effectively solves the problem, but makes the binding expression more complex. This variant of quick fix Q1.3 creates a helper with the generated filter to enhance the readability of the binding.

Using this variant, we would extract the filter created for the binding to the following context helper:

```
helper context UML!ActivityNode def:
  resolveActivitypartition2poolActivityNode: Boolean =
    if self.oclIsKindOf(UML!OpaqueAction) then
      true
    else
      if self.oclIsKindOf(UML!InitialNode) then
        self.incoming→isEmpty()
      else
        false
      endif
    endif;
```

While the binding would be rewritten as follows:

```
activities ← a.node→reject(e | e.oclIsKindOf(UML!ObjectNode))→
  select(v | v.resolveActivitypartition2poolActivityNode )
```

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. However, this quick fix is applicable only when the right part of the binding is mono-valued. For multi-valued expressions, we do not prevent the rule execution altogether on the basis that a few elements of the expression may cause a problem; in this case, we prefer using other available quick fixes.

For example, suppose we modify the running example as follows to include the rule objectnode (we also copy rule edges from the initial listing for clarity):

```
rule objectnode {
  from n : UML!ObjectNode
  to a : Intalio!Artifact
}

rule edges {
  from f : UML!ControlFlow
  to e : Intalio!SequenceEdge (
    source ← f.source,
    target ← f.target
  )
}
```

In such a case, our analyser reports *invalid target for resolved binding* for the two bindings in rule edges. This is so as the right part of these bindings can be resolved by rule objectnode, but the type created by rule objectnode is not compatible with the type of the target features source and target. Thus, for the problem in the first binding, we have $T_f = Intalio!Vertex$, $expr = f.source$, $T_{expr} = UML!ActivityNode$, and $R = \{objectnode\}$. To obtain set R, the static analyser first computes the set of possible guilty rules, which are three this case ({opaqueaction, initialnode, objectnode}). Then, the model finder is used to discriminate which rules actually cause the problem, selecting only objectnode. Applying the quick fix adds an additional condition to the filter of rule edges:

```
from f : UML!ControlFlow (
  not f.source.oclIsKindOf(UML!ObjectNode)
)
```

Actually, the previous algorithm would generate the expression let v = f.source in if v.oclIsKindOf(UML!ObjectNode) then false else true endif, but we have an optimization for the cases that involve only one rule.

Q1.2 Remove problematic binding. This quick fix, which simply removes the problematic binding, is only applicable when the lower cardinality of the feature of the target object is 0 in the meta-model. In the error *possibly unresolved binding* of line 25 in Listing 1, the lower cardinality of Lane.activities is zero, therefore, this quick fix is applicable.

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. For example, suppose that a *possible unresolved binding* error is notified because some source type is not matched by any rule, but the developer knows that the input models will never contain objects of this type, and hence, the error will never occur in practice. In such cases, applying this quick fix generates a transformation pre-condition that makes those assumptions (in this example, the lack of objects of certain type) 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. In practice, pre-conditions are implemented as comments prefixed with "@pre" in the transformation header, and are processed by anATLyzer to feed the model finder.

The generation process of pre-conditions uses a strategy similar to the generation of path conditions explained in Section 3.1, but in this case, we must ensure that every element that "goes through" the path satisfies exactly one of the input patterns of the resolving rules. The pre-condition generated for the problem in line 25 of Listing 1 is the following:

```
UML!ActivityPartition.allInstances()→forAll(a |
  a.node→reject(e | e.oclIsKindOf(UML!ObjectNode))→forAll(v |
    v.oclIsKindOf(UML!OpaqueAction) or
    if ( v.oclIsKindOf(UML!InitialNode) ) then
          v.incoming→isEmpty()
    else
          false
    endif))
```

This pre-condition forbids witness models as the one shown in Figure 5. It states the allowed shape of the ActivityPartition objects, and hence solves the problem as well.

Q1.5 Generate most general pre-condition. The pre-condition generation method explained above uses the whole path of the fixed problem. This makes the pre-condition too problem-specific, which means that it is unlikely that it fixes similar problems appearing in other rules. Section 5 explores this issue in more detail. An alternative is to use only the last part of the path to generate the pre-condition. Hence, the most general pre-condition generated for the problem in line 25 of Listing 1 is as follows:

```
UML!ActivityNode.allInstances()→forAll(v |
  v.oclIsKindOf(UML!OpaqueAction) or
  if ( v.oclIsKindOf(UML!InitialNode) ) then
    v.incoming→isEmpty()
  else
    false
  endif)
```

This pre-condition is more general than the previous one, where the restriction does not apply to the ActivityPartition.node collection, but more generally to all ActivityNode objects. This pre-condition style aims at documenting the kind of elements that are not supported by the transformation.

Q1.5b Generate meta-model restriction (variant). This variant, instead of generating a transformation pre-condition, generates an OCL invariant for the meta-model. This fix should be selected if the user considers that the restriction is not specific to the transformation but to the source meta-model in general. The invariant is generated in the context of T_f, the class type of the binding. Hence, in the example, the following invariant would be generated:

```
context ActivityNode inv constraint_activitypartition2pool:
   self.oclIsKindOf(OpaqueAction) or
   if ( self.oclIsKindOf(InitialNode) ) then
     self.incoming−>isEmpty()
   else
     false
   endif
```

It can be seen that the generation is similar to that for helpers.

Q2.1 Create new rule (only for *possible unresolved binding*). This template quick fix adds a new rule that 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, the output pattern uses the type of the assigned feature, and the rule filter takes into account the filter conditions of the resolving rules in order to avoid a rule conflict (i.e., two rules matching the same object). Note that if the type of the assigned feature is abstract, it cannot be used as output pattern; in that case, we heuristically select a non-abstract subclass that preferably is not used in any other rule. An alternative implementation could allow selecting the specific type manually.

As an example, the quick fix for the *possible unresolved binding* problem in line 25 generates the following rule:

```
rule restOfActivityNode2Activity {
  from n : UML!ActivityNode (
    not n.oclIsKindOf(UML!OpaqueAction) and
    not (if n.oclIsKindOf(UML!InitialNode) then
       n.incoming→isEmpty()
    else
       false
    endif)
  )
  to a : Intalio!Activity
}
```

The generated rule matches any ActivityNode not matched by opaqueaction or initialnode, and creates an Activity, which is the kind of object required by line 25. Being a template quick fix, it requires being completed by providing a value for the mandatory features of the created object a (e.g., name). This error can be eliminated manually or by applying another quick fix. As in previous quick fixes, a variant of this quick fix generates the filter condition in a helper.

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 and this will cause a delay when showing the list of available quick fixes. Instead, we use speculative quick fixes to deal with these cases (see Section 5).

Q3.2 Choose a different target feature (only for *invalid target for resolved binding*). This is a heuristic quick fix that changes the target meta-model feature assigned in the binding for an existing feature whose type is compatible with some guilty rule. As an example, let us consider that we have a rule to transform an ObjectNode into an Artifact (lines 1–4 in the following listing) and we are completing the activitypartition2pool rule to initialize a target reference which will hold the Artifact objects. We may write a binding like the one in line 9, which will be incorrect because vertices is of type Vertex.

```
1  rule objectnode {
2    from n : UML!ObjectNode
3    to a : Intalio!Artifact
4  }
5
6  rule activitypartition2pool {
7    from a : UML!ActivityPartition
8    to p : Intalio!Pool (
9      vertices ← a.node→select(e | e.oclIsKindOf(UML!ObjectNode))
10   )
11   l : Intalio!Lane ( ... )
12 }
```

This quick fix will look for references in Pool whose type is compatible with the output type declared in the guilty rules set ({objectnode} in this case). Then, it proposes changing the reference of the binding to the one that minimises the number of guilty rules. In this case, the quick fix will correctly propose changing vertices for artifacts.

4.2 Rule conflicts

In ATL, if two or more rules match the same element, there is a rule conflict. Rule conflicts are not detected statically by the standard ATL IDE even for simple cases, but the transformation fails at runtime. Our static analysis is able to report this situation statically, identifying the set of rules in conflict. Two quick fixes in our catalogue can solve the problem.

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. However, this procedure may yield non-applicable rules in the special case that one of the rules subsumes another one. For this purpose, we perform the following checking: if given two conflicting rules r_1 and r_2, the filter of r_2 subsumes the filter of r_1 (i.e., we have $filter(r_2) \implies filter(r_1)$), then, we do not add the negation of r_1's filter to r_2's filter, as otherwise, the modified filter of r_2 could never be satisfied. This is so as $(filter(r_2) \wedge \neg filter(r_1)) \wedge (filter(r_2) \implies filter(r_1))$ is always false. Checking subsumption is performed by model finding (roughly, checking if $filter(r_2) \wedge \neg filter(r_1)$ is satisfiable).

As illustration, suppose we extend the running example to distinguish two further transformation alternatives for initial nodes: if they receive an accept event action, then they are transformed into an Intalio activity with type EventStartMessage, while if they receive an accept event action which in addition has a time event as trigger, they are transformed into an Intalio activity with type EventStartTimer. The following two rules implement this behaviour:

```
rule initialnode_message {
  from n : UML!InitialNode (
    n.incoming→exists(edge |
      edge.source.oclIsKindOf(UML!AcceptEventAction))
  )
  to a : Intalio!Activity ( activityType ← #EventStartMessage )
}

rule initialnode_timer {
  from n : UML!InitialNode (
    n.incoming→exists(edge |
      if edge.source.oclIsKindOf(UML!AcceptEventAction) then
        edge.source.trigger→exists(t |
          t.event.oclIsKindOf(UML!TimeEvent))
      else
        false
      endif)
  )
  to a : Intalio!Activity ( activityType ← #EventStartTimer )
}
```

Our analyser detects a conflict between these two rules, since any initial node matched by rule initialnode_timer will be also matched by rule initialnode_message. In fact, the filter of initialnode_timer subsumes the filter of initialnode_message. That is, whenever initialnode_timer's filter is true, initialnode_message's filter will be true as well (filter(initialnode_timer) $\implies$ filter(initialnode_message)). Thus, applying the quick fix adds the negation of the filter of initialnode_timer to initialnode_message, but not the other way round. This solves the rule conflict.

The listing below shows the rule that gets modified after the quick fix application:

```
rule initialnode_message {
  from n : UML!InitialNode (
    n.incoming→exists(edge |
      edge.source.oclIsKindOf(UML!AcceptEventAction))
    and
    not n.incoming→exists(edge |
      if (edge.source.oclIsKindOf(UML!AcceptEventAction)) then
        edge.source.trigger→exists(t |
          t.event.oclIsKindOf(UML!TimeEvent))
      else
        false
      endif)
  )
  to a : Intalio!Activity ( activityType ← #EventStartMessage )
}
```

Q0.2 Remove one guilty rule. This quick fix is similar to Q3.1, as it removes one of the problematic rules. The quick fix is only applicable if the set of problematic rules includes two rules, as otherwise, the problem would remain even deleting one rule. This fix requires the user intervention to interactively select the rule to delete. As an assistance to the user, we treat the case that one rule subsumes the other in a special way, highlighting the more specific rule as the most promising to be deleted.

4.3 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" because the receptor object is null or undefined), and a special kind of the *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. The generated condition of the if expression checks that the receptor object is not undefined (for E10) or it is compatible with the type that defines the accessed feature (for E11). If the condition is not satisfied, 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 self.name + '_' + self.oclType().name in line 3 of Listing 1 may cause a runtime exception if self.name (the receptor object of the first + operator) is undefined. Applying this quick fix generates the following conditional, where the else branch returns the empty string as the default value of String expressions:

```
if (not self.name.oclIsUndefined()) then
   self.name + '_' + self.oclType().name
else
   ''
endif
```

Q9.1b Filter problem with select (variant). When the problematic statement is located within a collection operation, it may be more idiomatic to filter the collection before executing the operation. For instance, the following expression yields a problem because the language feature is defined in OpaqueAction but not in the ActivityNode superclass.

```
anActivity.node→select(oa | oa.language = 'OCL')
```

Applying this quick fix would modify the expression as follows:

```
anActivity.node→
  select(oa | oa.oclIsKindOf(UML!OpaqueAction))→
  select(oa | oa.language = 'OCL')
```

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 rule filter is modified to avoid accessing the undefined property. 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 to handle the definite case).

As an example, assume we have the following rule:

```
1 rule initialnode_message {
2   from n : UML!InitialNode
3   to a : Intalio!Activity (
4      name ← 'Initial_to_'+n.outgoing→first().name,
5      activityType ← #EventStartMessage
6   )
7 }
```

There is a possible undefined access in line 4 because the lower cardinality of ActivityNode.outgoing is 0. This quick fix would add the following filter to the rule (see line 2), making the access in line 4 unproblematic:

```
1 rule initialnode_message {
2   from n : UML!InitialNode ( n.outgoing→notEmpty() )
3   to a : Intalio!Activity (
4      name ← 'Initial_to_'+n.outgoing→first().name,
5      activityType ← #EventStartMessage
6   )
7 }
```

Q9.3 Generate transformation pre-condition. This quick fix has the same motivation and is implemented in the same way as Q1.4.

Q10.1 Change feature lower bound to 1 (only for *possible access to undefined property*). This quick fix sets the lower cardinality of the feature to 1 in the type declaration of the receptor object, in the source meta-model. Hence, for a mono-valued feature named feat, navigation expressions of the form obj.feat.feat'... become safe. If the feature is multi-valued, expressions of the form obj.feat→first().feat'... become safe (but not those containing selection operators between feat and first). In this way, this quick fix allows solving the problems in the examples presented in the two previous quick fixes.

As a side effect, this quick fix may cause *compulsory feature not initialized* problems in other transformations that use the same meta-model as target, though our analyser can detect these problems easily.

Q11.1 Create helper (only for *access to property defined in subclass*). The idea of this template quick fix is to emulate the property for the given class. For this purpose, the quick fix creates a new context helper on the type of the receptor object, with the same name as the feature and no parameters. The new helper returns an appropriate default value according to the type of the feature. This default value can be modified later by the developer to provide a more sensible value, if needed. For example, consider the following rule:

```
rule exec2activity {
  from n : UML!ExecutableNode
  to a : Intalio!Activity (
    name ← n.toIntalioName+'_exec_'+n.language
  )
}
```

This rule accesses feature language on objects of type ExecutableNode, but this feature is only defined in subclass OpaqueAction of ExecutableNode. Applying this quick fix produces the following context helper:

```
helper context UML!ExecutableNode
  def: language : String = '';
```

This fixing strategy guarantees that the call will be always resolved by either a meta-model feature or helper operation, which will be selected by the ATL engine using dynamic dispatch (i.e., looking at runtime the type of the receptor object).

4.4 Feature initialization (E4, E5, E6)

These problems correspond to incorrect initializations of features in rule bindings. For instance, the *assignment from higher cardinality* problem (E6) occurs when a binding assigns a collection to a mono-valued feature (i.e., a feature with upper bound equals to 1). In such a case, we provide a quick fix that concatenates the first() operator to the expression in the right part of the binding in order to retrieve just the first object in the collection.

Next, we focus on quick fixes for the *compulsory feature not initialized* problem (E5) since it is the most common error in ATL transformations [35]. This problem kind is signalled when a rule lacks a binding for some feature that is mandatory in the created object (i.e., a feature with lower bound bigger than 0). We provide four possible quick fixes to solve this problem.

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

Q5.1 Assign default value. This is the simplest strategy. It creates a new binding which assigns an appropriate default value to the feature according to its type. For primitive types, we assign the usual default values (e.g., an empty string, 0 for integers, etc.). For objects, we try to assign a target object whose type is compatible with the feature's type, and that is in the scope of the rule.

For example, assuming the cardinality of the reference Pool.lanes to be 1..*, then rule activitypartition2pool is missing a binding for feature lanes. As the rule creates an object of type Lane, the fix would modify the rule as follows:

```
rule activitypartition2pool {
  from a : UML!ActivityPartition
  to p : Intalio!Pool(
       lanes ← Sequence{l}
     ),
     l : Intalio!Lane(...)
}
```

Q5.2 Copy and adapt existing expression. This strategy relies on the same hypothesis as the GenProg system [23]: *"a program that makes a mistake in one location often handles a similar situation correctly in another"*. Indeed, recent empirical studies show that 29-52% of commits are temporally redundant at the token-level, meaning that they are rearrangements of existing code [26].

Inspired by these findings, we seek 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 that can be accessed from the current rule. A variable in the current rule is compatible with a variable used in the candidate binding if the feature calls over the candidate variable can be performed over the current rule variable. If there are several compatible bindings, we select one at random, which is copied to the rule once it is conveniently adapted for it.

For example, in lines 23 and 24 of Listing 1, there is no binding for the name feature. However, this feature is assigned in lines 16 (name ← a.name) and 32 (name ← n.toIntalio). From these two possibilities, the second one is discarded because our analyser detects that it has an error (there is no feature or operation named toIntalio); hence, only the former is available to initialize the missing bindings. Applying this quick fix to line 23 yields:

```
rule activitypartition2pool {
  from a : UML!ActivityPartition
  to p : Intalio!Pool(
       name ← a.name
     ),
     ...
}
```

Note that even if we correct line 32 to obtain the binding name ← n.toIntalioName, it is not offered as an option to fix the errors in lines 23 and 24. This is so as the context helper toIntalioName is defined for type Action, and thus, it cannot be applied to objects of type ActivityPartition (i.e., using a.toIntalioName in the previous listing would be incorrect).

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

For example, in rule activitypartition2pool, the created Pool and Lane objects miss the initialization of their feature name. This quick fix checks the objects in the from pattern (ActivityPartition) to look for attributes similar to "name". In this case, similarity is checked by maximal substring containment. Since ActivityPartition has an attribute name of type String, the binding name ← a.name is synthesized for both created objects.

4.5 Declaration mismatch (E8)

This problem is frequent in ATL transformations developed without any static analysis tool like ours, as ATL does not check either statically or dynamically variable, parameter and return type declarations. While a declaration mismatch does not cause runtime exceptions when the transformation is executed, it is a maintenance problem because developers may have expectations according to the declared types, which may differ from the real ones.

Q8.1 Change declared type with inferred type. This quick fix replaces the declared type with the type inferred by the static analyser. In the running example, the return type of helper toIntalioName in line 2 of Listing 1 (which was declared as Intalio!Activity) is fixed to:

```
helper context UML!Action def: toIntalioName : String =
  self.name + '_' + self.oclType().name;
```

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.

4.6 Feature/operation not found (E12, E14)

In this section, we analyse quick fixes related to accessing non-existing features or operations. Some of these quick fixes rely on creating helpers. 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. ATL also distinguishes between *context helpers*, which are defined in the context of a class and act as regular polymorphic methods, and *module helpers*, which do not have any context and behave as global functions. For practical purposes, we consider lazy and called rules to be module helpers since they are invoked like module helpers with one or more parameters.
Q12.1 Suggest existing feature/operation. We use several string distance metrics [6] to look up candidate features (including both helper and meta-model features) and operations with a similar name. In first place, we use the Levenshtein distance, which measures the number of character edits (insertion, deletion, swap) required to change one word into another one. This gives good results for spelling mistakes like writing toIntallioName instead of toIntalioName (distance 1). If no good proposal is found within a threshold, we switch to "longest common substring" distance, which favours strings with similar subsequences. This distance gives good results when names are approximated (e.g., perhaps not properly recalled by the developer), like typing toIntalio instead of toIntalioName. In the case of operations, we take into account the types of the parameters to narrow the search and improve the accuracy of the proposal. We also consider built-in functions, such as collection operations.
Q12.2 Create new context helper (only for calls over an object). This template quick fix is useful to automatically create a skeleton of an operation. The fix creates a new context helper whose context is the class inferred for the receptor object of the call, and its formal parameters are created according to the types of the actual parameters in the call. The body of the helper is initialized to a default value according to its return type.
Q12.2 Create new module helper (only for 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 between these three options is the following:

- **Lazy rule**. This option is selected when the problematic call is in a "binding assignment position", that is, a location in the right part of a binding that will make the result of the call be assigned to the binding's feature. For instance, if the call is a direct child of the binding (i.e., feature ← thisModule.call(param)) or if the call is within the body of a collect. Moreover, the feature initialized by the binding must be a reference (i.e., it must have a non-primitive type), the call must have exactly one argument, and this argument must be a single object.
- **Called rule**. The same as lazy rules, but either the call has more than one argument, or it has just one argument of primitive type or a collection.
- **Module helper**. Otherwise.

This heuristic reflects the most usual invocation patterns in ATL. As an example, suppose we modify the binding for feature pools in rule activity2diagram as follows:

```
rule activity2diagram {
  from a : UML!Activity
  to   m : Intalio!BpmnDiagram (
     name ←  a.name,
     pools ←  thisModule.allPartitions(a.allPartitions)
  )
}
```

As the transformation does not define a module helper or rule named allPartitions[7] with one parameter, the quick fix suggests creating the following called rule:

```
rule allPartitions(arg0 : Sequence(UML!ActivityPartition)) {
    to tgt : Intalio!Pool
}
```

While this is a template quick fix, and so the user should fill the helper body, the quick fix correctly infers the signature and return type of the helper.
Q12.3 Create feature in the meta-model (only for calls over an object). It adds a new feature to the receptor object's type. To automate this as much as possible, we try to infer the type of the feature from the expression in which it appears. We have defined a set of pre-defined locations for which it is possible to use other types inferred by the analyser as the type for the feature. As an example, let us suppose that OpaqueAction does not have a language feature and we have the code shown below.

```
helper def: normalize(str : String) : String =
  str.toUpperCase();

-- Faulty feature access
thisModule.normalize(anOpaqueAction.language)
```

Since the invalid feature access is in a parameter position and the types for the normalize helper have been

[7] The transformation does define two attribute helpers in the context of Activity and ActivityPartition, but not a one-parameter helper at the module level.

properly inferred, we can determine that language has to be an attribute of type String. If the type expected by the faulty expression's location is a collection, then we set the upper bound to *, otherwise to 1 (as in this example). By default, we set the lower bound to 1. If it is not possible to infer the type, the user would be asked to introduce the type of the feature through a dialog.

Q12.4 Change feature call to operation call, and vice versa. In ATL, the syntax to call an attribute helper requires no parentheses, as otherwise, it is interpreted as an operation helper. Thus, a call like activity.allPartitions() causes a runtime exception if the only helpers defined in the transformation are those of Listing 1. This is a common mistake among ATL beginners.

This quick fix proposes replacing an operation call by an attribute call if there is an attribute with the same name as the invoked operation. In the example, the call activity.allPartitions() would be changed to activity.allPartitions.

5 Impact of Quick Fixes

The application of a quick fix is a local action targeted to fixing the problem identified at the selected location. However, the action may have side effects in the form of new problems appearing in other locations or even existing problems being automatically fixed. Understanding these side effects is important both for the tool perspective (e.g., to rank quick fixes) and for the user perspective who would like to make an informed decision when determining how the transformation should be fixed.

As an example, let us consider the *possible unresolved binding* problems in lines 25, 47 and 48. If we apply the quick fix *Q2.1 Create new rule* to line 25, all three problems are solved at once. However, if we choose to apply quick fix *Q1.2 Remove problematic binding* or *Q1.3 Add filter to binding expression*, they will have a local effect, fixing only the targeted problem in line 25.

On the other hand, if we apply the quick fix Q2.1 to the problems in lines 47 or 48, it would create a new rule having Vertex in the to pattern, as this is the most general compatible type (see lines 2–12 in the listing below). This new rule is resolved by the three mentioned bindings (copied in the listing below in lines 17, 18 and 25), but now, a new error is introduced in line 25 since the activities feature is assigned an incorrect type.

```
1  -- Generated rule
2  rule ActivityNode2Vertex {
3    from n : UML!ActivityNode (
4      not n.oclIsKindOf(UML!OpaqueAction) and
5      not (if n.oclIsKindOf(UML!InitialNode) then
6            n.incoming→isEmpty()
7          else
8            false
9          endif)
10   )
11   to a : Intalio!Vertex
12 }
13
14 -- Fixed bindings
15 rule edges {
16 ... to a : Intalio!SequenceEdge (
17       source ← n.source,
18       target ← n.target
19     )
20 }
21
22 -- Problematic bindings
23 rule activitypartition2pool {
24 ... to l : Intalio!Lane (
25       activities ← a.node→reject(e | e.oclIsKindOf(UML!ObjectNode))
26     )
27 }
```

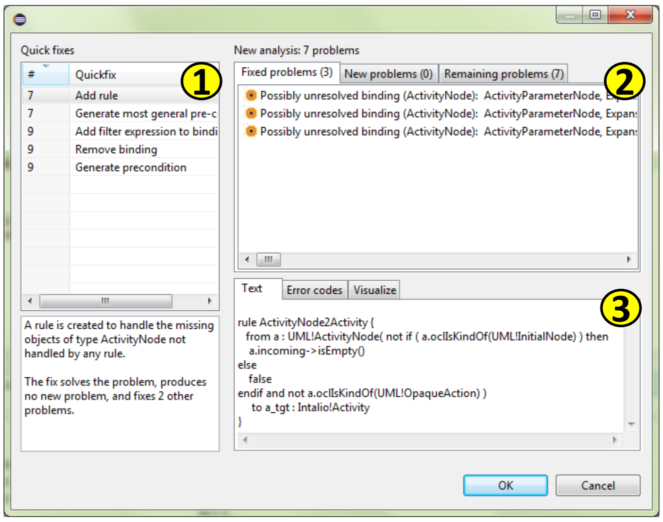
In this example, the best option is adding a new rule by quick fixing line 25, but the developer may need to try the different choices manually before reaching this verdict, as well as undoing the quick fix applications that lead to unsatisfactory repairs. Instead of this manual process, it would be useful to complement the quick fix proposals with a technique able to foresee the transformation state that would result from applying each proposed quick fix.

Therefore, we use *speculative analysis* to help the user understand the impact of a quick fix. This is a general technique to explore the consequences of modifying some piece of code before the change actually happens [3]. This idea has been applied to rank quick fixes in the context of Java and Eclipse [28] by reporting the number of errors left in a Java project after the application of each possible quick fix. However, considering the total number of remaining errors may be misleading when the code has several problems and a quick fix corrects some of them but it also introduces new problems in other locations.

In this work, we have developed a technique that provides a finer-grained analysis of the impact of a quick fix application. Our technique automatically detects the fixed problems and the newly generated ones after applying a quick fix without actually modifying the transformation text or its meta-models. The technique is detailed in Section 5.1. The information derived from this analysis can be used for the following purposes:

- **Providing impact information.** This information helps the user understand the consequences of applying a quick fix. As an example, Figure 8 shows the dialog that anATLyzer presents to the user, which contains the list of fixed and generated problems after speculatively applying the quick fix. This functionality is explained in Section 5.2.
- **Ranking quick fixes**. Code recommenders typically put the most valuable recommendations at the top of their rankings [34]. These rankings are calculated using models that predict the usefulness for the user. In our case, we use speculative analysis to provide a ranking where the fixes that remove more errors are listed first. We call this ranking *dynamic*. Alternatively, the ranking could be calculated statically (i.e., without speculative analysis) by analysing the actual impact of quick fix applications on a corpus of existing transformations. We discuss these rankings in Section 5.3.

– **Empirical evaluation of quick fix validity**. The information provided by the speculative analysis can be used to validate the quick fix implementation and take measurements about its behaviour. Hence, in Section 7, we use our technique to perform an empirical evaluation of the behaviour of our quick fixes.
– **Model transformation repair**. Speculative analysis can be used to implement repair techniques for model transformations. Whereas in this work we use speculative analysis for a single quick fix, it would be possible to generate a chain of fixes that completely repair a given transformation. This would require using search strategies to prune the state space in order to speed up the process. We leave this application for future work.



Quick fixes

#	Quickfix
7	Add rule
7	Generate most general pre-c
9	Add filter expression to bindi
9	Remove binding
9	Generate precondition

A rule is created to handle the missing objects of type ActivityNode not handled by any rule.

The fix solves the problem, produces no new problem, and fixes 2 other problems.

New analysis: 7 problems

Fixed problems (3) | New problems (0) | Remaining problems (7)

```
Possibly unresolved binding (ActivityNode):  ActivityParameterNode, Ex
Possibly unresolved binding (ActivityNode):  ActivityParameterNode, Expan
Possibly unresolved binding (ActivityNode):  ActivityParameterNode, Expan
```

```
rule ActivityNode2Activity {
  from a : UML!ActivityNode( not if ( a.oclIsKindOf(UML!InitialNode) ) then
   a.incoming->isEmpty()
else
   false
endif and not a.oclIsKindOf(UML!OpaqueAction) )
    to a_tgt : Intalio!Activity
}
```

Fig. 8 Dialog showing the impact of quick fixes applied speculatively for the problem in line 25 of Listing 1.

5.1 Speculative analysis of fixed and generated problems

The speculative analysis of a quick fix application is performed using the procedure sketched in Listing 2. Given a transformation T and a problem P to be fixed, the speculative analysis applies every available quick fix systematically to a fresh copy of the transformation (lines 4–9). Copying the original transformation is important to avoid interferences with the user's working copy. The copy function returns an exact copy of the abstract syntax tree T', and also the trace between the original transformation and the copy. In line 5, we take into account quick fixes that modify meta-models by copying them. This implies not only copying the meta-model, but also replacing every reference to the original meta-models in the abstract syntax tree to the copied version. Next, we create a copy P' of the problem which points to the copied transformation, and the original quick fix is modified to point to the copied problem. Finally, the quick fix is applied producing a modification on the copied transformation T' (lines 11–12), and this transformation is statically analysed (lines 14–16).

```
 1  Input: transformation T, problem P
 2
 3  Foreach Q in quickfixesOf(P)
 4    T', Trace <− copy(T)
 5    if Q.modifiesMetamodels
 6      copyMetamodels(T)
 7    end
 8    Create P' that refers to T' from P
 9    Change Q to point to P'
10
11    applyQuickfix(Q)
12    updateTrace(Q, Trace)
13
14    cleanTransformation(T')
15    staticAnalysis(T')
16    analyseImpact(T, T', Trace)
17  end
```

Listing 2 Speculative analysis procedure.

A subtle step is the need to update the trace information to reflect the changes made by the quick fix (line 12). This is particularly important for quick fixes that replace an AST element for another. Another consistency action is cleaning up the modified transformation T' (line 14), as we need to remove the existing TDG information from the abstract syntax tree of T' so that it can be freshly re-typed and analysed after the quick fix application. The more simplistic approach of overwriting the existing analysis information does not work. For example, if the original transformation has a binding resolved by two rules and a quick fix removes one of them, the new transformation will have stale information in the form of a binding with two resolving rules instead of one. An alternative to removing all the analysis information beforehand would be to perform an incremental static analysis, which is part of our future work.

Finally, we analyse the impact of the quick fix application. This procedure, which is outlined in Figure 9, is only possible when quick fixes are applied over the abstract syntax of the program. Since this is not the default option provided by Eclipse, we had to build an infrastructure to support our approach. Section 6 provides more details.

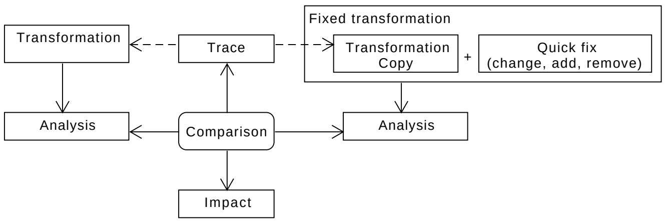


Fig. 9 Impact computation process.

We analyse the impact of a quick fix application in a generic way with the algorithm shown in Listing 3. We assume that every problem has a reference to the abstract syntax element marked as problematic (element attribute), and the trace model has an operation get-

Target which returns the abstract syntax element in the copied transformation that corresponds to a given original element. The intuition behind the algorithm is that, if given a problem in the new transformation (p' in T') we cannot find the corresponding problem in the original transformation, then it is a new problem (lines 8–16). The other way round, if we cannot trace a problem in the original transformation to a problem in the new transformation, then the problem has been fixed and thus no longer appears in the problem list of the new transformation (lines 24–37). We check if two problems are equal by comparing their problematic AST elements, but we also check that they are the same type of problem (lines 13 and 29). This check is needed because the same AST element may cause more than one problem.

```
 1 Input: transformation T, copied transformation T', trace Trc
 2 Output: set of newProblems, set of fixedProblems
 3
 4 newProblems = { }
 5 fixedProblems = { }
 6
 7 -- Detection of new problems
 8 Foreach p' in T'.problems
 9   e' = p'.element
10   found = false
11   Foreach p in T.problems
12     tgt = Trc.getTarget(p.element)
13     if e' = tgt and p.class = p'.class
14        found = true
15     end
16   end
17
18   if not found
19     newProblems.add(p')
20   end
21 end
22
23 -- Detection of fixed problems
24 Foreach p in T.problems
25   found = false
26   Foreach p' in T'.problems
27     e' = p'.element
28     tgt = Trc.getTarget(p.element)
29     if e' = tgt and p.class = p'.class
30        found = true
31     end
32   end
33
34   if not found
35     fixedProblems.add(p')
36   end
37 end
```

Listing 3 Calculating the impact of a quick fix.

5.2 Presenting impact information

The main application of our speculative analysis is to help the user reason about the consequences of applying a quick fix without the burden of modifying the transformation and undoing the undesired fixings by hand. Without speculative analysis, the user has to apply the selected quick fix. The quick fix will change either the transformation or the meta-models. To inspect the change, the user must locate the place where the change was made. This may not be straightforward until the developer is familiar with the catalogue of fixes. For example, applying quick fix Q0.1 to repair a rule conflict may modify two rule filters or just one. After applying the quick fix, the analyser will update the list of problems, but if the original transformation contained many errors, it may be difficult to identify which ones of them have been fixed. Moreover, if the result of a quick fix is not satisfactory, the user must undo the quick fix and try another one. This may not be possible automatically if the quick fix modified the meta-models.

Hence, our objective is to present the user a concise "picture" of the state that the transformation would have if the quick fix is applied. This picture includes five parts: (i) whether the quick fix will actually fix the targeted problem, (ii) the complete list of problems that the quick fix will solve, (iii) the list of generated problems, (iv) the list of remaining problems, and (v) information about the modifications that the quick fix will perform on the transformation (e.g., a piece of generated code).

Figure 8 shows the dialog that realises this idea. We perform our speculative analysis for each available quick fix (label 1). When the user clicks on the quick fix, the result is shown in the problems tabs (label 2) and in the text tab (label 3). For usability reasons, each speculative analysis is run in a separate thread so that the dialog is not blocked during the computation and the user can inspect the results as they finish.

To synthesize the piece of text in label 3, we take into account the type of change performed by the quick fix. Our quick fixes are implemented using a dedicated API to perform changes on the ATL abstract syntax model (see Section 6). The different types of change (e.g., deleting an element, inserting an element into a container, modifying a meta-model feature, etc.) are recorded including information about the modified or created element (or a copy if it was deleted). This information is used by a generic procedure that synthesizes the piece of text. Hence, any quick fix implemented with our infrastructure will have this feature for free.

5.3 Ranking quick fix proposals

Speculative analysis provides detailed information on the problems fixed, remaining errors and newly introduced errors that result from the application of a quick fix. This way, the applicable quick fixes can be ranked according to this information, so that quick fixes ranked first will be more likely selected by the user. However, gathering this information has some computational cost, and depends on the particular error being solved.

For this reason, we also provide a static ranking of quick fixes which is offered by default to the developer, without the need to perform speculative analysis. The ranking has been derived from empirical evidence of quick fix performance. In particular, we selected four transformations without errors, and injected errors in them using transformation-specific mutation operators. This produced 816 mutated transformations with errors.

Section 7.1 contains a detailed description of the selected transformations and the mutation procedure. Then, we automatically applied all possible quick fixes to each detected problem, and measured how many errors they fixed and introduced. Each quick fix qf was given as *efficacy* score a number between -1 and 1 calculated as $eff(qf) = (fixed - new)/(fixed + new)$. This way, a quick fix is ranked higher the more errors it solves and the less new errors it introduces. Then, we averaged the efficacy scores across the four evaluated transformations. Quick fixes with same score were ordered according to higher number of applicability (i.e., frequency). Table 2 shows the obtained static ranking. We only show the errors for which more than one type of quick fix is available. Moreover, we exclude from the ranking the quick fixes that require user intervention, like *Q0.2 Remove one guilty rule*, as they cannot be applied automatically.

Table 2 Static ranking (empirical) of quick fixes.

Problem	Quick fix	Score
E2: Unres. binding	Q1.5 (gen. gen. pre-cond)	1 (125)
	Q1.2 (rem. binding)	1 (94)
	Q1.1 (modify rule filter)	1 (38)
	Q1.4 (gen. pre-cond)	0,96
	Q1.3 (add filter to binding)	0,89
	Q2.1 (create new rule)	0,37
E3: Invalid target for resolved binding	Q1.5 (gen. gen. pre-cond)	1 (122)
	Q1.3 (add filter to binding)	1 (76)
	Q1.2 (rem. binding)	1 (61)
	Q1.4 (gen. pre-cond)	0,82
	Q1.1 (modify rule filter)	0,79
	Q3.2 (choose different)	0,39
	Q3.1 (rem. guilty rule)	-0,07
E5: Compulsory feat. not initialized	Q5.1 (assign. def. value)	1 (107)
	Q4.1 (modify feat. in MM)	1 (99)
	Q5.2 (copy & adapt exp.)	1 (79)
	Q5.3 (mapping to similar)	1 (30)
E7: Invalid type	Q7.1 (suggest from MM)	0,22
	Q7.2 (add to MM)	0,17
E10: Possible access to undefined prop.	Q9.3 (gen. pre-cond)	1
	Q9.1 (surround with if)	0,91
	Q9.2 (modify rule filter)	-0,04
E11: Access to prop. defined in subclass	Q11.1 (create helper)	1
	Q12.2 (create cont./mod.)	0,15
	Q9.1 (surround with if)	-0,2
	Q12.1 (suggest existing)	-0,45
	Q9.3 (gen. pre-cond)	-0,55
E12: Feature not found	Q12.2 (create cont./mod.)	0,71
	Q12.1 (suggest existing)	0,15
E14: Operation not found	Q12.1 (suggest existing)	0,81
	Q12.2 (create cont./mod.)	4,1
E15: Incompatible parameter	Q15.2 (change formal)	-0.5
	Q15.1 (create operation)	-1
E16: Invalid number of parameters	Q16.1 (add/rem. actual)	0,6
	Q16.2 (add/rem. formal)	0
	Q16.3 (choose other)	-0,25

A consequence of the chosen efficacy metric is that template quick fixes tend to score poorly and are ranked in the last positions, as they may introduce new issues. For example, *Q2.1 Create new rule* is ranked last among the quick fixes for E2, because it tends to generate many *compulsory feature not initialized* errors (E5). In future work, we might try to detect the user “mode”: when the transformation is under heavy construction, then template quick fixes might be more commonly used, while in the testing phase, repair quick fixes are more frequent.

We can also observe that the same quick fix is ranked differently depending on the targeted error, like for E2 and E3. However, in this case, generating a general precondition (Q1.5) seems to be less problematic and is applied more frequently. In Section 7.3, we validate our static ranking by comparison with the dynamic ranking calculated using speculative analysis.

6 Implementation

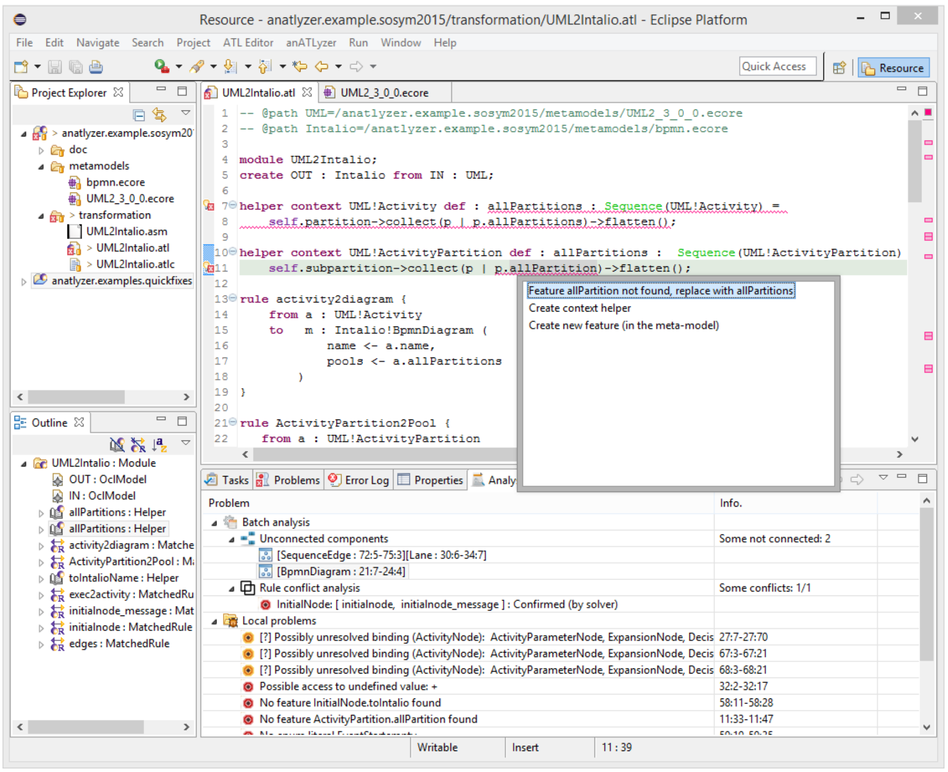
Our proposal is backed by an implementation atop anATLyzer [35], our static analyser for ATL that is integrated with the regular Eclipse/ATL IDE. Quick fixes are available through the standard facilities provided by Eclipse, complemented with a dedicated analysis view to easily inspect and fix detected problems (see Figure 10), as well as a control dialog to obtain information of the speculative analysis (see Figure 8).

The analysis view contains two sections, one with problems that need to be analysed in batch, and another with problems that are detected automatically and may need to be confirmed interactively (see Figure 10). Batch problems (label batch analysis) are typically the most costly to calculate and require the user to start the computation. These include rule conflict analysis (which uses the model finder to analyse a possibly high number of rule pairs) and unconnected components analysis (which determines if the transformation generates a connected model graph or several subgraphs). The automatically detected problems (label local problems) refer to problems in the transformation and are detected in the background. To improve the user experience, the problems that require confirmation using the solver have to be triggered by the user (first 3 rows below “local problems” in the figure). If they have not been confirmed yet, they are marked with a “[?]” (as in the figure). If they get confirmed by the solver, they are marked as “[C]”, while if they are discarded, they are signalled with a “[D]”. Errors show a red circle icon in the view, while warnings show a yellow circle icon. In all cases, the errors are also signalled in the code just like in regular programming IDEs.

For ease of use, the list of available quick fixes for an error is displayed via a shortcut key (shown in the picture), while the speculative analysis tool needs to be invoked separately (Figure 8). The list of quick fixes in Figure 10 is ordered according to the static ranking presented in Section 5.3. The idea is to present first those quick fixes with the highest probability of being the most suitable ones.

A screencast of the tool, the source code of the project 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


Resource - anatlyzer.example.sosym2015/transformation/UML2Intalio.atl - Eclipse Platform
File Edit Navigate Search Project ATL Editor anATLyzer Run Window Help
Quick Access
Resource
Project Explorer
UML2Intalio.atl
UML2_3_0_0.ecore
Outline
Tasks Problems Error Log Properties Analy
Feature allPartition not found, replace with allPartitions
Create context helper
Create new feature (in the meta-model)
Writable
Insert
11 : 39

```
-- @path UML=/anatlyzer.example.sosym2015/metamodels/UML2_3_0_0.ecore
-- @path Intalio=/anatlyzer.example.sosym2015/metamodels/bpmn.ecore

module UML2Intalio;
create OUT : Intalio from IN : UML;

helper context UML!Activity def : allPartitions : Sequence(UML!Activity) =
    self.partition->collect(p | p.allPartitions)->flatten();

helper context UML!ActivityPartition def : allPartitions :  Sequence(UML!ActivityPartition)
    self.subpartition->collect(p | p.allPartition)->flatten();

rule activity2diagram {
    from a : UML!Activity
    to   m : Intalio!BpmnDiagram (
            name <- a.name,
            pools <- a.allPartitions
        )
}

rule ActivityPartition2Pool {
   from a : UML!ActivityPartition
```

```
Problem                                                                                              Info.
Batch analysis
  Unconnected components                                                                             Some not connected: 2
      [SequenceEdge : 72:5-75:3][Lane : 30:6-34:7]
      [BpmnDiagram : 21:7-24:4]
  Rule conflict analysis                                                                             Some conflicts: 1/1
      InitialNode: [ initialnode,  initialnode_message ] : Confirmed (by solver)
Local problems
    [?] Possibly unresolved binding (ActivityNode):  ActivityParameterNode, ExpansionNode, Decis     27:7-27:70
    [?] Possibly unresolved binding (ActivityNode):  ActivityParameterNode, ExpansionNode, Decis     67:3-67:21
    [?] Possibly unresolved binding (ActivityNode):  ActivityParameterNode, ExpansionNode, Decis     68:3-68:21
    Possible access to undefined value: +                                                            32:2-32:17
    No feature InitialNode.toIntalio found                                                           58:11-58:28
    No feature ActivityPartition.allPartition found                                                  11:33-11:47
```

Fig. 10 Screenshot of the tool.

API that we have built. This decision, which was originally motivated by the need to automatically apply quick fixes in our experiments, provides the necessary infrastructure to support advanced options like speculative analysis. Working at the abstract syntax level 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.

Our catalogue of quick fixes is extensible by means of an Eclipse extension point and a set of pre-defined abstract quick fixes which provide useful functionality to implement concrete quick fixes. Quick fixes benefit from the services of our API to modify the ATL abstract syntax.

7 Evaluation

This section reports on the evaluation of our system. First, in Section 7.1, we evaluate its completeness and validity by systematically generating mutants of a set of transformations in order to create a wide range of problems, which we try to fix with our catalogue of fixes. We evaluate completeness by assessing whether at least one quick fix is applicable for each error reported by the analyser, and we evaluate validity by checking whether an applied quick fix has effectively fixed the targeted problem.

Then, in Section 7.2, we study the impact of each quick fix application by looking at the problems it fixes or introduces as a side effect. We have used this information to derive the static ranking of quick fixes presented in Section 5.3.

Finally, in Section 7.3, we evaluate the usefulness of our catalogue of quick fixes and its ranking to repair a set of transformations different from those used in the first experiment, and written by a third-party. In this experiment, we record the fixing strategy of two independent ATL developers to solve every error. Then, we study whether a quick fix is available to solve the problem in the same way, as well as the position of the quick fix in the static and dynamic rankings.

The section concludes with an analysis of possible threats to the validity of our evaluation (Section 7.4) and a general discussion about our quick fix system (Section 7.5).

All the data gathered from the experiments and the artefacts used (source code, transformations and mutants) are available at `http://miso.es/qfx_exp_sosym2015`.

7.1 Evaluating validity and completeness

We say that a quick fix is *valid* if it always fixes its targeted error. An error is *completely* covered by the quick fix catalogue, if there is always an *applicable* quick fix. In this respect, note that even if we implement a quick fix per problem type, there may be actual problems without any fix due to the application condition of the fix.

To evaluate the validity and completeness of our quick fix catalogue, we have performed an experiment based on applying mutations to generate possibly faulty transformations, and then using speculative analysis to measure how many quick fixes are applicable (for completeness) and to determine if each applicable quick fix resolves the targeted problem (for validity).

We have used four error-free transformations for the evaluation: 1) *PNML2PetriNet* from the *Grafcet to PetriNet* scenario in the ATL zoo[8], which we used for an initial evaluation in [36]; 2) an extended and error-free version of the *UML2Intalio* transformation used as running example in this paper; 3) *Ant2Maven* from the ATL zoo; and 4) an extended version of the *Class2Table* transformation from the ATL zoo. The two latter transformations were chosen because they have few and easily fixable problems and the domains were known for us (i.e., we want to introduce as little bias as possible when fixing the transformations manually). Table 3 summarizes the details of the transformations and the number of mutants generated.

Table 3 Transformations used in the evaluation. *Mut.* is the number of generated transformation mutants, and *Ev.* is the number of mutants evaluated (i.e., a mutant needs to have at least a problem to be valid for the experiment). *Rules/Helpers* shows the number of rules/helpers in the transformation, and *Classes* shows the number of source/target classes.

Transformation	Mut.	Ev.	Rules/Helpers	Classes
PNML2PetriNet	318	264	5/0	13/9
UML2Activity	318	210	9/6	248/20
Ant2Maven	242	160	30/0	48/59
Class2Table	260	182	8/4	6/5

From each transformation, we generated a set of new transformations obtained by applying one mutation operator once to the original transformation. Table 4 shows the transformation-specific mutation operators considered in our experiment. In this way, for each mutation, the generated transformation mutant is expected to have only one problem. In case it contains several problems, we classify them in a hierarchy so that we only try to fix problems that do not depend on others. In this way, we avoid the noise introduced by errors possibly propagated by the depending problem. Transformation mutants that contained no error were discarded for the experiment.

We run the experiment separately for each of the four transformations, applying the automation script in Listing 4 to each mutant of the transformations and aggregating the results obtained in each run. We ruled out from the evaluation quick fixes that require user intervention, like Q0.2, and did not consider variants of the

Table 4 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 parameter in operation or called rule 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 type parameter (e.g., oclIsKindOf(Type))
Feature name modification	navigation expression target of binding
Operation modification	predefined operator (e.g., and) predefined operation (e.g., size) collection operation (e.g., includes) iterator (e.g., exists, collect) operation/attribute helper invocation

quick fixes as they share the same behaviour. In this experiment, we took advantage of our speculative analysis to apply a quick fix while keeping trace links to the original transformation (as explained in Section 5.1) in order to compute the set of fixed and new problems. This is a key difference with the evaluation performed in [36], where we were not able to accurately determine if a quick fix was valid or not.

```
Input: Mutated transformation T
Step 0: Run the static analyser on T, obtaining its list of problems Lp
Step 1: Confirm or discard potential problems by finding a model witness
        If discarded, remove problem from Lp
Step 2: Foreach P in Lp
            Retrieve the set of available quick fixes for P
            Foreach applicable quick fix Q
                Discard quick fix if it requires user intervention
                Count Q as applicable for the problem type of P
                Apply Q on T speculatively
                    Copy T into Tq
                    Apply Q on Tq
                    Run the static analyser on Tq
                    Confirm or discard problems for Tq, as in step 1
                Compute the impact to obtain fixedProblems
                    and newProblems
                If P belongs to newProblems
                    Mark the application of Q as valid
```

Listing 4 Procedure to perform the experiment based on mutation.

Table 5 shows the results of the experiment by aggregating the data obtained for the four transformations. 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 simultaneous quick fix proposals in the transformations (*Min*, *Max*). Column *#Valid* shows the number of quick fixes deemed as valid by the speculative analysis, and *Valid?* indicates if all proposed quick fixes for the type of problem are valid. Columns *#Fixed* and *#Gen* show the number of fixed and newly generated problems after applying

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

Table 5 Aggregated results for the errors and quick fixes in *PNML2PetriNet*, *UML2Intalio*, *Ant2Maven* and *Class2Table*. *Type* is the type of quick fix (Heuristic, Repair, Template). *#Occ* indicates the number of occurrences of each problem/quick fix. *Avg*, *Min* and *Max* show the average number of applicable quick fixes per error, the minimum and the maximum respectively. *#Valid* is the number of quick fixes identified as valid. *#Fixed* and *#Gen* show the total number of fixed and generated problems. *#Extra* shows the problems fixed in addition to the quick fix's target problem.

	Type	#Occ	#Qfx	Avg	Min	Max	#Valid	Valid?	#Fixed	#Extra	#Gen
E00		59	59	1.00	1	1	59		104		19
Q0.1	R	-	59	-	-	-	59	Yes	104	45	19
E02		154	455	3.40	1	6	455		837		195
Q1.1	R	-	24	-	-	-	24	Yes	38	14	0
Q1.2	R	-	84	-	-	-	84	Yes	94	10	0
Q1.3	R	-	70	-	-	-	70	Yes	71	1	4
Q1.4	R	-	65	-	-	-	65	Yes	83	18	2
Q1.5	R	-	65	-	-	-	65	Yes	125	60	0
Q2.1	T	-	147	-	-	-	147	Yes	426	279	189
E03		84	431	5.31	1	6	420		650		271
Q1.1	R	-	28	-	-	-	28	Yes	44	16	5
Q1.2	R	-	49	-	-	-	49	Yes	61	12	0
Q1.3	R	-	76	-	-	-	76	Yes	77	1	0
Q1.4	R	-	74	-	-	-	74	Yes	101	27	10
Q1.5	R	-	74	-	-	-	74	Yes	123	49	0
Q3.1	R	-	84	-	-	-	84	Yes	205	121	239
Q3.2	H	-	46	-	-	-	35	No	39	0	17
E05		99	307	3.10	2	4	307		315		0
Q4.1	R	-	99	-	-	-	99	Yes	99	0	0
Q5.1	R	-	99	-	-	-	99	Yes	107	8	0
Q5.2	H	-	79	-	-	-	79	Yes	79	0	0
Q5.3	H	-	30	-	-	-	30	Yes	30	0	0
E06		30	26	0.89	0	1	26		30		3
Q6.1	R	-	26	-	-	-	26	Yes	30	4	3
E07		10	20	2	2	2	20		28		19
Q7.1	H	-	10	-	-	-	10	Yes	14	4	9
Q7.2	R	-	10	-	-	-	10	Yes	14	4	10
E08		43	34	0.82	0	1	34		36		3
Q8.1	R	-	34	-	-	-	34	Yes	36	2	3
E10		75	263	3.5	3	4	263		371		62
Q9.1	R	-	75	-	-	-	75	Yes	90	15	4
Q9.2	R	-	38	-	-	-	38	Yes	53	15	58
Q9.3	R	-	75	-	-	-	75	Yes	102	27	0
E11		15	59	3.94	3	5	38		53		57
Q9.1	R	-	7	-	-	-	4	No	4	0	6
Q9.3	R	-	7	-	-	-	4	No	4	0	15
Q11.1	T	-	15	-	-	-	12	No	12	0	0
Q12.1	H	-	15	-	-	-	6	No	6	0	16
Q12.2	T	-	15	-	-	-	12	No	27	12	20
E12		101	156	1.58	0	2	126		188		81
Q12.1	H	-	94	-	-	-	64	No	86	0	64
Q12.2	T	-	62	-	-	-	62	Yes	102	40	17
E14		118	116	1.07	0	2	107		150		40
Q12.1	H	-	59	-	-	-	50	No	72	13	8
Q12.2	T	-	57	-	-	-	57	Yes	78	21	32
E15		133	13	0.22	0	2	2		2		8
Q15.1	T	-	11	-	-	-	0	No	0	0	2
Q15.2	R	-	2	-	-	-	2	Yes	2	0	6
E16		51	80	1.58	1	3	29		30		12
Q16.1	H	-	26	-	-	-	26	Yes	27	1	7
Q16.2	R	-	51	-	-	-	0	No	0	0	0
Q16.3	H	-	3	-	-	-	3	Yes	3	0	5
E17		201	95	0.87	0	2	95		105		0
Q17.1	R	-	95	-	-	-	95	Yes	105	10	0
E18		8	8	1	1	1	8		8		1
Q18.1	R	-	8	-	-	-	8	Yes	8	0	1

the quick fix, i.e., its impact, which will be analysed in more detail in Section 7.2.

The average number of proposed quick fixes for *binding-related problems* (*E02*, *E03* and *E05*) ranges between 3 and 5. In the case of *E00*, there is only one quick fix that does not require user intervention and can be automatically evaluated. In all cases, the minimum number of applicable quick fixes is greater or equal to 1. Other typical kinds of problem in model transformations are *E10 Possible access to undefined property* and, in particular for ATL, *E11 Access to property defined in subclass*. For these, the average number of quick fixes is about 3.5 and there is always at least one applicable quick fix. In the case of problems not specific of model transformations, our catalogue is less complete as there are normally only one or two proposals at most, and for some types of problems, the average number of applicable quick fixes is low. Nevertheless, our focus when designing the catalogue has not been on typical problems of object-oriented languages, as they are substantially covered by IDEs like Eclipse/JDT or IntelliJ. As a matter of fact, it should be simple to implement in our tool many of the quick fixes available in them, since the object-oriented concepts used by ATL are essentially the same as in mainstream programming languages. Hence, we can claim that our catalogue satisfies the completeness criteria for *transformation-specific problems*, but it is limited for more general kinds of problems.

Regarding validity, most quick fix applications have fixed the targeted problem. In general, validity is important to avoid misleading the user when a quick fix is applied. The only quick fix types for which the *#Occ* and *#Valid* columns differ are *heuristic* quick fixes (except for three occurrences in each quick fix of *E11* which are due to our analyser behaving incorrectly for these three specific mutations). This shows that our quick fixes for invalid invocations are not precise enough. We have observed two reasons. On the one hand, the synthetic changes made by the mutants do not work well with our string distance heuristics, which are intended to fix small typos. On the other hand, due to implementation limitations, we do not take into account typing information in these particular quick fixes. Finally, checking for validity is also a way of testing our implementation, as an unexpected number of non-valid quick fixes is a smell of bugs in the implementation of the quick fixes.

Table 6 shows the results for the *PNML2PetriNet* transformation. We present this particular transformation of the evaluation to allow its comparison with the results obtained in [36]. We have increased the number of applicable quick fixes in most cases. In particular, we have more variety of quick fixes for problems *E02*, *E03* and *E05* (i.e., the maximum of simultaneous applicable quick fixes is increased while the average is similar or greater). For *E10*, we have given support to pre-condition generation. For the other types of errors, there are also some quick fixes available whose applicability is similar. We have added new quick fixes, like Q18.1, which is very useful to correct style problems. Moreover, this evaluation is more robust than the one in [36], since we can reliably track the fixed and newly generated problems (columns *Fix* and *Gen*) and test the validity of quick fix applications. Altogether, this new evaluation confirms and generalises the previous one.

7.2 Evaluating quick fix impact

We have used the information obtained in the previous experiment to study the impact of quick fixes. We are interested in understanding their side-effects, that is, which types of quick fixes tend to cause certain types of problems, and which ones tend to fix other problems (in addition to the targeted ones).

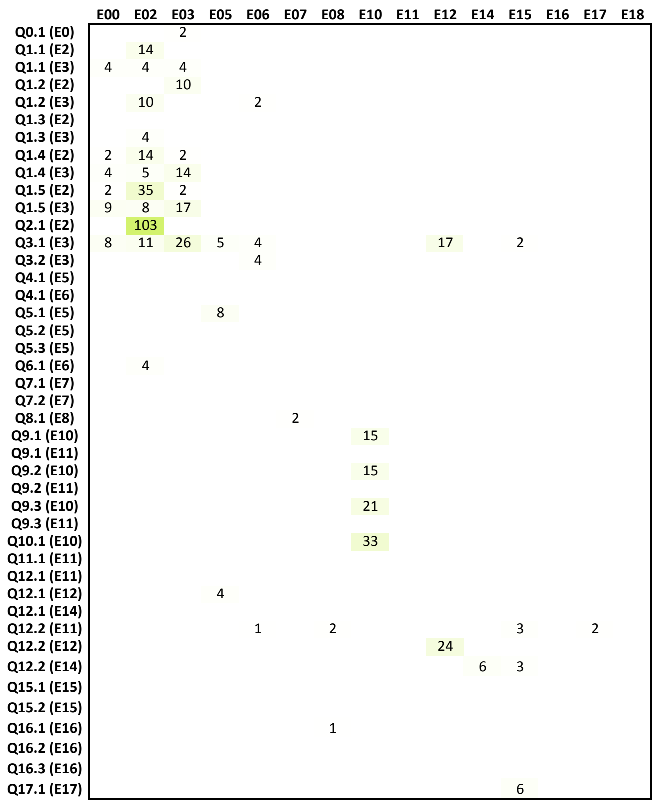
Fixed problems. Sometimes, the application of a quick fix solves other problems different from the targeted one as a side effect. Column *#Extra* in Table 5 shows the number of extra errors fixed (i.e., *#Fixed* - *#Extra* if the number is positive, 0 otherwise). These side-effects are pervasive for some quick fixes, like *Q2.1 Create new rule* which fixes 279 extra problems in 147 applications. Figure 11 shows a heat map representing which types of problems are fixed by a given quick fix when applied to a certain kind of problem. To normalize the data, for each applied quick fix, we count 1 if there is at least one extra problem fixed, and 0 if there is none. We summarize the different impact relationships next.

Table 6 Errors detected in mutants of the *PNML2PetriNet* transformation, and applied fixes.

Prob.	Occ	Qfx	Avg	Min	Max	Val	Fix	Gen
E00	5	5	1.0	1	1	5	5	0
Q0.1	-	5	-	-	-	5	5	0
E02	61	138	2.3	1	5	138	257	149
Q1.2	-	17	-	-	-	17	18	0
Q1.3	-	20	-	-	-	20	21	4
Q1.4	-	20	-	-	-	20	24	0
Q1.5	-	20	-	-	-	20	24	0
Q2.1	-	61	-	-	-	61	170	145
E03	30	164	5.5	4	6	154	275	124
Q1.1	-	12	-	-	-	12	16	4
Q1.2	-	9	-	-	-	9	10	0
Q1.3	-	30	-	-	-	30	30	0
Q1.4	-	30	-	-	-	30	50	8
Q1.5	-	30	-	-	-	30	64	0
Q3.1	-	30	-	-	-	30	90	96
Q3.2	-	23	-	-	-	13	15	16
E05	65	202	3.1	2	4	202	210	0
Q4.1	-	65	-	-	-	65	65	0
Q5.1	-	65	-	-	-	65	73	0
Q5.2	-	45	-	-	-	45	45	0
Q5.3	-	27	-	-	-	27	27	0
E06	9	7	0.8	0	1	7	11	0
Q6.1	-	7	-	-	-	7	11	0
E07	8	16	2.0	2	2	16	24	19
Q7.1	-	8	-	-	-	8	12	9
Q7.2	-	8	-	-	-	8	12	10
E08	9	9	1.0	1	1	9	11	0
Q8.1	-	9	-	-	-	9	11	0
E10	33	124	3.8	3	4	124	184	55
Q10.1	-	33	-	-	-	33	48	0
Q9.1	-	33	-	-	-	33	48	0
Q9.2	-	25	-	-	-	25	40	55
Q9.3	-	33	-	-	-	33	48	0
E12	18	26	1.4	1	2	26	31	18
Q12.1	-	18	-	-	-	18	18	14
Q12.2	-	8	-	-	-	8	13	4
E14	29	29	1.0	0	2	27	33	14
Q12.1	-	16	-	-	-	14	20	1
Q12.2	-	13	-	-	-	13	13	13
E15	36	0	0.0	0	0	0	0	0
E16	14	20	1.4	1	2	6	6	4
Q16.1	-	6	-	-	-	6	6	4
Q16.2	-	14	-	-	-	0	0	0
E17	45	4	0.1	0	1	4	4	0
Q17.1	-	4	-	-	-	4	4	0
E18	6	6	1.0	1	1	6	6	0
Q18.1	-	6	-	-	-	6	6	0

Quick fix Q0.1, which adds additional filters to conflicting rules, may fix *E3 Invalid target for resolved binding* if some guilty rule of *E3* is one of the fixed conflicting rules.

Quick fix Q1.1 implies modifying a rule filter with an additional constraint and can be applied over *E2 Possible unresolved binding* and *E3 Invalid target for resolved binding.* In both cases, when there are problems similar to the one being fixed within the same rule, they will be fixed at once. This pattern appears in *PNML2Petrinet*



	E00	E02	E03	E05	E06	E07	E08	E10	E11	E12	E14	E15	E16	E17	E18
Q0.1 (E0)			2												
Q1.1 (E2)		14													
Q1.1 (E3)	4	4	4												
Q1.2 (E2)			10												
Q1.2 (E3)		10			2										
Q1.3 (E2)															
Q1.3 (E3)		4													
Q1.4 (E2)	2	14	2												
Q1.4 (E3)	4	5	14												
Q1.5 (E2)	2	35	2												
Q1.5 (E3)	9	8	17												
Q2.1 (E2)		103													
Q3.1 (E3)	8	11	26	5	4					17		2			
Q3.2 (E3)					4										
Q4.1 (E5)															
Q4.1 (E6)															
Q5.1 (E5)				8											
Q5.2 (E5)															
Q5.3 (E5)															
Q6.1 (E6)		4													
Q7.1 (E7)															
Q7.2 (E7)															
Q8.1 (E8)						2									
Q9.1 (E10)								15							
Q9.1 (E11)															
Q9.2 (E10)								15							
Q9.2 (E11)															
Q9.3 (E10)								21							
Q9.3 (E11)															
Q10.1 (E10)								33							
Q11.1 (E11)															
Q12.1 (E11)															
Q12.1 (E12)				4											
Q12.1 (E14)															
Q12.2 (E11)					1		2					3		2	
Q12.2 (E12)										24					
Q12.2 (E14)											6	3			
Q15.1 (E15)															
Q15.2 (E15)															
Q16.1 (E16)							1								
Q16.2 (E16)															
Q16.3 (E16)															
Q17.1 (E17)												6			

Fig. 11 Heat map showing the relationships between quick fix types and fixed problems according to the error types. The smallest number of fixed problem is assigned white, while the largest number is assigned dark green.

and *UML2Intalio* in the assignment of features from and to for control flow edges (as in lines 47–48 in Listing 1). This quick fix has also an impact in other bindings that depend on the modified rule. If the impacted binding had problem *E3* and was resolved by the modified rule, the quick fix may solve this problem as well. This possibility is less likely and it has not arisen in our experiment. In some cases, rule conflicts (*E0*) may become fixed as well.

Quick fix Q1.2 removes the problematic binding. If the binding has other binding-related problems, they will become automatically fixed (e.g., *E6*).

Quick fix Q1.3 filters the right-hand part of a binding to constrain the elements that will be resolved. If the quick fix is applied for *E3*, it can fix *E2* as a side effect since the binding filter is ruling out elements that may be the ones causing *E2*. However, if the quick fix is applied over *E2*, it cannot fix *E3* because, in this case, the quick fix is using the resolving rule filters (which may coincide with the guilty rules of an *E3* for the same binding) to force the right-hand part of the binding to select only the elements satisfying them. This provides a hint of the order in which we should fix a binding by using *Q1.3*: first *E3* problems and then *E2* if needed.

Quick fixes Q1.4 and Q1.5 generate a pre-condition. This has an impact on problems that require confirmation using the solver. In particular, it can fix problems *E0*, *E2* and *E3*. However, we have found an undesirable scenario which happens when the generated pre-condition contradicts other problems that require confirmation by the solver. As an example, in one mutant of *PNML2PetriNet*, we are generating the following pre-condition:

```
PNML!NetContentElement.allInstances()→
  forAll(v | v.oclIsKindOf(PNML!Place))
```

This pre-condition is too strong as it rules out every subtype of NetContentElement different from Place, including Transition. Thus, any problem caused by an instance of Transition will be discarded as an actual problem. We are not treating this case in a special manner in the evaluation. Nevertheless, when this happens, the pre-condition is still useful as a template for the user. Moreover, one could use the constraint solver to detect this situation and notify the user.

Quick fix Q2.1 adds a new rule. This tends to solve *E2 Possible unresolved binding* problems related to the input type of the generated rule. The heat map shows that this situation is common.

Quick fix Q3.1 removes a guilty rule detected in problem *E3*. Other problems of type *E3* affected by the same rule may become fixed at the same time. Similarly, a rule conflict problem *E0* may become fixed if the deleted rule is one of the conflicting rules. In addition, any problem within the deleted rule will disappear, no matter its type (*E5*, *E12* and *E15* in the experiment).

Quick fix Q3.2 proposes a target feature with a type compatible with the guilty rules. Our current implementation does not check the feature cardinality and thus it may provoke problem *E06*. This could be easily fixed with *Q6.1*.

Quick fix Q5.1 initialises a compulsory binding with a default value. This quick fix only solves the targeted problem, except in the case of bidirectional references having two compulsory ends, in which case it solves two problems at once. For example, in the following listing, outgoingArc[1..*] and from[1] are not initialised, and two problems are reported (in rules Transition and TransitionToPlace). Since the references are one the opposite of the other, fixing one of them with *Q5.1–Q5.3* solves both problems.

```
rule Transition {
 from n : PNML!Transition
 to p : PetriNet!Transition (
  ...
  -- missing 'outgoingArc' (opposite of from)
 )
}

rule TransitionToPlace {
 from n : PNML!Arc ( ... )
 to p : PetriNet!TransitionToPlace (
  ....
  -- missing 'from' (opposite of outgoingArg)
 )
}
```

Quick fix Q6.1 uses first() to obtain an element of a collection in the right-hand side of a binding, and assign this value to a mono-valued feature. A troubling issue is that our underlying solver (USE Validator) only supports Set types, and hence, first() is not supported. Our interface with the USE Validator uses several workarounds to solve this issue, but sometimes, some results are unreliable[9]. This arises in the experiment in the form of *Q6.1* fixing *E02 Possible unresolved binding*.

Quick fix Q8.1 replaces a declared type by the inferred one when they are not compatible. If the original type declaration refers to a missing class, this is fixed as well (2 fixes for *E07 Invalid type* in the heat map). For example, in the following listing, replacing allArcs with the inferred type Sequence(PNML!Arc) also fixes the reference to InvalidClass.

```
let allArcs : Sequence(PNML!InvalidClass) = PNML!Arc.allInstances()...
```

Quick fixes Q9.1 and Q9.2 add an if statement to an expression that may access an undefined value, or modify the container rule filter to avoid reaching the problematic access, respectively. If there are several nested accesses to the same feature, the quick fix will solve all of them at once. Our current implementation of *Q9.1* puts the if in the outermost location. For instance, for the following code in the *UML2Intalio* transformation, lines 1 and 2 are problematic:

```
1 if anActivity.name.toUpper() = 'MY_ACTIVITY' then
2   anActivity.name + '_mine'
3 else
4   anActivity.name
5 endif
```

If we fix line 2 with *Q9.1*, a new if expression is placed around the first if, thus solving both problems at once (as the following listing shows). Hence, the 15 additional fixes of *E10* for *Q9.1* and *Q9.2* in the heat map. This may also arise when the quick fixes are applied to *E11*, but our mutants have not generated this scenario.

```
if not anActivity.name.oclIsUndefined() then
  if anActivity.name.toUpper() = 'MY_ACTIVITY' then
    anActivity.name + '_mine'
  else
    anActivity.name
  endif
else
  ''
endif
```

Quick fix Q9.3 generates a pre-condition to ensure that the value of a "possibly undefined" property is always defined, solving similar accesses at once. *Q10.1* behaves similarly, but setting the feature lower bound to 0. This latter quick fix is preferred when the problem is not in the transformation, but it lies on the meta-model cardinalities.

Quick fix Q12.1 is a heuristic quick fix which replaces an access to a non-existing feature by an access to an existing feature. If the invalid feature is used in the right-hand side of a binding, the quick fix may solve problem *E05* if it proposes a feature that needs to be initialised.

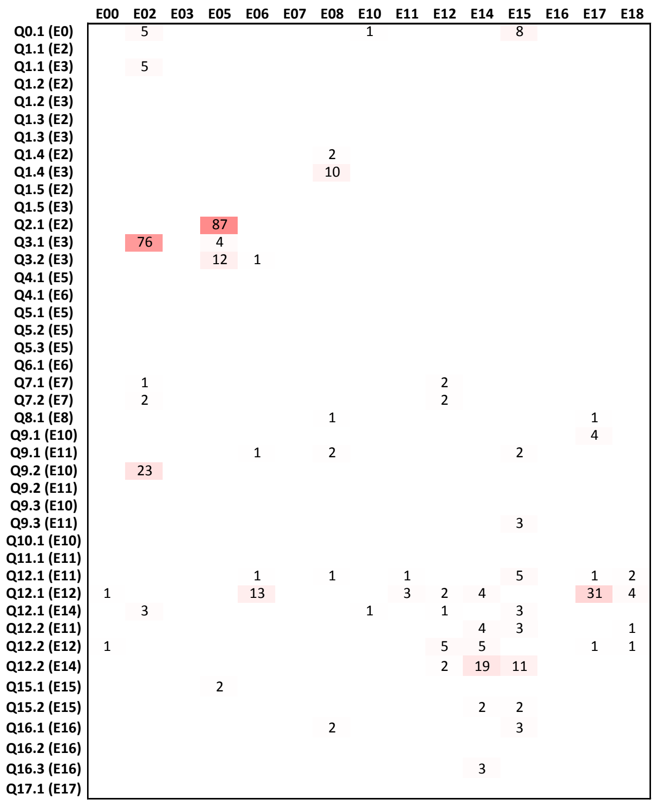
[9] We are not yet able to precisely determine in which circumstances the solver behaves correctly and in which not. Thus, we notify the user that the analysis is not completely reliable.

For example, one of the mutants contained the following piece of code which has problem *E12* because name does not exist in Build. Our system proposed defaultGoal as a possible fix, which fixed *E12* and *E05*.

```
rule AntProject2Maven {
 from a : Ant!Project ( ... )
 to
   -- The mutant replaced MavenProject!Project with Build
   -- Problem E05 in mp because defaultGoal is compulsory
   mp : MavenProject!Build (
     -- Problems E12 because name does not exist
     name <- a.name,
   )
}
```

Quick fix Q12.2 creates a helper that resolves a call to a feature not found, fixing the same problem in other locations.

Generated problems. A quick fix may or may not generate additional problems as a side-effect of its application. As Table 5 shows, some quick fixes never generate new problems while others are more prone to this behaviour. Figure 11 shows a heat map representing which types of problems are generated by a given quick fix when applied to a certain kind of problem. Next, we report the more interesting findings.



	E00	E02	E03	E05	E06	E07	E08	E10	E11	E12	E14	E15	E16	E17	E18
Q0.1 (E0)		5						1				8			
Q1.1 (E2)															
Q1.1 (E3)		5													
Q1.2 (E2)															
Q1.2 (E3)															
Q1.3 (E2)															
Q1.3 (E3)															
Q1.4 (E2)							2								
Q1.4 (E3)							10								
Q1.5 (E2)															
Q1.5 (E3)															
Q2.1 (E2)				87											
Q3.1 (E3)		76		4											
Q3.2 (E3)				12	1										
Q4.1 (E5)															
Q4.1 (E6)															
Q5.1 (E5)															
Q5.2 (E5)															
Q5.3 (E5)															
Q6.1 (E6)															
Q7.1 (E7)		1								2					
Q7.2 (E7)		2								2					
Q8.1 (E8)							1							1	
Q9.1 (E10)														4	
Q9.1 (E11)					1		2					2			
Q9.2 (E10)		23													
Q9.2 (E11)															
Q9.3 (E10)															
Q9.3 (E11)												3			
Q10.1 (E10)															
Q11.1 (E11)															
Q12.1 (E11)					1		1		1			5		1	2
Q12.1 (E12)	1				13				3	2	4			31	4
Q12.1 (E14)		3						1		1		3			
Q12.2 (E11)											4	3			1
Q12.2 (E12)	1									5	5			1	1
Q12.2 (E14)										2	19	11			
Q15.1 (E15)				2											
Q15.2 (E15)											2	2			
Q16.1 (E16)							2					3			
Q16.2 (E16)															
Q16.3 (E16)											3				
Q17.1 (E17)															

Fig. 12 Heat map showing the relationships between quick fix types and generated problems according to error types. The smallest number of fixed problem is assigned white, while the largest number is assigned dark red.

Quick fix Q0.1 solves a rule conflict but tends to introduce additional *E2 Possible unresolved bindings* because the modified rule filters cover fewer cases. Any other problem existing in the conflicting rule filters is duplicated when copying the negated filter's condition. In particular, the mutation operators which caused *E0* are creating other problems in the original transformation which are duplicated by *Q0.1* (e.g., *E15* and *E10* in the heat map). In general, any quick fix that copies pieces of code to another location may have the same issue.

Quick fixes Q1.1–Q1.5 resolve binding–rule problems and do not introduce many problems. *Q1.1* introduces some *E2* errors because changing a rule filter to make it more constrained may have this effect.

Quick fix Q2.1, which adds a new rule, produces many errors due to uninitialized compulsory features. It is worth noting that *Q2.1* does not introduce rule conflicts because our implementation takes care of generating adequate rule filters.

Quick fix Q3.1 deletes guilty rules, and hence, the generation of *E2 Possible unresolved bindings.* A subtle effect of rule removal is that it may cause *E5 Compulsory feature not initialized.* As an example, let us consider this excerpt from one mutant of the *PNML2PetriNet* transformation. Before the removal of TransitionToPlace, the analyser does not raise problem *E5* for outgoingArc because its inverse has already been initialised (from). However, if we remove this rule, we are invalidating that condition and causing problem *E5*.

```
-- Guilty rule. Will be removed by Q3.1
rule TransitionToPlace {
 from n : PNML!Arc ( ... )
 to p : PetriNet!TransitionToPlace (
   -- from is the inverse of outgoingArc
   from ← n.source,
   to ← n.target
  )
}

-- PetriNet!Transition has outgoingArc compulsory feature
rule Transition {
 from n : PNML!Transition
 to PetriNet!Transition
}
```

Quick fix Q3.2 replaces a target feature. If the replaced feature is compulsory, *E5* problems arise. Similarly, if the cardinality of the new target feature is not compatible with the right-hand side of the binding, *E6* problems appear.

Quick fixes Q7.1 and Q7.2 correct references to invalid meta-model types. If the proposed meta-model type is incorrect, it may trigger even more problems. An alternative for these quick fixes would be not to use distance metrics, but to propose the type that is likely to correct more errors.

The behaviour of quick fixes *Q12.x* to *Q16.x* is more unpredictable as they are template quick fixes and some of them are not *valid* (see Table 5). In most cases, in addition to not fixing its error, they introduce additional ones. In other cases, the quick fix may be correcting the problem, but it does not take into account its context. For instance, in a binding feat ← obj.wrongFeature, if *Q12.1* suggests an existing feature for which there is no resolving rule, then *E2 Possible unresolved binding* will be raised. A similar situation occurs if the new proposed feature has primitive type but the binding expected an object (or the other way round). This is the reason for the amount of *E17* in the heat map.

Drawing on the experiment, one outcome is that the behaviour of quick fixes can be roughly classified as *global* and *local.* Global quick fixes like *Q4.1*, *Q2.1*, *Q10.1* and *Q12.2* may fix many errors at once. In turn, global quick fixes can be classified as *predictable* and *non-predictable.* Quick fix *Q10.1* is predictable because it is straightforward to identify the locations that will be affected, whereas *Q2.1* is non-predictable because its impact typically depends on OCL expressions that need to be analysed, likely using a solver. A local quick fix affects a limited scope and its ability to fix many problems at once is small. Examples are *Q5.1–Q5.3*, which will likely fix only the targeted problem, even though it is possible that they fix related problems (e.g., if there are opposite references).

The heat map for fixed problems, Figure 11, reveals strong relationships between rule–binding problems and their associated quick fixes. We believe that this is due to the implicit rule scheduling algorithm of ATL (i.e., all matched rules are matched "at once") and the implicit binding resolution algorithm (i.e., a binding is dynamically resolved according to the runtime type of its right-hand side).

In the case of generated problems, the heat map in Figure 12 is more scattered. There are clear relationships between a few quick fixes and their generated problems, like *Q2.1* and *E5*, *Q3.1* and *E2* or *Q9.2* and *E2*. However, for other quick fixes, typically heuristic and template ones like *Q12.x*, the quick fix behaviour is less clear.

Altogether, this study provides an understanding of the effects of applying quick fixes to ATL transformations and it is a cornerstone to build a hierarchy of quick fixes with less or more constrained application conditions to fix/generate errors.

7.3 Evaluating usefulness of catalogue and ranking

Next, we report on an experiment to evaluate the usefulness of the system to fix errors in real transformations. For this purpose, we selected 12 transformations from the ATL zoo (i.e., developed by a third party), whose details are shown in Table 7. The transformations range from small (e.g., Book2Publication with 32 LOC and 1 rule) to medium-size (e.g., Mantis2XML with 505 LOC and 5 rules, or Maven2Ant with 273 LOC and 30 rules). In addition to cover this size range, we selected transformations with meta-models or domains close to the knowledge of the participant developers, and favoured transformations with a low number of errors, or at least, a low number of error types (e.g., Bugzilla2XML and Mantis2XML have many errors but of the same kind).

Table 7 Transformations used for the *usefulness* experiment.

Transformation	LOC.	Rules	Helpers	Classes	Num. errors	Error types
BibText2DocBook	171	9	4	21/8	4	E5
Book2Publication	32	1	3	2/1	1	E18
Bugzilla2XML	422	8	7	9/5	82	E5
Class2Relational	87	6	1	5/4	2	E18, E8
Ecore2Class	30	3	0	18/5	2	E2, E10
Families2Persons	42	2	2	2/3	1	E10
JavaSource2Table	59	4	2	5/3	1	E8
KM32EMF	125	10	1	15/20	3	E2, E18, E5
KM32SimpleClass	56	4	0	16/5	9	E2(2), E3, E18(6)
Mantis2XML	505	5	1	10/5	97	E5(95), E17(2)
Maven2Ant	273	30	4	59/48	7	E2, E8(2), E10(2), E12, E18
XML2Book	27	2	1	5/2	4	E2, E3, E10, E18

Altogether, the selected 12 transformations have 213 errors (all detected statically and confirmed by the solver when needed), and the types of errors found cover reasonably well the range of possible errors our analyser is able to detect (see Figure 6).

The goal of the experiment was twofold. On the one hand, to check to what extent real faulty transformations can be fixed *according to the intention* of the developers, using quick fixes of the catalogue. On the other hand, to evaluate the utility of the two rankings (dynamic and static) with respect to the choices made by the developer.

The experiment was carried out by two developers in an independent way. In a first step, they were asked to explain how to solve each one of the identified errors *without* the assistance of our quick fixes. Then, in a second step, they were allowed to use the catalogue of quick fixes to solve the same problems, and were asked whether there was some quick fix equivalent to the solution they had originally devised, the position of the quick fix in the ranking, as well as whether the result of the fix was the one they expected.

After the experiment, the data analysis revealed that both developers agreed on how to resolve 32 problems (which is less than the total number of errors because resolving one issue may resolve others as a side effect), while they disagreed on the resolution of 6 issues in 3 different transformations. The disagreements concerned the resolution of errors *E2* (*possible unresolved binding*) and *E3* (*invalid target for resolved binding*). Both errors required considering the possibility of receiving input models with unexpected features for the transformation, which could hardly be mapped to the target model. While one developer decided to use *Q1.5* and *Q1.4* in these cases (generate general pre-condition, or generate pre-condition), the other decided to use *Q1.1* and *Q1.3* (modify rule filter, or add filter to binding expression). The strategy of the first developer results in not transforming the problematic models, while the strategy of the second results in transforming only the parts of the model that are considered by the transformation logic, but neglecting the problematic parts.

Overall, each developer could not find a suitable quick fix in 2 cases, in order to solve the same 2 problems, both in the Maven2Ant transformation. In the first case (*E12 Feature not found*) a binding mentioned an undefined property PropertyName.value. Here, the developers solution was to remove the binding (as the error seemed a copy/paste error from a similar rule), but the system only suggested choosing a similar feature (*Q12.1*). In the other case (*E8 Declaration mismatch*), the system proposed changing the declared type of a helper by the inferred type (*Q8.1*), while in this case, the developers choice was to modify the helper body by adding flatten operators to the returned collections.

Developers also recorded whether the quick fix applications had the consequences they expected. This occurred in all cases but two (the same situation was noticed by both developers). The first case occurred when fixing an *E10 Possible access to undefined property* error with quick fix *Q9.1*, in an expression deeply nested in a conditional. This resulted in a new if condition enclosing the existing conditional, while the developers' expectation was that the condition should have been added to the branch of the conditional containing the error. In the second case, fixing an *E5 Compulsory feature not initialized* error with quick fix *Q5.1* resulted in the initialization of a binding with OclUndefined. However, both developers were expecting that the fix would also create a default object in the output pattern of the rule, and that this object were assigned in the binding instead of OclUndefined.

Altogether, both developers could reasonably fix most errors in the selected 12 transformations. All repair actions (except 2) could be performed using quick fixes of the catalogue, while in all cases (except 2) this resulted in the expected behaviour.

Finally, we compare the static and dynamic rankings provided by our approach, by analysing whether the quick fixes selected by the developers in our experiment are located in top positions in both rankings. Table 8 shows the results. This table contains aggregated information of the errors and quick fixes applied by both developers (first column); the average and median of the selected quick fix as calculated dynamically by the speculative analysis (second column); the average and median of the selected quick fix as calculated by the static ranking (third column); the number of quick fixes of each kind applied (fourth column); and the number of errors fixed by the quick fix application (fifth column).

Table 8 Comparing dynamic and static rankings of quick fixes. *Dyn. rank* and *Static rank* columns contain the average and median (with format average/median) of the ranks of the selected quick fixes.

Errors & Quick fixes	Dyn. rank	Static rank	# Quick fixes	# Fixed errors
Possible unresolved binding (E2)				
Q1.1	4/4	3/ 3	2	2
Q1.3	1,2/1	4,5/5	6	6
Q1.4	2,5/2,5	4/4	2	3
Q1.5	1/1	1/1	2	3
Invalid target for resolved binding (E3)				
Q1.1	2/2	5/ 5	1	1
Q1.5	1/1	4/ 4	1	1
Compulsory feature not initialized (E5)				
Q4.1	1/1	2/ 2	6	362
Q5.1	1/1	1/ 1	2	2
Declaration mismatch (E8)				
Q8.1	1/1	1/ 1	4	4
Possible access to undefined property (E10)				
Q9.1	1,4/1	1,8/ 2	10	10
Incompatible types (E17)				
Q17.1	1/1	1/ 1	4	4
Style warnings (E18)				
Q18.1	1/1	1/ 1	22	22

Normally, developers tend to look at the first one or two options of the list of offered fixes. Therefore, we analyse to what extent the choice made by the developers in the experiment was offered in the first or second positions by both rankings. We can observe that the fix chosen by the developers was the first offered by the dynamic ranking in 88,7% of the cases (55 times), while it was the first of the static ranking in 36% of the cases (36 times). We can also see that the developers chose the first or second option of the dynamic ranking in 92% of the cases (57 times), while the choice was first or second in the static ranking in 82% of the cases. Hence, these high percentages show the appropriateness of the rankings for practical use.

The ranking provided by speculative analysis is generally more accurate (regarding how the choice of the developers scored in both rankings). As Table 8 shows, the average and median of the positions in the dynamic ranking is always lower or equal than for the static ranking, with the exception of *Q1.1* for error *E2*. More in detail, the dynamic rank was strictly better than the static one in 32% of the cases (20 times). The static ranking was as good as the dynamic one in 63% of the cases (39 times), while it was better in 4,8% of the cases (3 times). These are reasonable results, as speculative analysis checks the consequence of every quick fix, while the static ranking is made by empirical analysis. However, being as good as the dynamic over 60% of the times indicates that the model of the static ranking is an adequate default. It must be stressed that the advantage of using speculative analysis is not only its more accurate ranking, but on the wealth of extra information offered to the developer, as explained in Section 5. On the other hand, the ranking offered by the static ranking is accessible in a quicker, more agile way through the standard quick fix tool, which does not require from opening a separate dialog window, with less disruption of the programming flow.

7.4 Threats to validity

The main threat to the internal validity of the experiment to determine the validity and completeness of our catalogue (Section 7.1) is that our analyser may report some false positives (i.e., indicate an error incorrectly). In this case, we may be cataloguing a correct quick fix application as invalid. The analyser may also report some false negatives (i.e., fail to report a true error) which may lead to marking a quick fix application as valid when it is not. In our experience, the analyser has a low rate of false positives/negatives (we present a preliminary evaluation in [35]). Nevertheless, we have manually checked any suspicious result and performed many tests to try to avoid this situation. In addition, our analyser uses a model finder (USE) based on the "small scope hypothesis", which limits the search of witness models to a given scope. To minimise the number of false negatives due to this reason, we have used reasonably wide searching scopes.

Related to the previous issue, 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 after applying a quick fix is also performed using anATLyzer as oracle function. Unfortunately, to the best of our knowledge, there is no other analyser of ATL with which we can cross-validate the results. This implies that different versions of anATLyzer (e.g., due to bug fixes) may provide slightly different values for the experiments and for our quick fix ranking.

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 from the quick fixes. Our solution to this threat has led to a different threat, which is that our mutation operators are not exercising all the quick fixes. For instance, *Q12.4 Change feature call to operation call (and vice versa)* is never applied. In the same line, another threat is related to the size and complexity of the transformations used in the experiment. We have used four different transformations with different characteristics but we cannot claim that all ATL features are present in these transformations. Nevertheless, we have checked that all "transformation-specific" quick fixes are exercised at least once.

Regarding the internal validity of our evaluation of the impact of quick fixes, one threat is that we are using just first-order mutations. For instance, we know from our manual evaluation in Section 5.3 that *Q4.1 Modify feature cardinality in meta-model* will fix any *E04 Feature initialization* problem for the same meta-model feature. However, Figure 12 does not reveal this because we are mutating a correct transformation in only one place, while we would need to remove at least two bindings to enable this effect. For this, we would need at least second-order mutation operators.

Regarding the experiment in Section 7.3, the main threat to its internal validity is the possible bias of the developers towards using quick fixes available in the tool, but which would not repair the transformation as they planned. To mitigate this risk, there were two developers (as opposed to the experiment in [36], made by one developer only), and their large agreement in the selection of how to fix the 12 transformations is an indication that they were not biased. As for the external validity, we might extend the experiment with more transformations and more developers, to obtain data of more types of errors and quick fixes. Also, the experiment was designed to fix already finished (and hopefully tested) transformations, for which we favoured transformations with few errors. However, the results of the experiment cannot be generalized to the scenario of creating a transformation from scratch, for which another experiment would be needed.

For all experiments, there is an internal threat related to experimenter bias because the experiments have been carried out by the authors. To minimize the impact, we have split the work (i.e., analyser implementation vs. mutation operator implementation) and we have selected transformations written by third-parties. There is also an external threat regarding the generalization to other transformation languages, as we only cover ATL. The features of other transformation languages may limit or impose additional constraints in some of the proposed 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.

7.5 Discussion

Next, we comment on our experience using our quick fix system, beyond the previous empirical analysis.

We have found the system useful to fix many types of problems, as confirmed by our experiments. However, from a usability perspective, sometimes the generated code may be difficult to understand. This is particularly the case of quick fixes for rule resolution, which 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, and we have quick fix variants to encapsulate complex expressions. In contrast to quick fixes for e.g., Java, our code is inherently more complex (i.e., some of our quick fixes copy and adapt pieces of code from other locations). Hence, more optimizations need to be studied.

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 generally 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.

We have found our speculative analysis useful to reason about the consequences of a quick fix. In particular, it uncovers information that is many times unknown by the developer, like the implicit effects of modifying a piece of code (even a simple one). We have thoroughly studied these effects in Section 7.2.

Regarding performance, our system is responsive enough to be used as an editing facility. Using the model finder is time-consuming, but we mitigate this problem by pruning the input meta-model into a so-called *error meta-model*, as discussed in [35]. With respect to the speculative analysis, it typically requires some time to complete the results (e.g., around 5 seconds). The total time depends on two main factors. First, the size of the transformation because its abstract syntax model is copied once per quick fix. While this process is typically fast, it may impact the overall performance in case of very large transformations. Strategies such as copy-on-demand could be implemented to improve performance. The second and dominant factor is the number of problems that require solver confirmation, since the execution time of the solver varies from a few milliseconds to a few seconds. Nevertheless, our implementation does not block the user interaction but it runs the analysis in several execution threads.

Concerning the applicability of our proposal to other languages beyond ATL, we believe it is conceptually applicable to any transformation language, especially to OCL-based ones. In practice, there are three dimensions that should be considered to determine the effort required to implement our system for other transformation languages, namely: architecture, static analysis, and applicability of the catalogue of quick fixes.

The architecture of our system could be applied to any transformation language, notably if it is built as an Eclipse plug-in. The only requirement is that the abstract syntax tree of the transformation definitions needs to be available for the quick fixes, since our speculative analysis works at the abstract syntax level and not at the text level.

On the other hand, if the static analysis provided by a language is powerful, more advanced and precise quick fixes can be implemented. Existing transformation languages vary between *strongly typed languages* such as Kermeta [14] and QVT [32], where all types are resolved at compile time and the abstract syntax is annotated with type information, and *dynamically typed languages* such as ATL and ETL [18], where type checking is performed at runtime. Unfortunately, these languages do not make available valuable information such as rule dependencies and helper invocations, which we expose in our TDG. Our static analyser is specific to ATL, thus it cannot be directly reused for other languages, which limits the adoption of our catalogue by other languages.

Finally, regarding the general applicability of our catalogue of quick fixes, next we discuss which quick fixes are applicable to different model transformation languages. We will assume that each language provides (or could provide) a static analyser.

ETL [18] is very similar to ATL. Hence, our catalogue is easily applicable to ETL with minor adaptations. However, the fact that ETL has less constrained imperative features may make the implementation more difficult, while OCL-related quick fixes may need to be adapted to the query language of ETL (EOL).

Regarding the QVT languages, both QVT Operational (QVTo) and QVT Relational (QVTr) use OCL to navigate models, hence, quick fixes related to OCL typing are applicable. QVTo does not support the implicit execution of rules; thus, quick fixes related to rule resolution are not directly applicable, although they could be adapted to handle problems related to incomplete mapping rules. Quick fixes for rule conflicts could also be adapted to ensure the disjunction of when clauses, while binding-related quick fixes could be adapted to ensure proper initialization of features. In the case of QVTr, top rules are akin to ATL matched rules, while non-top rules are lazy. Quick fixes for rule conflicts and feature initialization could also be adapted to QVTr as in QVTo.

Finally, our catalogue is less applicable to languages which do not use OCL, such as graph-based languages like Viatra [37] and Henshin [2]. Nevertheless, these languages could reuse some of our ideas, such us quick fixes for rule conflicts and feature initialization.

8 Related Work

In this section, we focus on recent research on code recommenders, quick fixes, and automated program repair. We leave out from this review works on fault localization and validation&verification of model transformations, and redirect the interested reader to [33,35] for a revision on these topics. Most of the works we analyse come from the programming languages community, as works dealing with quick fixing or repairing model transformations are virtually non-existent.

8.1 Quick fixes and code recommenders

Different strategies for proposing and ranking quick fixes have been studied in the programming community. For example, in [28], quick fixes are ranked according to the number of errors that remain after their application. MintHint [16] uses statistical correlation analysis to identify expressions that are likely to appear in patches. BugFix [13] uses ideas from machine learning to automatically learn from previous bugs that have been fixed over time, in order to report a prioritized list of relevant fix suggestions when new bugs are detected. Fix suggestions are textual descriptions of the changes needed to remove a bug, and so they must be manually encoded by the developer. Instead, our fixes can be applied automatically to solve a detected problem, and similar to these approaches, we rank them statically according to their efficacy on a set of transformations, but also dynamically using speculative analysis.

Our tooling includes a facility to visualize the result of applying a quick fix speculatively. This allows analysing the impact of a fix before its application. Other systems with similar functionality include the one presented in [27], which provides a semi-automatic wizard to see the result of possible fix alternatives for buffer overflows in C code. Similar to our approach, the system uses SMT solving to assert whether a buffer overflow can ever occur in practice, given a potentially faulty statement. Its quick fixes are empty C code skeletons where certain values, which are computed by the SMT solver, limit the index variables to take only values within the buffer range. On a different area, GoalDebug [1] is a debugging system for spreadsheets where users can report expected values for cells that yield an incorrect value, and the system generates change suggestions ranked according to a number of heuristics. Change suggestions can be interactively explored, applied or rejected.

Altogether, although there are previous works tackling the generation and ranking of fixes for different areas, there are few works on this topic in the MDE literature, and none tackling model transformations that we are aware of.

Solutions for quick fix generation have also been applied to Domain-Specific Modelling Languages (DSMLs). For example, [12] 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, quick fixes implement pre-defined correction strategies, and we rank them according to the problems they introduce.

8.2 Fixing errors in model transformations

Although many efforts have been devoted to the analysis and verification of model transformations in recent years [33], works for their automated or assisted repair are scarce.

In [4], the authors synthesize OCL pre-conditions for graph transformation rules from meta-model integrity constraints. The generated pre-conditions ensure rule correctness (i.e., the rule application always yields a model conformant to its meta-model). The work [7] tackles the same problem. In [17], the authors define the necessary conditions that graph transformations should fulfil to ensure the satisfaction of containment constraints. Though these approaches do not include a fault localization phase, they allow fixing potential errors *a priori*. However, the kind of detected errors is limited (meta-model conformance or graph constraint satisfiability), and they are restricted to graph transformation. In our case, the challenge is bigger as ATL is dynamically typed and more expressive.

In [20,39], 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.

The catalogue of refactorings for model-to-model transformations presented in [40] aims at improving transformation quality. We believe our method can be applicable to the automated refactoring of ATL transformations, but we leave this aspect for future work.

8.3 Automated program repair

Automated program repair [22] aims at correcting faulty programs automatically, where faults are detected by the dynamic testing of the programs. By relying on dynamic testing, and different patch search heuristics, program repair is typically a very time-consuming task. Instead, our quick fixes are a much lighter technique, aimed to be used interactively, and solving localized problems detected statically. While automated transformation repair is left for future work, we took inspiration from existing works in this area, which we describe next.

Some representative works in this area include Autofix [29] (for Objective-C), which uses pre/post-conditions and invariants to synthetize repairs; NOPOL [8] which represents traces from successful test executions as an SMT problem, whose solution can be translated into a source code patch; and GenProg [21, 23,38], which uses genetic-programming to guide the repair process. From GenProg, we adapted the heuristic for quick fix *Q5.2*.

Martinez and Monperrus [25] use repair models extracted from the analysis of real patches in software repositories, and decorated by a probability distribution that enables reasoning on the search space of program repair. We used this work as inspiration for our static ranking of quick fixes. However, we did not have access to real patches produced by ATL developers. Instead, our quick fix ranking model was heuristically built from automatically applied quick fixes on mutated transformations.

Works in automated program repair focus on general purpose programming language, and handle general problems for programming languages like infinite loops, memory allocation errors, overflows or underflows [23,24, 27,30]. Instead, we focus on the ATL model transformation language. To the best of our knowledge, ours is the first work targeting the generation of fixes for a model transformation language. Hence, the range of problems we are able to fix are transformation-specific on the one hand, and on the other hand, type-related problems due to the dynamic nature of ATL.

9 Conclusions and Future Work

In this paper, 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. We have developed a technique to perform speculative analysis, which provides information on the impact of the application of each applicable quick fix. Speculative analysis provides a dynamic quick fix rank, but in addition, we have constructed a static ranking empirically by the automated application of quick fixes on transformations. We have evaluated several aspects of our approach. First, its validity and completeness, by taking a large set of mutated transformations. In this set, we show that our catalogue covers a wide range of problems, and that the quick fixes actually fix most of the problems. Then, a second experiment has shown the usefulness of our proposal by comparing the repair actions of developers with respect to the available quick fixes and their rankings. The implementation of the tool and the detailed results of the evaluation are available at `http://miso.es/qfx` and `http://miso.es/qfx_exp_sosym2015` respectively.

To improve the recommendation aspect of the system, we plan to extend our current static model by taking into account quick fixes previously selected by the user. We also plan to tackle automated transformation repair by applying sequences of quick fixes and providing different search heuristics. The quick fixes of our catalogue are directed to syntactically fix a transformation, but they do not consider semantic issues (the intent of the transformation developer). In this way, we plan to use transformation *contracts*, like those provided by PaMoMo [10,11], as an oracle to test transformation fixes, and then complement the static analysis with dynamic testing. Finally, in order to provide stronger evidence of the usefulness of the approach, in particular when developers are defining ATL transformations from scratch, we also plan to perform a controlled experiment with users.

Acknowledgements

Work supported by the Spanish Ministry of Economy and Competitivity (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. R. Abraham and M. Erwig. GoalDebug: A spreadsheet debugger for end users. In *ICSE*, pages 251–260. IEEE Computer Society, 2007.
2. T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. In *MoDELS*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.
3. Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: exploring future development states of software. In *FoSER*, pages 59–64. ACM, 2010.
4. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Synthesis of OCL pre-conditions for graph transformation rules. In *ICMT*, volume 6142 of *LNCS*, pages 45–60. Springer, 2010.
5. M. A. Cibran. Translating BPMN models into UML activities. In *Business Process Management Workshops*, pages 236–247. Springer, 2009.
6. 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.
7. F. Deckwerth and G. Varró. Attribute handling for generating preconditions from graph constraints. In *ICGT*, volume 8571 of *LNCS*, pages 81–96. Springer, 2014.
8. F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA*, pages 30–39. ACM, 2014.
9. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.
10. E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013.
11. E. Guerra and M. Soeken. Specification-driven model transformation testing. *Software and System Modeling*, 14(2):623–644, 2015.
12. Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, and D. Varró. Quick fix generation for DSMLs. In *VL/HCC*, pages 17–24. IEEE, 2011.
13. D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *ICPC*, pages 70–79. IEEE Computer Society, 2009.
14. J.-M. Jézéquel, O. Barais, and F. Fleurey. Model driven language engineering with kermeta. In *GTTSE'09*, volume 6491 of *LNCS*, pages 201–221. Springer, 2011.
15. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comp. Programming*, 72(1):31–39, 2008.
16. S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. MintHint: automated synthesis of repair hints. In *ICSE*, pages 266–276. ACM, 2014.
17. C. Köhler, H. Lewin, and G. Taentzer. Ensuring containment constraints in graph-based model transformation approaches. *ECEASST*, 6, 2007.
18. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
19. 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.
20. 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.
21. C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, pages 3–13. IEEE, 2012.
22. C. Le Goues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.
23. 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.
24. F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA*, pages 133–146. ACM, 2012.
25. M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.
26. M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches. In *ICSE*, pages 492–495. ACM, 2014.
27. P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert. Automated generation of buffer overflow quick fixes using symbolic execution and SMT. In *SAFECOMP*, volume 9337 of *LNCS*, pages 441–456. Springer, 2015.
28. 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.
29. 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.
30. J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102. ACM, 2009.

31. S. Proksch, S. Amann, and M. Mezini. Towards standardized evaluation of developer-assistance tools. In *RSSE*, pages 14–18. ACM, 2014.
32. QVT. `http://www.omg.org/spec/QVT/`.
33. L. A. Rahim and J. Whittle. A survey of approaches for verifying model transformations. *Software and System Modeling*, 14(2):1003–1028, 2015.
34. M. P. Robillard, R. J. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.
35. J. Sánchez 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.
36. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Quick fixing ATL model transformations. In *MoDELS*, pages 146–155. IEEE, 2015.
37. D. Varró and A. Balogh. The model transformation language of the viatra2 framework. *Science of Computer Programming*, 68(3):214–234, 2007.
38. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374. IEEE, 2009.
39. 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.
40. M. Wimmer, S. Perez, F. Jouault, and J. Cabot. A catalogue of refactorings for model-to-model transformations. *JOT*, 11(2):1–40, 2012.