JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Fixing Multiple Type Errors in Model Transformations with Alternative Oracles to Test Cases

Zahra VaraminyBahnemiry, Jessie Galasso, and Houari Sahraoui
University of Montreal,Montreal, Canada

ABSTRACT

This paper addresses the issue of correcting type errors in model transformations in realistic scenarios where neither predefined patches nor behavior-safe guards such as test suites are available. Instead of using predefined patches targeting isolated errors of specific categories, we propose to explore the space of possible patches by combining basic edit operations for model transformation programs. To guide the search, we define two families of objectives: one to limit the number of type errors and the other to minimize the alteration of the transformations' behavior. To approximate the latter, we study two objectives: minimizing the number of changes and keeping the changes local. Additionally, we define four heuristics to refine candidate patches to increase the likelihood of correcting type errors while limiting behavior deviations. We implemented our approach for the ATL language using the evolutionary algorithm NSGA-II, and performed an evaluation based on three published case studies. The evaluation results show that our approach was able to automatically correct on average more than 82% of type errors for two cases and more than 56% for the third case.

KEYWORDS Model transformations, Program repair, multiobjective optimization.

1. Introduction

Model-Driven Engineering (MDE) is increasingly used for product development in industries like automotive, telecom or banking (Whittle et al. 2014). In those industries, the primary interest in modeling recently shifted from producing complex models – mainly for documenting software systems – to using these models to (semi-)automatically generate software artifacts by means of model transformations (Combemale et al. 2016).

Model transformations usually take as input models expressed in a modeling language (i.e., metamodel), which can be of general-purpose (e.g., UML) or domain-specific (e.g., AUTOSAR for automotive systems). The outputs of model transformations can be either models (possibly conforming to different metamodels), or texts such as source code or XML documents. In this paper, we focus on the former, i.e., model-to-model transformations. Model transformation programs can be written in general programming languages or transformation-dedicated languages such as ATL (Jouault et al. 2008). These programs usually describe transformation rules that indicate how to transform elements of the input models into elements of the output models. Whether they are learned automatically from examples, like in (Baki & Sahraoui 2016), or written manually, these transformations must be checked to ensure they are free of errors. Transformation languages such as ATL are dynamically typed, making transformation programs particularly prone to type errors, such as referring to elements that do not exist in the metamodels, or initializing properties with values of the wrong types.

A way to automatically correct type errors is to provide predefined patches for each category of errors (Cuadrado et al. 2018). Although this approach may be useful for developers, it suffers from two limitations. Firstly, predefined patches require an intensive knowledge to modify them or to define new ones (e.g., for new categories of errors). Secondly, they fix errors individually without considering possible interactions between

JOT reference format:
Zahra VaraminyBahnemiry, Jessie Galasso, and Houari Sahraoui. *Fixing Multiple Type Errors in Model Transformations with Alternative Oracles to Test Cases*. Journal of Object Technology. Vol. 20, No. 3, 2021.
http://dx.doi.org/10.5381/jot.2021.20.3.a9

them (Cuadrado et al. 2018) and may thus introduce new errors while trying to fix existing ones. Another way to tackle the correction of type errors is to use automatic program repair techniques such as search-based algorithms. These techniques have been proven to efficiently support developers for debugging and correction tasks (Monperrus 2018). Contrary to predefined patches, they enable to explore a space of potential patches, and may help overcome the aforementioned limitations. These techniques closely relate to oracles checking whether the program behavior is correct after applying a patch, test suites being the most popular oracles (Monperrus 2018). A transformation program has an incorrect behavior (i.e., suffers from behavior deviations or "semantic errors") if it produces output models that are not the expected ones. However, a substantial amount of knowledge is required to provide representative test suites that would constitute a relevant oracle (Staats et al. 2011), especially for transformations that use complex structures as input/output. Moreover, in the specific context of type errors, valuable patch solutions may fix most of the errors but not all, and the resulting transformation cannot be executed – and then be tested – as type errors are syntactic errors. Relying on test suites to guarantee that type error patches preserve a transformation behavior is thus hardly possible.

In this paper, we define an automated method for patch recommendation fixing type errors in model transformation programs without relying on predefined patches nor test suites. This method does not seek end-to-end correction, but rather to alleviate developers' tasks by avoiding patch maintenance and test suites definition. Thus, its goal is to recommend to developers patches correcting the most errors possible while minimizing the alteration of the behavior of a given faulty transformation. In a first phase, we propose to explore the space of possible combinations of basic edit operations to find the sequences (i.e., patches) that repair several type errors simultaneously. To limit the transformation's behavior deviation, we explore the idea of using several objectives to guide the search, as surrogate to test oracles. We test two objectives: a) minimizing the changes introduced by the patches and b) preserving the transformation footprint with respect to the involved input/output languages. We analysed the behavior of faulty transformation programs corrected by this first phase and identify four types of recurring behavior deviations, along with the edit operations introducing them, which may be prevented by following simple guidelines. However, implementing these guidelines in objectives would be too resource-consuming and make the method non-tractable. Thus, we define four heuristics to improve the decisions made during the exploration phase and apply them once, in a second phase, on the best patches obtained in the first phase, to further prevent possible behavior deviations.

We evaluate these two phases using three existing ATL model-to-model transformations, with a published dataset containing several mutations of these transformations with various errors and error categories. The evaluation of the first phase showed contrasting results: while we succeeded to correctly fix, on average, respectively 80% and 73% of the type errors while preserving a correct behavior for two transformations, this correction rate was lower (36%) for the third transformation. However, after applying the heuristics during the second phase, the correction rates increased, on average, to 83%, 82% and 57%, respectively.

We made the following contributions:

- We adapt an evolutionary algorithm to automatically generate patches which can fix several type errors at the same time in model transformation programs;
- We show that two objectives (namely, minimizing the changes and keeping the changes local) help to guide the patch generation to limit the behavior deviation of a corrected model transformation program;
- We define four heuristics to refine the obtained patches and show that these heuristics further limit behavior deviation.

The remainder of this paper is organized as follows. Section 2 gives the necessary background and discusses issues related to automatically fixing type errors in model transformations. Section 3 describes the two-step approach to fix type errors without predefined patches nor test cases. An implementation and an evaluation of our approach are provided in Section 4. Section 5 presents related work. We discuss our findings and conclude in Section 6.

2. Background

In this section, we first give some background information about ATL and type errors in ATL transformation programs. Then, we discuss the challenges of repairing these transformations. Finally, we present NSGA-II (Deb et al. 2000), the evolutionary algorithm we use in our approach.

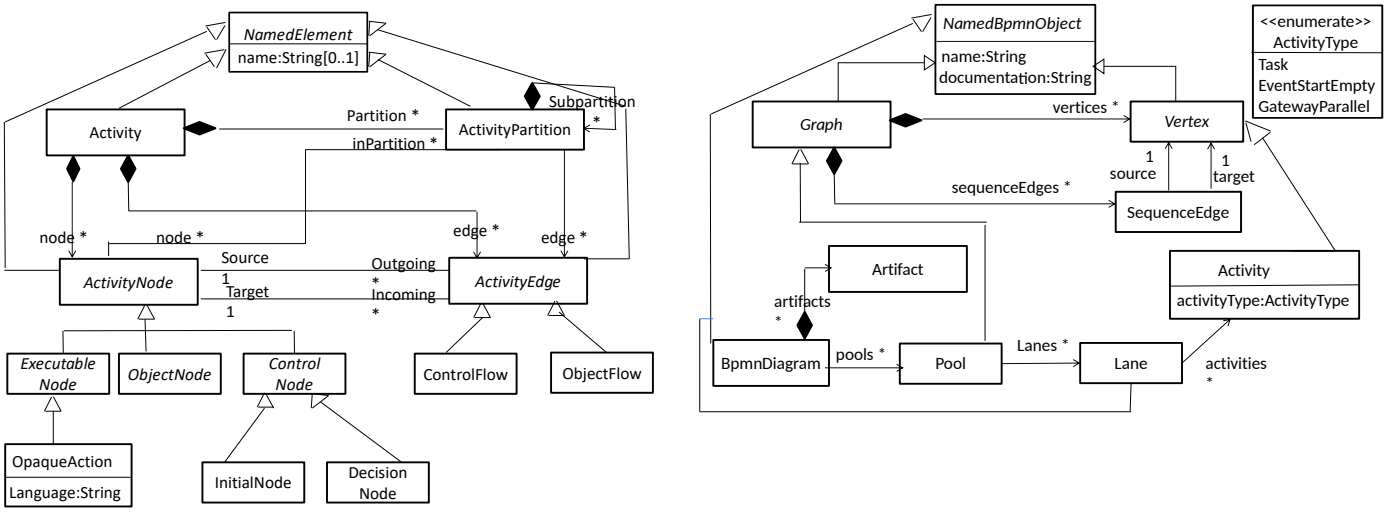
2.1. Type Errors in ATL Transformations

Listing 1 presents an excerpt of an ATL transformation program of UML activity diagrams into Intalio business process models[1], borrowed from (Cuadrado et al. 2018). The two metamodels are shown in Fig. 1.

ATL transformation programs consist in a source metamodel (`IN`), a target metamodel (`OUT`), and a set of transformation `rules`. Each rule is named and describes a pattern in the source metamodel (`from` part, also called the input pattern) and a pattern in the target metamodel (`to` part, also called the output pattern). An ATL transformation program uses an execution mechanism triggering a rule when an object in the input model matches the input pattern of the rule. When the rule is executed, an object is created in the output model according to the output pattern of the rule. For example, the rule `activity2diagram` (lines 7-12) states that each object instance of the `Activity` class of UML (line 8) triggers the creation of an object, instance of the `BpmnDiagram` class of Intalio (line 9).

```
1 create OUT : Intalio from IN : UML;
2 ...
3 helper context UML!Activity def: allPartitions
4    :Sequence(UML!ActivityPartition) =
5    self.partition->collect(p |  p.allPartitions)->flatten();
6
7 rule activity2diagram {
8   from    a : UML!Activity
9   to      d : Intalio!BpmnDiagram (
```

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



(a) UML AD metamodel **(b)** Intalio BPMN metamodel

Figure 1 Excerpts of UML activity diagrams (AD) metamodel and Intalio Business process model (BPMN) metamodel

```
10              artifacts <- a.name,
11              pools <- a.allPartitions
12          )}
13
14  rule activitypartition2pool {
15  from    a : UML!Comment
16  to      p : Intalio!Pool,
17          l : Intalio!Lane (
18              activities <- a.node->reject(
19                  e|e.oclIsKindOf(UML!ObjectNode))
20          )}
21  ...
```

Listing 1 Excerpt of an ATL transformation program, from UML Activity Diagram to Intalio BPMN

An input object may trigger the creation of several output objects. For instance, the rule `activitypartition2pool` (lines 14-20) states that each object instance of the `Comment` class of UML (line 15) triggers the creation of two objects in the output model: one instance of the `Pool` class of Intalio (line 16) and the other instance of the `Lane` class of Intalio (line 17). Input and output objects are related by a trace link: it is possible to access properties of the input object and to set those of the output object. For instance, the rule `activity2diagram` initializes the properties *artifacts* and *pools* of `BpmnDiagram` depending on properties it accesses in `Activity` (lines 10-11). Property initialization, called *binding* in ATL, may use a property of the input object (line 10), a `helper` (similar to methods, as defined in lines 3-5) to reshape the input object property (line 11), or OCL expressions (lines 18-19). Properties can be attributes with native types, or references towards objects. When a binding's right-hand side (RHS) is a reference to an object of the input model, it needs to be transformed into elements of the output model to be assigned to the property of the left-hand side (LHS). In this case, a binding resolution mechanism takes place to retrieve the corresponding elements of the output models. It relies on rules which can perform this transformation, i.e., with a `from` part corresponding to the type of the input model object (binding's RHS), and a `to` part corresponding to the type of the output model property (binding's LHS).

Models are primary artifacts that are exploited through model transformations (Sendall & Kozaczynski 2003). Transformations use a type system mostly defined by the source and target metamodels, *i.e.*, the input and output pattern elements in transformations have to refer to existing elements in the involved metamodels (Cuadrado et al. 2017). Consequently, a type error can be introduced in a transformation program by accident during development (developer or domain expert error) by wrongly using the metamodel types. It can also result from changes in the metamodels it uses, but this case is out of the scope of this paper. Resolving type errors in ATL is thus difficult because of the declarative nature of the transformation language and the dependencies towards the involved metamodels. In the following, we illustrate type errors using the transformation program excerpt of Listing 1.

A common type error concerns properties' types in bindings, such as in line 10. In the Intalio metamodel, the property *artifacts* refers to objects of type `Artifact`. However, in the RHS of this binding, the input object property *name* is of type `String`, causing an *incompatible type* error. Invalid types are also frequent errors. As mentioned earlier, each rule is triggered by an input object that is compatible with the `from` part of the rule. The rule `activitypartition2pool` is thus triggered by objects conforming to `Comment` in the UML metamodel (line 15). If we look closely to the UML metamodel of Fig. 1, there is no `Comment` element: this raises the error *invalid type*. Another common error concerns the binding resolution. Let us consider the binding of line 11. The RHS of the binding calls a helper returning objects of type `ActivityPartition` from UML metamodel. The LHS of the binding is the property *pools*, with type `Pool` from Intalio metamodel. To resolve this binding, there must be a rule in the transformation program having `UML!ActivityPartition` as input pattern, and `Intalio!Pool` as output pattern, which is not the case in our excerpt: this raises a *possible unresolved binding* (Cuadrado et al. 2018) error.

2.2. Challenges of Fixing Model Transformations

Existing research on model transformation repair generally follows the precept that errors sharing "the same symptoms, the same root cause, or the same solution" can be fixed in the same fashion (Martinez et al. 2014). Concretely, for a class of equivalent errors, a predefined patch is applied to all instances of this kind of error. Cuadrado et al. present an evolvable list of patches tailored as a response to every characterized type of syntactic error (Cuadrado et al. 2018). The authors point that the proposed list may evolve with new error types or with the refinement of existing patches. Additionally, one may want to adapt patches to other transformation languages. Defining, refining and adapting patches require an important amount of knowledge, thorough study of their impact and manual maintenance effort. Another issue mentioned by the authors is that the order in which one applies predefined patches may bring unexpected interactions and side effects on the transformation, *e.g* new errors can be injected or contradictory changes may loop. We believe that an approach that dynamically explores candidate patches, rather than applying predefined ones, can circumvent the above-mentioned issues. As patches can be seen as sequences of basic edit operations, such an approach can automatically explore the space of possible sequences that fix several typing errors at once, without creating new ones.

Another important issue is to ensure that the original behavior/semantics of the transformation is not altered – or at least that the semantic discrepancy is circumscribed and characterized. The common way to ensure this behavior preservation after changes is to use an oracle such as test suites, pre-/post-conditions, or possibly other specifications. Yet, in the context of domain-specific problems such as those MDE offers to solve, the necessary knowledge required to build a relevant and trustworthy oracle is not always available (Baudry et al. 2010). Since we are dealing with faulty transformations that cannot be always executable, we cannot rely on test cases to systematically evaluate potentially good candidate patches. Indeed, these may not correct all errors, resulting in partially corrected transformations that potentially cannot be executed. In our approach, we propose to consider, in addition to the objective of correcting the most type errors possible, two other objectives to limit behavior deviation, and thus view the exploration of the space of candidate patches as a multi-objective optimization problem.

2.3. NSGA-II, a Multi-Objective Evolutionary Algorithm

Evolutionary population-based algorithms (EPAs) are automated methods solving optimization problems by iteratively evolving a population of candidate solutions toward a near-optimal one. For multi-objectives optimization problems, EPAs are designed to find a set of optimal solutions, called non-dominated solutions, or Pareto set. A non-dominated solution provides a suitable compromise between all objectives such that one objective cannot be further improved without degrading another objective.

In this paper, we use NSGA-II, a well-known fast multi-objective genetic algorithm, that is suitable to the kind of problem we are solving (Ali et al. 2020). The adaptation of NSGA-II to our problem is described in Section 3. NSGA-II works on

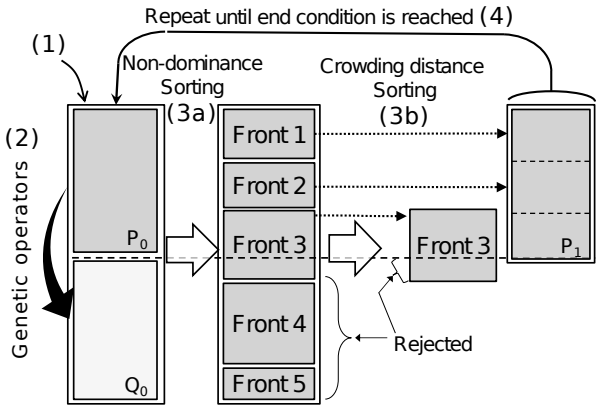


Figure 2 NSGA-II Algorithm (Deb et al. 2000)

a population of N candidate solutions. In the first iteration, a population P_0 of $N/2$ solutions is randomly created (Fig. 2 (1)). The solutions of P_0 are then bred to create a population Q_0 of $N/2$ new solutions (2) by using two genetic operators: a mutation operator slightly changing a candidate solution of P_0 to add it in Q_0, and a crossover operator cutting two candidate solutions of P_0 in two parts and recombining them to obtain two new solutions for Q_0. Q_0 is then merged with P_0 into an initial population of size N. Then, thanks to an objective function, the N solutions are sorted into dominance fronts (3a). A solution s_1 dominates a solution s_2 for a set of objectives $\{O_i\}$ if $\forall i, O_i(s_1) \geqslant O_i(s_2)$ and $\exists j \mid O_j(s_1) > O_j(s_2)$. The first front includes the non-dominated near-optimal solutions. The second front contains the solutions that are dominated only by the solutions of the first front, etc. The fronts are included in the parent population P_1 of the next generation following the dominance order until the size of $N/2$ is reached. If this size coincides with part of a front, the solutions inside this front are sorted, to complete the population, according to a crowding distance which favors diversity in the solutions (Deb et al. 2000) (3b). In this way, P_1 retains the N/2 best solutions of the current iteration. This process is repeated (4) until a stop criterion is reached, e.g., a number of iterations or one or more objectives greater than a certain threshold. At each iteration, the best retained solutions are not the same, and as new solutions are bred based on the best candidates of the previous iteration, they tend towards optimal-solutions. This process does not require user intervention.

3. Multi-step derivation of patches

3.1. Approach Overview

We propose a two-step approach to generate repair patches for transformations containing multiple type errors, as depicted in Fig. 3. The first step, "Exploration phase", takes as input a faulty transformation and the source and target metamodels defining the domain type system, and produces candidate patches. This step has two goals: (1) exploring the space of possible patches with the objective of correcting as much as possible type errors, and (2) minimizing the deviation from the original behavior by

combining two lightweight surrogate objectives to testing.

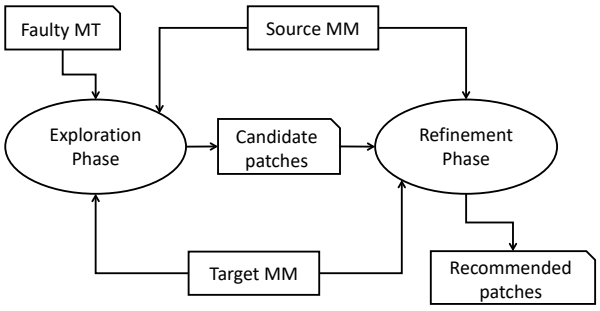


Figure 3 Overview of the proposed two-step approach

As the first step is based on an evolutionary population-based algorithm, the exploration evaluates an important number of solutions. Consequently, we cannot afford to use resource-consuming objectives to limit behavior deviations. Alternatively, after a patch solution is produced by the first step, we refine it in a second step to increase the likelihood that type-error fixes do not alter the behavior. "Refinement phase" exploits four heuristics that better determine the parameters of some operations included in the candidate patches or propose alternative operations, without introducing new type errors in the transformation. The refinement phase thus directly modifies the patches produced during step 1. As step 1 consists in a multiobjective optimization process, it produces a set of solutions rather than a unique one, i.e., the pareto set. Thus, an automated heuristic can select a solution to refine or we can refine all the alternative solutions. After the refinement, the alternative patch sets could be presented to the user for her to select the ones to be applied to fix the faulty transformation. If the set of alternatives is too large, it is possible to use a recommendation system that selects a smaller set of representative solutions as defined in (Batot et al. 2017). Note that the generation of candidate patches and the application of selected ones are entirely automated. The only required user intervention is the selection or approval or alteration of the patches to apply. In the remainder of this section, we describe both steps of this process.

3.2. Exploration Phase

In order to adapt NSGA-II, like any evolutionary population-based algorithm, to our problem, three key points must be defined: solution representation, solution derivation, and fitness evaluation.

Solution representation. As our approach seeks to produce relevant patches, solutions handled by NSGA-II will take the form of sequences of edit operations. We define a sequence of edit operations with a *vector* of *n* positions, where n is the number of edit operations to be applied on the ATL transformation program. We use the basic edit operations listed in Table 1 (Cuadrado et al. 2018) to build our candidate sequences.

The operation *binding creation* adds a new binding in a rule. *Type of source/target pattern element* changes the type of the `from` or `to` part of a rule. *Type of variable collection* changes the type of a collection such as the type `UML!ActivityPartition` of the Sequence in line 4 of the listing. The operation *Type parameter* changes the parameter *Type* of a function such as `oclIsKindOf()`. *Navigation expression* replaces the property in the RHS of a given binding by another property, and *Target of binding* replaces the property of the LHS of a given binding by another property. *Predefined operation call modification*, *Collection operation call modification* and *Iterator call modification* replaces a function call by another one (e.g., `collect()` or `flatten()` from line 5).

Table 1 Set of basic edit operations of model transformations taken from (Cuadrado et al. 2018)

Operator	Target
Creation	Binding
Type modification	Type of source/target pattern element
	Type of variable or collection
	Type parameter (e.g., oclIsKindOf(Type))
Feature name modification	Navigation expression (binding RHS)
	Target of binding (binding LHS)
Operation modification	Predefined operation call (e.g., oclIsKindOf)
	Collection operation call (e.g., includes)
	Iterator call (e.g., exists, collect)
Deletion	Rule, helper, binding ...

As we are dealing with type errors in transformations, it is important to pay a special attention to the delete operators. Indeed, sequences with these operators may artificially resolve some errors by removing faulty fragments of statements, statements or rules containing errors. Behavior-safe guards are essential to prevent the solution to simply delete the faulty parts, but as stated before, we cannot rely on them in our case. Therefore, we ignore delete operators at this stage of our work. For the sake of consistency, in our evaluation in Section 4, we do not consider errors that require delete operations.

Fig. 4 presents an example of a sequence of two edit operations (i.e., a patch) which can be applied on Listing 1 to fix some of the type errors identified in Section 2.1.

In candidate sequences, each edit operation is identified by a name as defined in Table 1. The two operations of Fig.4 have four parameters: *ruletoModify*, *objectToModify*, *oldValue*, and *newValue*. For example, the edit operation `TargetOfBinding` changes the target of the binding (its LHS) from *artifacts* to *documentation* in the rule `activity2diagram` in line 10. The edit operation `TypeOfSourcePatternElement` replaces *Comment* by *Activity* in the input pattern of the rule `activitypartition2pool`. Applying this patch on Listing 1 produces the Listing 2 in which two type errors of different categories have been simultaneously corrected: the *incompatible type* error of line 10 is fixed as properties *documentation* and *name* are both of type `String`, and the *invalid type* error of line 15 is fixed as `Activity` is an existing element of the source metamodel. Note that this patch does not correct all type errors but only some of them. It is likely that it will not be retained as

Operation1: **TargetOfBinding** ("activity2diagram", "d", "artifacts", "documentation")	Operation2: **TypeOfSourcePatternElement** ("activitypartition2pool", "a", "Comment", "Activity")

Figure 4 Example of a sequence of two edit operations (patch) which can be applied on the transformation of Listing 1

a good candidate by the automated approach.

```
1 create OUT : Intalio from IN : UML;
2 ...
3 helper context UML!Activity def: allPartitions
4       :Sequence(UML!ActivityPartition) =
5        self.partition->collect(p |  p.allPartitions)->flatten();
6
7 rule activity2diagram {
8   from    a : UML!Activity
9   to      d : Intalio!BpmnDiagram (
10              documentation <- a.name,
11              pools <- a.allPartitions
12          )}
13
14 rule activitypartition2pool {
15  from    a : UML!Activity
16  to      p : Intalio!Pool,
17          l : Intalio!Lane (
18              activities <- a.node->reject(
19                  e|e.oclIsKindOf(UML!ObjectNode))
20          )}
21 ...
```

Listing 2 Model transformation program of Listing 1 after applying the patch of Fig. 4

A solution is then defined by selecting a sequence of operations and by assigning values to their parameters. The solution space thus spans over all potential combinations of operations, their parameterizations and their order.

As stated in Section 2.3, the first iteration will generate a population of random sequences of edit operations. These candidates are then evaluated using an objective function to retain the best ones, that will be bred in the next iterations to derive new better candidates. Usually, numerous breeding iterations are needed before obtaining near-optimal solutions.

Solution derivation. At each iteration, two operators derive new candidate solutions from existing ones: *crossover* (recombination of the existing genetic material), and *mutation* (injection of new genetic material). A sequence of operations is a convenient representation for breeding through genetic operators. In our adaptation, we use single point crossover operator. This operator consists in cutting the operation sequences of two selected solutions into two parts and in swapping the parts at the right of the cut point to create two new solutions, as illustrated in Fig 5.

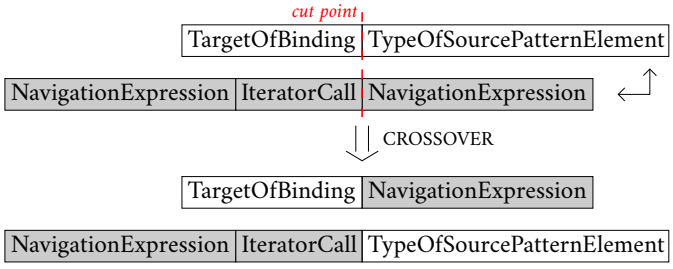


Figure 5 Example of single point crossover operation on the patch of Fig. 4 and another patch of 3 operations

The mutation operator introduces random changes into candidate solutions. In our adaptation, it selects one or more operation(s) from a solution sequence and either replaces them by another type of edit operation or modifies the parameters, as illustrated in Fig. 6

TargetOfBinding ("activity2diagram", "d", "artifacts", "edge")	**NavigationExpression** ("activity2diagram", "d", "a.name", "a.source")

Figure 6 Example of 2 mutation operations applied on the patch of Fig. 4: one mutation on a parameter ("documentation" replaced by "edge") and another one on the operation type ("TypeOfSourcePatternElement" replaced by "NavigationExpression")

Fitness evaluation. For NSGA-II to select the best sequences of edit operations for the next iteration, it needs to rank candidate solutions, which is done by an objective (or fitness) function. A good solution is a sequence of operations which, when applied on a transformation, *(i) fixes the type errors* and *(ii) limits the behavior deviations.* The objective of fixing type errors can be directly evaluated by tools based on transformation language features, such as static fault analysis. This is our first objective:

(1) Fixing type errors *or, to minimize the number of transformation errors.* We used this objective to check the number of errors in the transformation rules after applying the sequence of change operations. To measure the number of errors, we use the AnATLyzer tool (Cuadrado et al. 2018), which finds a wide range of syntactic errors (including type errors) in ATL transformations using static analysis. Formally, the objective function for a solution S is: $Minf1(S) = |Errors(S)|$.

The objective to limit behavior deviations poses a significant challenge and is difficult to capture with a single objective. In this paper, we explore the combination of two additional objectives that we believe could help limit behavior alterations:

(2) To favor solutions of small size *or, to minimize the number of operations.* This objective represents the number of operations in (i.e., the size of) a candidate sequence. We used this objective to reduce the deviation from the initial transformation, and then the risk of changing the semantic. Additionally, we want to prevent the solutions to grow unnecessarily large and escape the bloating effect (de Jong et al. 2001). Formally: $Minf2(S) = |V(S)|$, where V is the solution's sequence of operations.

(3) To keep changes local *or, to minimize the alteration of the metamodels' footprint:* The footprint of the source or target metamodels defined by an (initial or candidate) transformation is estimated by the number of elements from both source and target metamodels that the candidate solution employs (resp. does not employ) whereas the original trans-

formation does not (resp. employs) (Burgueño et al. 2015). Formally, the third objective can be expressed as follows: $Minf3(S) = |SFP(O) - SFP(S)| + |TFP(O) - TFP(S)|$, where SFP and TFP are the footprints in the source and target metamodels, extracted from the original transformation O and the candidate transformation S. To extract the footprint set of a transformation for a metamodel, we use the tool defined by Burgueño et al. (Burgueño et al. 2015).

These objectives are conflicting in essence. For instance, fixing several type errors may necessitate the use of several edit operations, which is conflicting with the objective of minimizing the number of edit operations used in a patch. The difficulty lies in finding a good compromise between these objectives: we solve this multi-objective patch derivation problem by adapting the evolutionary population-based algorithm NSGA-II (Deb et al. 2000) described in Section 2.3. Once the best candidates of the current iteration are selected, they are used as a basis to derive new solutions in the next iteration.

3.3. Refinement Phase

The exploration phase produces a set of candidate patch solutions corresponding to the Pareto front (first front) of the last iteration of NSGA-II. These solutions may remove completely or partially syntactic type errors detected by AnATLyzer, but may also alter the expected behavior of the transformation while doing so. There are many reasons that could explain this phenomenon. For example, the choice of a parameter for a given change operation is made without checking the global consistency with the other change operations in the sequence. Another example is when many type-compatible possibilities exist for a given parameter, one is selected randomly without a proper way to evaluate the likelihood of each possibility to semantically correct the error.

One can sophisticate the decision process of operation and parameter choices in the exploration phase to limit behavior deviation, but this comes at high computation cost considering the number of explored solutions. Thus, we decided to alternatively refine one or more solutions produced by the exploration phase. As the refinement concerns a few solutions, we can afford a more resource-consuming decision process. After analyzing the used change operations, we identified four for which we can define heuristics to improve the decisions made during the exploration phase. In what follows, we present the improvement heuristics for these operations. The goal of these heuristics is to limit behavior deviations that may have been introduced while correcting type errors during the exploration phase. They may also correct some remaining type errors. Even though the heuristics directly modify the patches (i.e., sequences of edit operations), we illustrate their mechanisms on transformation program excerpts, as it is easier to comprehend.

(1) Target of binding. This edit operation changes the LHS of a binding. It may thus produce several bindings having the same target property in a given rule, even though a property should not be initialized more than once.

In these cases, the heuristic seeks to change the LHS of necessary bindings until no property is initialized several times, as illustrated in Fig. 7. First, the heuristic performs an edit distance computation between the target property and the different RHS properties: the binding with the minimum distance is ignored on the next step as it is considered the correct initialization (❶). Then, the heuristic retrieves the list of accessible target properties, and computes the edit distance with each RHS of the remaining bindings (❷). Finally, it modifies the target properties with the closest property of this list (❸).

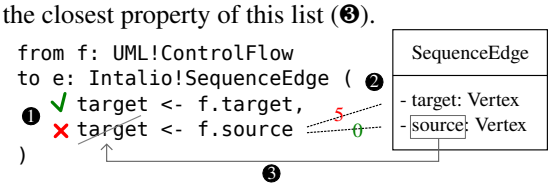


Figure 7 Heuristic for the operation TargetOfBinding

When the binding RHS is of type `String`, changing the LHS to any `String` property prevents typing errors. However, it is common to select an incorrect LHS with regards to semantics. Steps ❷ and ❸ can be applied in this particular case. Applying this heuristic in Listing 2 would change the binding `documentation <- a.name` (line 10) to `name <- a.name`, which is more coherent.

(2) Navigation expression. This operation may change a binding's RHS to have a different type than the binding's LHS (see Fig. 8), causing a *type mismatch* error (❶). When the property in the binding's LHS is of type `String`, the heuristic first retrieves the list of accessible properties which are of `String` type in the input model element (❷). Then, it selects from this list the property's name having the smallest edit distance with the LHS property's name (❸) and replaces the RHS accordingly (❹). This heuristic only applied for String properties.

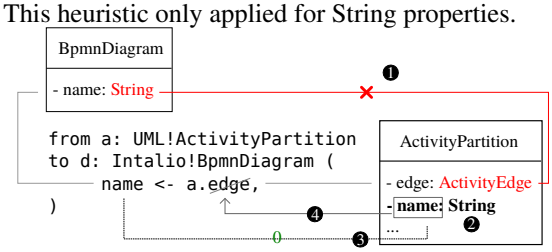


Figure 8 Heuristic for navigationExpression

(3) Type of source (target) pattern element. This edit operation may introduce an improper type in the `from` part of a rule. We have seen previously that each binding which takes into account references towards objects should be resolved (i.e., associated with the correct rule in the transformation). The correct rule has its `from` part corresponding to the type of the RHS of the binding and its `to` part corresponding to the type of the LHS of the binding. We could use the RHS of the binding that refers to that rule to infer the correct type for the `from` part. The third heuristic (see Fig. 9) checks existing bindings to find the one with a LHS whose type is equivalent to the `to` part of a given rule (❶). Then, it verifies if the type of the RHS of this binding corresponds to the `from` part of the rule, and changes the latter accordingly if it is not the case (❷). This heuristic can be applied when there exists only one rule having a given type in its `to` part.

The same verification can be done the other way around for verifying the `to` part of the rule.

(4) Type parameter (e.g., oclIsKindOf(Type)). This edit

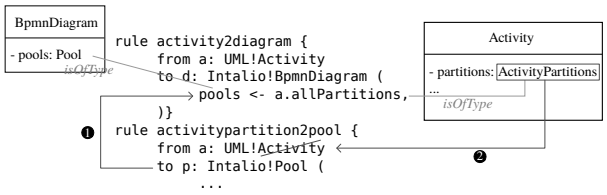


Figure 9 Heuristic for typeOfSourcePatternElement

operation changes the `Type` parameter defined in functions such as `oclIsKindOf` or `oclAsType`. Based on OCL definition, `Type` parameter of `oclIsKindOf`, for example, must inherit from the type defined before (i.e., the inferred type). For instance, in Fig. 10, `UML!NamedElement` should inherit the inferred type of `a.node`, i.e., `ActivityNode`. The fourth heuristic first retrieves the inferred type (❶), then checks whether the `Type` parameter inherits from this type. If not (❷), the heuristic changes the `Type` parameter by a subclass of the inferred type (❸). If `oclIsKindOf` is followed by a property, e.g., `(e|e.oclIsKindOf(UML!NamedElement) -> select(e|e.Language))`, the heuristic randomly chooses a `Type` which has access to that property (here `OpaqueAction`).

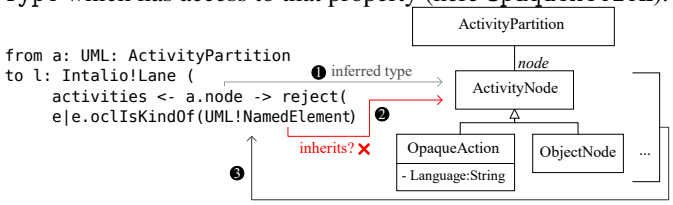


Figure 10 Heuristic for TypeParameter

Listing 2 contains two semantic errors (on lines 10 and 15) that are fixed by heuristics 1 and 3. This produces the transformation in Listing 3, now free of type error and behavior deviation.

```
1 create OUT : Intalio from IN : UML;
2 ...
3 helper context UML!Activity def: allPartitions
4       :Sequence(UML!ActivityPartition) =
5        self.partition->collect(p |  p.allPartitions)->flatten();
6
7 rule activity2diagram {
8   from    a : UML!Activity
9   to      d : Intalio!BpmnDiagram (
10              name <- a.name,
11              pools <- a.allPartitions
12          )}
13
14 rule activitypartition2pool {
15   from    a : UML!ActivityPartition
16   to      p : Intalio!Pool,
17           l : Intalio!Lane (
18               activities <- a.node->reject(
19                   e|e.oclIsKindOf(UML!ObjectNode))
20           )}
21 ...
```

Listing 3 Model transformation program of Listing 2 after applying the heuristics

4. Automatix - Preliminary Tool and Evaluation

We implemented our approach in a tool, called Automatix, and performed an empirical evaluation[2]. The rest of this section describes the investigated research questions, details the evaluation procedure used, presents the results, and discusses the threats to the validity of our evaluation.

4.1. Research Questions

As we explore many solutions during our evolutionary algorithm, it is legitimate to question whether the results are due to our search strategy or to the amount of candidate solutions explored during the search. Thus, we start by performing a sanity check to compare the number of type errors fixed by patches obtained with our approach during the exploration phase and by patches obtained with a random search. Then, we assess whether the patches obtained after the exploration phase limit the alteration of the behavior of the transformations. Note that we do not evaluate the behavior of the output models that can be generated with a corrected transformation, but the behavior of the transformation program itself. Finally, we do the same evaluation, but for patches obtained after the refinement phase. In summary, we formulate the following research questions:

RQ0: Are our results attributable to an efficient exploration of the search space, or are they due to the large number of candidate solutions we explore?

RQ1: Is the exploration phase able to correct type errors in transformations while limiting behavior deviations?

RQ2: Is the refinement phase (combined with the exploration phase) able to correct type errors in transformations while limiting behavior deviations?

4.2. Evaluation Setup

To assess our approach's performance, we followed a rigorous protocol depicted in Fig. 11.

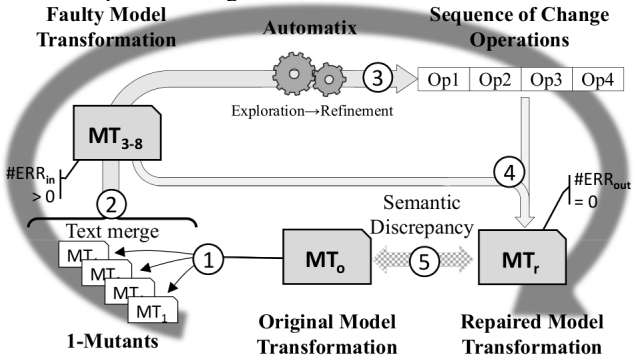


Figure 11 Evaluation procedure overview

Steps 1 and 2 correspond to the creation process of faulty transformations from existing correct model transformations. Then, step 3 is the patch generation with Automatix, and step 4 the application of these patches on the faulty transformations. Finally, we assess the behavior deviation of the repaired transformations in step 5, by performing a semi-automated comparison

[2] For the review process, the experimental data and the code of Automatix can be downloaded using the link https://bitbucket.org/zahravaraminy/ecmfa2021/src/master

against the original model transformations (RQ1 and RQ2). Note that this comparison is not part of our approach, and is only carried out for the evaluation process. The sanity check (RQ0) uses the output of step 2 to perform a random search, and compare the results with the output of step 4.

We applied our evaluation on three existing third-party transformations, *Class2Table*, *PNML2PN* and *UML2BPMN* from the ATL Zoo[3]. *Class2Table* takes as source a class diagram and outputs a relational database schema. *PNML2PN* enables to produce a Petri net from an XML Petri net representation in the PNML format. Finally, *UML2BPMN* transforms a UML Activity Diagram into a business process model (Intalio BPMN) (Schumacher et al. 2013). Table 2 presents information characterizing the 3 transformations and their input/output metamodels.

Table 2 Transformations used in the evaluation. Cells with two values represent input/output metamodels

	Classe2Table	PNML2PN	UML2BPMN
LoC	136	91	118
Rules	8	5	9
Helpers	4	0	6
Classes	6/5	13/9	248/20
Attributes	3/1	4/3	103/13
Associations	11/8	28/20	774/59
Inheritance associations	5/3	14/8	291/28

To limit introducing bias in our evaluation, we used a list of existing faulty transformation mutants provided by the QuickFix project (Cuadrado et al. 2018) (Fig. 11, step 1). Each mutant MT_1 corresponds to the original correct transformation MT_o in which one error of a given class was injected, among the type error categories in ATL transformations (Cuadrado et al. 2018). To create transformations with multiple errors, we selected randomly, for each of the three transformation problems, 6 sets with respectively 3 to 8 mutants coming from distinct error categories. Then, we merged the mutants in each set to form 6 faulty transformations MT_{3-8} with various numbers of type errors (Fig. 11, step 2). Note that we performed the merge sequentially and, then, the number of errors in the resulting transformations can be lower or higher than the number of merged mutants, as some errors may overlap or create new errors as side effects. This allowed us to consider faulty transformations with different numbers of errors, from all error categories, except those requiring delete operations to be fixed, as explained in Section 3.2. As we are using a probabilistic approach, we run this creation process 5 times for each transformation problem. Thus, for a given number of mutants (between 3 and 8), we obtained 5 different faulty transformations, based on different types of mutants. In the end, we obtained for each transformation problem 30 different runs (5 * 6 faulty transformations MT_{3-8}).

For each faulty transformation, we created an initial population of 50 random solutions, *i.e.,* sequences of edit operations. The edit operations are generated for rules with flagged type errors. We complete the initial population by 50 additional solutions obtained by crossover and mutation. We limited the number of generations to 500: for each run, our algorithm thus explores 50,000 possible solutions (500x100). For the other parameters, the crossover and mutation rates are respectively set to 0.8 and 0.2, values that usually perform well (Haupt & Ellen Haupt 2004).

In this evaluation, Automatix takes as input a faulty transformation, and produces candidate patches in the form of sequences of change operations in two phases: one of exploration, and another of refinement (Fig. 11, step (3)). To perform the evaluation, we define the following independent variables w.r.t. to the faulty transformations:

- **#MUT** – Number of mutations applied on the original transformation to derive the faulty one.
- **#ERR**$_{in}$ – Number of type errors found on the transformation after #MUT mutations were applied.

We then define dependent variables w.r.t. the obtained candidate patches:

- **#ERR**$_{out}$ – Number of type errors found on the transformation after a recommended patch has been applied.
- **#OPE** – Size of a recommended patch in number of change operations.
- **#ITE** – Number of algorithm iterations before a recommended patch is found, *i.e.* $\#ERR_{out(patches)} = 0$.
- **SEM** – Rate of errors corrected while preserving the behavior, after the exploration phase (**EP**) and after the refinement phase (**RP**).

We used AnATLyzer to detect the type errors in the input and output transformation programs. This tool also allows us to identify which type errors of the input faulty transformation have been corrected.

Although, the exploration phase may produce more than one solution in the Pareto set, we decided to select only one for the refinement and for the comparison with the random search. To this end, we first select the solution that fixes the highest number of type errors according to AnATLyzer. In the case of a tie, we choose one with the shortest operation sequence. The two criteria were enough to reduce the possibilities to only one solution for all runs.

4.3. Evaluation Results

Table 3 summarizes the results of the different runs of our approach on the mutant configurations described in the setup, for respectively *Class2Table*, *PNML2PN* and *UML2BPMN* transformations (col. 1). #ERR$_{in}$, #ITE and #OPE are average values of the 5 faulty transformations based on the same number of mutants (#MUT). For **#ERR**$_{out}$, we give the min, average and max values for the 5 runs. On average, the majority of errors introduced by the mutants (#ERR$_{in}$ - col. 3), were successfully corrected (#ERR$_{out}$ - col. 6-8 indicating the min, average and max of the number of errors left after applying the patches found by Automatix), according to AnATLyzer. We checked manually that the errors left are those intially introduced and not newly created ones by the patches. For all the cases, we obtained at

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

least one solution without any error left ($min = 0$). Additionally, we did not observe a significant correlation between the number of inputs errors/mutants and the number of generations to find a solution (#ITE - col. 4). We note that we obtained slightly better correction rates for transformations having fewer errors, along with smaller execution times.

Table 3 Results of Automatix. #ERR$_{in}$, #ITE and #OPE represents averages values for the 5 faulty transformations obtained after merging #MUT mutants.

	#MUT	#ERR$_{in}$	#ITE	#OPE	#ERR$_{out}$			SEM	
					min.	avg.	max.	EP	RP
Class2Table	3	3,4	134	3	0	0	0	68%	68%
	4	6	44,8	4	0	0	0	76%	**82%**
	5	7,8	56,8	5	0	0	0	91%	**93%**
	6	8,4	226,8	6	0	0	0	91%	91%
	7	9,6	261,8	5,8	0	0,6	1	71%	**75%**
	8	10,6	276,5	7	0	0,2	1	87%	87%
PNML2PN	3	5,8	86,4	3,2	0	0	0	78%	**89%**
	4	7	137,2	4,2	0	0,4	2	72%	**79%**
	5	8,4	188	5,2	0	0	0	80%	**86%**
	6	9,2	179,2	5,8	0	0,2	1	76%	**87%**
	7	8,4	78,6	6	0	0,8	1	69%	**76%**
	8	9,8	244,6	7,4	0	0,4	1	67%	**78%**
UML2BPMN	3	3,2	162,4	3	0	0	0	45%	**83%**
	4	4,2	35	3.4	0	0,6	1	34%	**41%**
	5	6,8	116	5,2	0	0,2	1	44%	**72%**
	6	6,8	19,2	5,4	0	0,8	1	40%	**53%**
	7	7,8	229,6	5,6	0	0,8	2	25%	**52%**
	8	8	72	5,8	0	1,2	2	26%	**40%**

4.3.1. RQ0 - Sanity Check

To perform the sanity check we limited ourselves to a sample of runs. We considered faulty transformations with 2, 4, 6 and 8 mutants for the problem of *Class2Table*. We compare the results obtained by Automatix to those of a random search for the considered transformations. Since Automatix explores 50,000 solutions for each run, the random exploration also picks, for each run, the best individual from 50,000 solutions generated randomly, such as the initial population in Automatix. As for Automatix, the random exploration was also performed 5 times for each faulty transformation.

As shown in Table 4, the solutions obtained with Automatix correct on average clearly more errors than random ones. Except for transformations with two mutants (first line), the difference between the two strategies is statistically significant (T-Test with a *p-value* lesser than 0.001), and with a high effect size (Cohen's d greater than 3)[4].

Table 4 Automatix vs random results for *Class2Table*.

	Average #ERR$_{out}$		Mann Witney	Effect Size
#MUT	*Automatix*	*RDN*	p-value	Cohen's d
2	0.0	0.2	0.374	-
4	0.0	2.8	<0.001	10.6
6	0.4	5.8	<0.001	8.6
8	3.0	6.4	<0.001	3.24

4.3.2. RQ1 - Error Correction after Exploration Phase (EP)

We consider that an error is actually corrected in a transformation program when the change brought by the patch matches the corresponding code fragment in the original correct transformation MT_o. To assess that (Fig. 11, step 5), we followed a two-step process. We started by applying an automated text diff between the original transformation MT_o (the ground truth) and the transformation MT_r fixed by a patch obtained after the exploration phase. Then, we manually checked the discrepancies flagged by the diff to determine the number of errors that were corrected without altering the behavior of the transformation (call them semantically fixed) and reported the rate of these errors with respect to #ERR$_{in}$ in columns SEM(EP). In this way, we are sure to determine if the applied patches obtained automatically are actually correcting type errors and not just making AnATLyzer not detecting them.

As shown in column SEM(EP) of Table 3, the actual correction rates were good for two transformation problems. Indeed, we succeeded to correct on average between 68% and 91% of errors for *Class2Table*, and between 67% and 80% for *PNML2PN*. For the third transformation problem *UML2BPMN*, the results were less good with an average actual correction rate between 25% and 45%, although some executions reached higher scores. After this phase, we noted that most of the residual type errors still detected by AnATLyzer are *Operation not found* and *Feature not found*, suggesting that they are the most difficult to fix. Among the type errors that have been corrected but that do not preserve the behavior of the transformations, we observed notably *Invalid type*, *Feature not found* and *Compulsory feature not initialized*. These classes of errors require substituting or adding one of the many features present in the metamodels with compatible types: this obviously increases the risk of choosing a wrong feature. When analyzing the semantic discrepancies for *UML2BPMN*, we noticed that, in addition to the complexity of the involved metamodels, these make an intensive use of inheritance. In fact, in a large hierarchy, there are potentially many types that can access the same set of attributes/references: having many options to fix a type error by substituting a type by another one increases the possibility to select the wrong type, and thus introducing a behavior deviation. Fixing errors like *Invalid type* and *Feature not found* with correct solutions is thus

[4] According to Sawilowsky (Sawilowsky 2009), an effect size greater than 2 is considered as huge

even more difficult in this case.

4.3.3. RQ2 - Error Correction after Refinement Phase (RP)

To answer RQ2, we perform the same semantic discrepancy but this time on patches obtained after executing the refinement phase on the candidate patches generated by the exploration phase. As shown in column SEM(RP) of Table 3, the results indicate an improvement of the correction rates with behavior preservation in all three transformation problems (increased rates are shown in boldface in the table). For *Class2Table*, the correction rate increased on average from 80.7% to 82.7%. Over the 6 faulty transformations, 3 witnessed a higher rate (transformation with 4, 5, and 7 mutants). The improvement was more important for *PNML2PN*, for which the average correction rate jumped from 73.7% to 82.5%. In this case, all the faulty transformations saw their correction improve. Finally, for *UML2BPMN*, we observed sizeable improvements of correction rates from 35.7% on average to 56.8%. Here again, the improvement concerned all the faulty transformations reaching a rate of 83% for the ones with 3 mutants.

Similarly to the exploration phase, we observed that most of remaining type errors still detected by AnATLyzer are of classes *Operation not found* and *Feature not found*. Among the errors whose correction introduce behavior deviations, we noted *Compulsory feature not initialized*, and *Feature not found* for the *UML2BPMN* transformation, suggesting that this second phase works particularly well with *Invalid type* errors. In the rates shown, we do not include errors that were partially corrected thanks to the heuristics. For example, we observed that for some bindings, the RHS was actually corrected but not the LHS. This means that the impact of the refinement phase can be much higher than one indicated by the correction rates.

In conclusion, we show that the proposed approach is able to correct multiple type errors at the same time. The evaluation reveals that the two behavior-oriented objectives of the first phase circumscribe the risk of behavior alteration. It also shows that the combination of exploration and refinement phases allows to generate patches that correct most of the type errors while preserving the behavior. These results are evidence that deepening the analysis of edit operations and the possible behavior deviations they may introduce help guiding the search through new objectives or refinement heuristics.

From another perspective, when running this experimentation, we found that a limitation of this approach is the execution time, as obtaining patches takes hours. In fact, the high execution time is mainly due to the objective to keep the changes local. For this objective, we retrieve metamodels' footprints using an external tool which is not optimized for the kind of problem we solve. For the rest, including the 2 other objectives, the execution time takes a few seconds. We plan to develop our own implementation of the footprint retrieval to optimize the execution.

4.4. Threats to Validity

There are some threats that may call into question the validity of our evaluation results. First, the faulty transformations used in the evaluation contain mutants and not errors actually introduced by developers. We used this external data set because it is independent from our project and was used to evaluate the state-of-the-art work. Moreover, it covers a large spectrum of error types. Finally, the random combination of basic mutants we used can be representative of the randomness with which errors can be introduced by developers.

Another limitation of our work at this stage of our project, is that we do not consider some of the error types (mutations). In addition to errors that require delete operations mentioned earlier in the paper, we do not handle errors on ATL transformation helpers. We expect to extend our work in the near future to also consider both families of errors. In the same vein, we tested our approach only with the Atlas Transformation Language (ATL) as a model transformation language. We do believe that the presented approach can be generalizable and adapted to consider other transformation-dedicated languages. This can be done by considering specific versions of edit operations corresponding to the targeted language.

Our approach does not produce a single solution, but a Pareto set of solutions. For the sake of automated evaluation, we selected from the Pareto set the solution with the minimum number of errors left. In the case of a tie, we choose the solution with minimum #OPE. We did the same for RQ0 with the random exploration. In a real setting, other solutions, discarded for their larger size, can be presented to the user, as well as alternative solutions. This can be done by using a diversity strategy to propose a representative sample of solutions (Batot et al. 2017).

We performed a manual inspection of the solutions to evaluate the semantic discrepancy between fixed transformations and original ones. In future evaluations, we plan to use a test suite of pairs of input-output models to assess the behavior preservation. Of course, this is possible only for solutions with no type errors.

5. Related Work

The work presented in this paper crosscuts two research areas: program repair in general and verification and validation of model transformations. In the following subsections, we discuss representative work of both areas.

Program Repair: There is a plethora of works that try to automatically fix bugs in programs using different approaches such as genetic programming (Le Goues et al. 2013), machine learning (Jeffrey et al. 2009; Martinez & Monperrus 2013) or SMT solvers (Demarco et al. 2014). Most of the existing work targets a specific type of errors such as buggy IF conditions, memory allocation errors and infinite loops (Le Goues et al. 2012; Logozzo & Ball 2012; Muntean et al. 2015; Perkins et al. 2009; Demarco et al. 2014). To evaluate the patches, most of the approaches use test suites as oracle. However, other oracles such as specifications (pre-/post-conditions) were also explored (Pei et al. 2014). Although these approaches produced promising results, they cannot be used to fix transformation typing errors. As mentioned in Section 2.2, test suites are difficult to use and specifications are often not available. Moreover, we aim at correcting simultaneously a variety of errors.

Model Transformation verification and validation: In

this research area, there are three families of work: transformation testing, verification and validation of transformations, and transformation repair.

In the first family, Gogolla et al. (Gogolla & Vallecillo 2011), for example, presented a model transformation testing approach based on the concept of Tract, which defines a set of constraints on the source and target metamodels, a set of source-target constraints, and a tract test suite, i.e., a collection of source models satisfying the source constraints. Then, they automatically generated input models and transformed the source model into the target model. Finally, they verified that the source/target models satisfy those constraints. There are other approaches to test the model transformations using other techniques such as graph patterns (Balogh et al. 2010), model fragments (Mottu et al. 2008), Triple Graph Grammars (TGGs) (Wieber et al. 2014) or a combination of these approaches (Giner & Pelechano 2009).

In the second family, for example, Troya et al. (Troya et al. 2018) presented the Spectrum-Based Fault Localization technique and used the results of test cases to determine the probability of each rule of transformation to be faulty. Similarly Burgueño et al. (Burgueño et al. 2015) presented a static approach for detecting the faulty rules in model transformations. Their approach uses matching functions that automatically create the alignment between specifications and implementations. Oakes et al. (Oakes et al. 2018) presented one method to fully verify pre-/post-condition contracts on declarative portion of ATL model transformations. Their approach transforms the declarative portion of ATL transformations into DSLTrans and uses a symbolic-execution to produce a set of path conditions, which represent all possible executions to the transformation. To verify the transformation, they verify pre-/post-condition contracts on these path conditions. Finally, Cuadrado et al. (Cuadrado et al. 2014) presented a combining approach involving a static analysis and constraint solving to detect errors in model transformations. They detected potentially problematic statements and then used a witness model to confirm the erroneous statements. We also used this technique for calculating the number of errors, which is defined as a fitness function.

All the above-mentioned approaches allow to find behavior errors and/or localize the faulty rules/statements. However, they do not propose patches to repair the errors, which is the goal of our approach. Yet, they can be used upstream of our approach like we did with AnATLyzer.

For work that fixes transformation errors, we distinguish between errors generated by the evolution of metamodels as addressed by Kessentini et al. (Kessentini et al. 2018) and errors introduced by the developers. For these errors, to the best of our knowledge, the only existing work is Quick fix (Cuadrado et al. 2018), which allows the correction of detected errors in ATL model transformation. In this approach, they used the static analyser presented in (Cuadrado et al. 2014) to identify errors in ATL model transformations. Then, they extended the analyser to generate a catalogue of quick fixes, which depends on static analysis and constraint solving, for identified errors in ATL transformations. Then, quick fixes propose changes in the transformation based on the kind of error. The user selects a suitable fix among the proposed ones and applies interactively. The differences with our work are that we aim at fixing errors jointly without predefined patch patterns, and also we target to generate a patch without requiring human assistance, except for selecting, among the final candidate patches, the one to be applied.

Model transformations are not the only MDE artefacts targeted by repair approaches. There are many research contributions to generate patches for various modeling artifacts. Models are those that gather much attention as evidenced by the study of Macedo et al. (Macedo et al. 2017). Another example of MDE artifact repair is given by Hegedus et al. (Hegedus et al. 2011) in which the authors used state-space exploration techniques to generate quick fixes for Domain-Specific Modelling Languages (DSMLs).

6. Conclusion and future work

In this paper, we explored the idea of fixing type errors in model transformation programs without relying on predefined patches. Considering that a patch is a sequence of basic edit operations, our approach explores the space of candidate sequences guided by two families of objectives: correcting type errors and limiting behavior deviations. While the correction of type errors is relatively easy to measure using transformation language features, behavior preservation poses many challenges. To tackle these issues, we proposed a two-phase approach to find candidate sequences that limit behavior deviations. The first phase combined two objectives during the exploration to prevent behavior deviations: minimizing the size of the sequence and keeping the changes local. During a second phase, we applied four heuristics on the obtained patches to improve the decisions made during the exploration phase. An evaluation of our idea showed that the first phase corrected a majority of type errors for two transformation problems, *Class2Table* and *PNML2PN*, most of the time without altering the behavior. We also showed that refining the patches obtained after the exploration using the four proposed heuristics significantly improved the quality of the patches in terms of limiting behavior deviations for the three transformation problems, including *UML2BPMN*.

As a future work, we plan to further investigate alternative objectives to limit behavior alterations to achieve correct and complete patches. We also envision to inject some heuristics when selecting edit operations (initial population generation and mutations) to decrease the probability of altering the behavior. Finally, we seek to widen our approach to repair semantic errors, i.e., incorrect behaviors of transformation programs, leading them to produce output models that are not the expected ones. Semantics errors may be caused by binding the wrong attribute, or using the wrong helper, without necessarily inducing a type error. Until now, we worked with ATL transformations having type errors and that could not always be executed, which hindered the possibility to use a fully automated process to assess the transformation behavior alteration and work precisely on semantic errors. Our work is now at a stage where we can obtain ATL transformations with no type error: in the future, we thus plan to use more rigorous alternatives to assess behavior alterations, such as pairs of correct input/output model examples.

Using such behavior-safe guards will also allow us to safely consider the delete operators in the patch derivation process.

Acknowledgments

This work was partially funded by NSERC (grant RGPIN-2019-07168), IVADO, and the Canada First Research Excellence Fund / Apogée.

References

Ali, S., Arcaini, P., & Yue, T. (2020). Do quality indicators prefer particular multi-objective search algorithms in search-based software engineering? In *12th Int. Symp. on Search Based Software Engineering* (pp. 25–41).

Baki, I., & Sahraoui, H. (2016). Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Transactions on Software Engineering and Methodology*, *25*(3), 1–37.

Balogh, A., Bergmann, G., Csertán, G., Gönczy, L., Horváth, Á., Majzik, I., ... others (2010). Workflow-driven tool integration using model transformations. In *Graph Transformations and Model-Driven Engineering* (pp. 224–248). Springer.

Batot, E., Kessentini, W., Sahraoui, H., & Famelis, M. (2017). Heuristic-based recommendation for metamodel - OCL co-evolution. In *20th Int. Conf. on Model Driven Engineering Languages and Systems* (pp. 210–220).

Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., & Mottu, J.-M. (2010). Barriers to systematic model transformation testing. *Communications of the ACM*, *53*(6), 139–143.

Burgueño, L., Troya, J., Wimmer, M., & Vallecillo, A. (2015). Static fault localization in model transformations. *Transactions on Software Engineering*, *41*(5), 490-506.

Combemale, B., France, R., Jézéquel, J.-M., Rumpe, B., Steel, J., & Vojtisek, D. (2016). *Engineering modeling languages: Turning domain knowledge into tools*. Chapman and Hall/CRC.

Cuadrado, J. S., Guerra, E., & de Lara, J. (2017). Static analysis of model transformations. *IEEE Transactions on Software Engineering.*, *43*(9), 868–897.

Cuadrado, J. S., Guerra, E., & de Lara, J. (2018). AnATLyzer: An Advanced IDE for ATL Model Transformations. In *40th Int. Conference on Software Engineering: Companion (ICSE-Companion)* (pp. 85–88).

Cuadrado, J. S., Guerra, E., & de Lara, J. (2018). Quick fixing atl transformations with speculative analysis. *Software and Systems Modeling*, *17*(3), 779–813.

Cuadrado, J. S., Guerra, E., & d. Lara, J. (2014). Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In *25th Int. Symposium on Software Reliability Engineering* (pp. 34–44).

Deb, K., Agrawal, S., Pratap, A., & Meyarivan, T. (2000). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In *Int. Conf. on Parallel Problem Solving from Nature*.

de Jong, E. D., Watson, R. A., & Pollack, J. B. (2001). Reducing bloat and promoting diversity using multi-objective methods. In *Proc. of the Conf. on Genetic and Evolutionary Computation* (pp. 11–18).

Demarco, F., Xuan, J., Le Berre, D., & Monperrus, M. (2014). Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis* (pp. 30–39).

Giner, P., & Pelechano, V. (2009). Test-driven development of model transformations. In *Int. Conf. on Model Driven Engineering Languages and Systems* (pp. 748–752).

Gogolla, M., & Vallecillo, A. (2011). Tractable model transformation testing. In *European Conference on Modelling Foundations and Applications* (pp. 221–235).

Haupt, R. L., & Ellen Haupt, S. (2004). *Practical genetic algorithms*. Wiley Online Library.

Hegedus, A., Horvath, A., Rath, I., Branco, M. C., & Varro, D. (2011). Quick fix generation for DSMLs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (pp. 17–24).

Jeffrey, D., Feng, M., Neelam Gupta, & Gupta, R. (2009). Bugfix: A learning-based tool to assist developers in fixing bugs. In *2009 IEEE 17th International Conference on Program Comprehension* (pp. 70–79).

Jouault, F., Allilaire, F., Bézivin, J., & Kurtev, I. (2008). Atl: A model transformation tool. *Science of computer programming*, *72*(1-2), 31–39.

Kessentini, W., Sahraoui, H., & Wimmer, M. (2018). Automated co-evolution of metamodels and transformation rules: A search-based approach. In *Search-Based Soft. Eng.* (pp. 229–245). Springer.

Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012). Genprog: A generic method for automatic software repair. *Transactions on Soft. Eng.*, *38*(1), 54-72.

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

Logozzo, F., & Ball, T. (2012). Modular and verified automatic program repair. In *the 27th ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM SIGPLAN.

Macedo, N., Jorge, T., & Cunha, A. (2017). A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering*, *43*(7), 615-640.

Martinez, M., & Monperrus, M. (2013). Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Soft. Eng.*, *20*.

Martinez, M., Weimer, W., & Monperrus, M. (2014). Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion proc. of the 36th int. conf. on soft. eng.* (pp. 492–495).

Monperrus, M. (2018). Automatic software repair: A bibliography. *ACM Comput. Surv.*, *51*(1), 1–24.

Mottu, J. M., Baudry, B., & Le Traon, Y. (2008). Model transformation testing: oracle issue. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop* (pp. 105–112).

Muntean, P., Kommanapalli, V., Ibing, A., & Eckert, C. (2015). Automated generation of buffer overflow quick fixes using

symbolic execution and SMT. In *Computer Safety, Reliability, and Security* (pp. 441–456).
Oakes, J. B., Troya, J., Lúcio, L., & Wimmer, M. (2018). Full contract verification for ATL using symbolic execution. *Softw. Syst. Model.*, *17*(3), 815–849.
Pei, Y., Furia, C. A., Nordio, M., Wei, Y., Meyer, B., & Zeller, A. (2014). Automated fixing of programs with contracts. *Transactions on Soft. Eng.*, *40*(5), 427-449.
Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S. P., Bachrach, J., Carbin, M., … Rinard, M. C. (2009). Automatically patching errors in deployed software. In *Symp. on operating systems principles* (p. 87-102).
Sawilowsky, S. S. (2009). New effect size rules of thumb. *Journ. of Modern Applied Statistical Methods*, *8*(2), 26.
Schumacher, F., Schröck, S., & Fay, A. (2013). Transforming hierarchical concepts of grafcet into a suitable petri net formalism. *IFAC Proceedings Volumes*, 295–300.
Sendall, S., & Kozaczynski, W. (2003). Model transformation: the heart and soul of model-driven software development. *IEEE Software*, *20*(5), 42-45.
Staats, M., Whalen, M. W., & Heimdahl, M. P. E. (2011). Programs, tests, and oracles: the foundations of testing revisited. In *33rd Int. Conf. on Soft. Eng.* (pp. 391–400).
Troya, J., Segura, S., Parejo, J. A., & Ruiz-Cortés, A. (2018). Spectrum-based fault localization in model transformations. *ACM Trans. Softw. Eng. Methodol.*, *27*.
Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE Software*, *31*, 79-85.
Wieber, M., Anjorin, A., & Schürr, A. (2014). On the usage of TGGs for automated model transformation testing. In *International Conference on Theory and Practice of Model Transformations* (pp. 1–16).

About the authors

Zahra VaraminyBahnemiry is a PhD student at the department of Computer Science and Operations Research (GEODES Group) of the Université de Montréal (Canada) You can contact the author at varaminz@iro.umontreal.ca.

Jessie Galasso is a post-doc at the department of Computer Science and Operations Research (GEODES Group) of the Université de Montréal (Canada). You can contact the author at jessie.galasso-carbonnel@umontreal.ca.

Houari Sahraoui is a professor at the department of Computer Science and Operations Research (GEODES Group) of the Université de Montréal (Canada). You can contact the author at sahraouh@iro.umontreal.ca .