Improving Repair of Semantic ATL Errors using a Social Diversity Metric

Journal:	*Software and Systems Modeling*
Manuscript ID	SOSYM-23-00004493
Manuscript Type:	Regular Paper
Keyword:	model-driven engineering, model transformations, ATL, evolutionary algorithms, social diversity

Improving Repair of Semantic ATL Errors using a Social Diversity Metric

Zahra VaraminyBahnemiry, Jessie Galasso, Bentley Oakes, Houari Sahraoui

Département d'Informatique et de Recherche Opérationnelle (DIRO), Université de Montréal, 3150 Jean Brillant St, Montréal, H3T 1N8, Quebec, Canada.

*Corresponding author(s). E-mail(s): sahraouh@iro.umontreal.ca;
Contributing authors: varaminz@iro.umontreal.ca; jessie.galasso-carbonnel@umontreal.ca;
bentley.oakes@umontreal.ca;

Abstract

Model transformations play an essential role in the Model-Driven Engineering paradigm. However, writing a correct transformation requires the user to understand both *what* the transformation should do, and *how* to enact that change in the transformation. This easily leads to *syntactic* and *semantic* errors in transformations which are time-consuming to locate and fix. In this article, we extend our evolutionary algorithm (EA) approach to automatically repair transformations containing *many semantic errors*. To prevent the *fitness plateaus* and the *single fitness peak* limitations from our previous work, we include the notion of *social diversity* as an objective for our EA to promote repair patches tackling errors that are less covered by the other patches of the population. We evaluate our approach on four ATL transformations, which have been mutated to contain up to five semantic errors simultaneously. Our evaluation shows that integrating social diversity when searching for repair patches improves the quality of those patches and speeds up the convergence even when up to five semantic errors are involved.

Keywords: model-driven engineering, model transformations, ATL, evolutionary algorithms, social diversity

1 Introduction

Model-driven engineering (MDE) is an efficient approach to reduce the complexity of software development complexity by increasing the level of abstraction [24]. In this context, MDE sees models as first-class artifacts where domain-specific modeling languages to capture specific aspects of the solution.

Model transformations then play an an essential role in MDE, as they specify how to transform elements of a model conforming to a source meta-model into elements of a model conforming to a target meta-model. by producing from these models low-level artifacts such as source code, documentation, and test suites [34], or by optimizing or simulating the model itself [22].

Model transformations can be written with general-purpose programming languages or in dedicated transformation languages such as DSLTrans [3] or the ATLAS Transformation Language (ATL) [17].

Errors in Model Transformations

Writing a correct model transformation requires the developer to be proficient with the source and

target meta-models, to have a clear understanding of the mapping between the elements of the two and to know how to exploit the transformation mechanisms of the language to properly describe this transformation. Transformations are thus complex and error-prone, and finding and fixing errors in them typically involve a tedious and time-consuming effort by developers.

Several types of errors can affect a transformation. *Syntactic errors* usually prevent the transformation from compiling and producing an output model. To alleviate the developers' effort when fixing syntactic errors, works such as the one of Cuadrado *et al.* [9] propose predefined corrective patches to be applied on errors detected with syntactic analysis tools (e.g., AnATLyzer [8] for the ATL language).

In contrast, when the transformation compiles but the implemented behavior is not the one that was intended by the developers, we say that it contains *semantic errors*. As a consequence, semantically incorrect transformations can produce output models, but these models are different from the ones that the user expects. Because semantic errors pertain to the transformation's behavior and each faulty transformation needs tailored patches, predefined patches are not well-suited for semantic errors.

Correcting Errors with Evolutionary Algorithms

Population-based evolutionary algorithms (EAs) have been widely used to correct errors in programs [25], including both syntactic [42] and semantic [41] errors in transformations. Formulating transformation repair as an optimization problem enables such search-based approaches to find patches that will fix a given faulty transformation in the space of all possible patches. EAs maintain a population of candidate patches which undergo a process of evolution across several generations until an optimal patch is found. At each generation, the evolution process creates new solutions based on the population of the previous generation, and the best candidates are retained for the next, hopefully better, generation.

Finding suitable patches with this approach is a fully automated process, at the end of which, the best fitting patches can be presented to the expert to make a final decision about the repair to be applied. To fix errors related to a transformation's behavior, automated approaches usually rely on a specification of the expected behavior (e.g., test cases or examples) to assess the fitness of a patch, and thus efficiently guide the search strategy.

In our previous work [41], we used EAs with test cases to correct semantic errors in ATL transformations. This approach usually finds patches to correct transformations having fewer errors, but in the presence of more errors, the approach cannot find a solution or will take too long to converge toward suitable patches. Preliminary analysis showed that using test cases to assess the fitness of the corrective patches makes the search space difficult to explore efficiently due to *fitness plateaus* [36], an issue of EAs which impedes the ability of the approach to converge toward optimal patches. In addition, EAs are known to give more power to good solutions, which can cause converging issues due to loss of diversity, a problem known as *single fitness peak*. Using behavior specifications such as test cases to guide the search in EAs can exacerbate these limitations [4, 36].

Contributions and Structure

In this paper, we extend our EA-based approach from [41] to automatically find patches to correct transformation with a greater number of semantic errors. In particular, to improve the efficiency and effectiveness of EAs using test cases, our improved approach leverages the notion of *social diversity* [4]. This metric promotes patches which tackle errors that are less covered by the other patches of the population. Our hypothesis is that including this measure in the process will maintain or improve the diversity of the patches, thereby reducing the negative impact on convergence of single fitness peak and fitness plateaus. To include this notion in EAs, we formulate the transformation repair as a *multi-objective optimization problem*, where solutions must optimize several objectives including social diversity. Our approach is implemented using the NSGA-II algorithm, a fast multi-objective EA [10].

We perform an evaluation on four ATL transformations which have been mutated, assessing the impact of social diversity on the convergence of EA-based repair. We reuse the two faulty transformations from our previous work and also consider two new transformations taken from the ATL

zoo[1] to thoroughly evaluate the impact of our approach on transformations having many errors. The evaluation shows that social diversity is able to improve both the efficiency and the efficacy of EAs to fix faulty transformations, even when they contain up to five semantic errors.

In Section 2, we briefly describe ATL transformations. We provide examples of defects and those patches repairing such defects. Section 3 presents the use of EAs for repairing semantic errors in transformations. Section 4 describes our EA multi-objective approach using social diversity to automatically generate patches which attempt to overcome these limitations. Our approach is evaluated in Section 5 to determine the impact of introducing social diversity in EAs on improving convergence and repairing a greater number of errors. The benefits, limitations, and threats to our approach are discussed in Section 6. Section 7 presents related work and Section 8 concludes the paper.

2 Background

In this section, we first provide background about model-to-model transformations. We focus on transformations written in the well-known ATLAS Transformation Language (ATL) [17], but the approach presented in this paper is generic and can be adapted to other transformation languages. We then present the types of errors that can be found in such transformations, including semantic errors, which are the target of this work. We demonstrate patches to repair faulty transformations and discuss why their generation is challenging. Finally, we present a formalization of the problem.

2.1 ATL Transformations

Model transformations are an approach for specifying and automating the process of transforming a source model into a target model. A transformation relies on meta-models describing both the source and the target models: these meta-models can be the same (*endogeneous transformations*) or they can be different (*exogeneous transformations*). Thus, a given transformation is defined for a pair of meta-models, and can only transform source models conforming to the input meta-model into a target model conforming to the output meta-model.

ATL [17] is a well-known textual transformation language which is studied in the MDE literature [8, 9, 16, 29]. Examples of ATL transformations can be found in the ATL zoo, a repository which also includes the necessary meta-models along with documentation. We present ATL by examining an example inspired from the *Class2Relational*[2] transformation of the ATL zoo. The *Class2Relational* transformation transforms an UML class diagram into its equivalent relational schema. Figure 1 shows simplified versions of the UML Class Diagram meta-model and the Relational Schema meta-model.

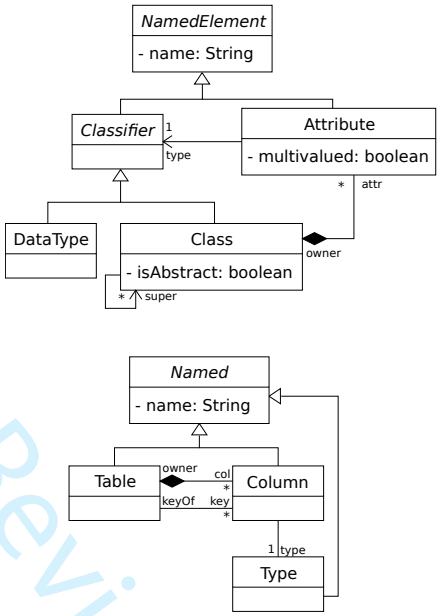


Fig. 1 UML Class Diagram meta-model (top) and Relational Schema meta-model (bottom)

Listing 1 presents a simplified excerpt of the *Class2Relational* ATL transformation. This transformation is the correct version in which erroneous code (which we will discuss in the next section) appears in the commented lines (lines 7, 8, and 32, starting with '−−'). The two meta-models are identified on line 1: the source meta-model is *Class* (`IN`) and the target meta-model is *Relational*

[1] https://www.eclipse.org/atl/atlTransformations/

[2] https://www.eclipse.org/atl/atlTransformations/#Class2Relational

(`OUT`). Rules are introduced by the keyword `rule` (lines 3, 15, and 22) and have a name (for instance, `Class2Table` in line 3). Each rule has two parts: a `from` part defines a pattern of elements from the source meta-model, and a `to` part defining how to transform elements of the `from` part into elements of the target meta-model. Patterns can represent types: for instance on line 4, the rule *Class2Table* applies to each element conforming to the type *Class*. They can also be refined with constraints, here defined with the OCL language: for instance, lines 16-18 states that the rule *SingleValuedDataTypeAttribute2Column* applies on elements conforming to the type *Attribute*, having an attribute *type* representing a native type, and an attribute *multiValued* being false.

The `to` part describes how to create elements of the target model based on the elements of the source model matching the associated `from` part. The `to` part may create one element (e.g., in lines 19-20, a Column is created when an Attribute is matched) or several ones (e.g., in line 5 and line 12, both a Table and a Column are created when a Class is matched). For each created element, one can define *bindings* to associate values to the attributes of the created element. Bindings can use values of the source model elements matched in the `from` part to initialize the target model elements. For instance, in line 6, the attribute *name* of Table is initialized using the name of the matched *Class* (i.e., *c.name*). Bindings can define collections (e.g., a Sequence in line 9, a Set in line 11 and may use iterator or operation calls in initialization (e.g., *firstToLower()* in line 30).

Listing 1 Excerpt of a repaired ATL transformation from Class Diagram to Relational Schema. Commented lines present an Defects fixed by the patch in Figure 4 are commented out.

```
1  create OUT : Relational from IN : Class;
2
3  rule Class2Table {
4  from c: Class!Class
5  to   out: Relational!Table (
6          name <- c.name,
7          -- col <- Sequence {key} -> excluding(
8             --c.attr->collect(e | not e.
                multiValued)),
9          col <- Sequence {key} -> union(
10            c.attr->select(e | not e.multiValued)
              ),
11         key <- Set {key}),
12       key: Relational!Column (
13          name <- c.name + 'Id')}
14
15 rule SingleValuedDataTypeAttribute2Column {
16 from a: Class!Attribute (
17         a.type.oclIsKindOf(Class!DataType)
18       and not a.multiValued)
19 to   out: Relational!Column (
20         name <- a.name)}
21
22 rule MultiValuedClassAttribute2Column {
23 from a: Class!Attribute (
24         a.type.oclIsKindOf(Class!Class)
25       and a.multiValued)
26 to   out: Relational!Table (
27         name <- a.owner.name + '_' + a.name,
28         col <- Sequence {id, foreignKey}),
29     foreignKey1: Relational!Column (
30         name <- a.owner.name.firstToLower() + '
                Id'),
31     foreignKey2: Relational!Column (
32         --name <- a.type + 'Id')}
33         name <- a.name + 'Id')}
```

Figure 2 shows an example of the target model (right-hand side) conforming to the Relational meta-model obtained when running the transformation of Listing 1 on a source model (left-hand side) conforming to the class diagram meta-model.

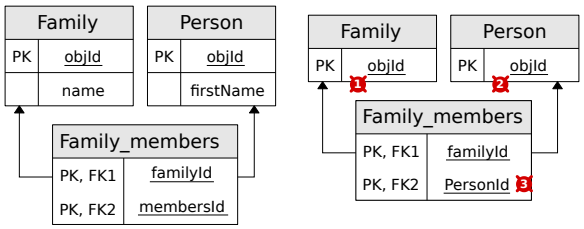
Fig. 2 Relational Schema (output model, right-hand side) obtained when applying the transformation of Listing 1 to the class diagram (input model, left-hand side)

2.2 Defects in Transformations

Transformations thus highly depend on the elements of the two meta-models. *Syntactic errors* can be due to type misuse such as referring to elements that are not in the meta-models or setting properties with values of the wrong type. Syntactic errors usually hinder the proper compilation and execution of the transformation. Tools such as AnATLyser [8], a static analyzer for the ATL language, can be used to check syntactic errors.

Semantic errors make a transformation behave in a way that differs from what is expected, i.e., the transformation is semantically incorrect with respect to a specification of the expected behavior. These errors do not necessarily hinder the compilation and execution processes, but may cause the transformation to produce the wrong outputs. A straightforward way to outline the intended behavior of a transformation is to provide a set of inputs-outputs examples, i.e., test cases defining input models and their corresponding expected output models. When provided with the test case input models, the transformation will produce some output models. The input models can then be checked against the test case output models to detect behavior deviations. In other words, if the outputted models are different from those of the provided examples, it shows that the transformation is semantically incorrect with regards to the provided test cases. Figure 2 thus represents a test case for the transformation of Listing 1 with an input model (left-hand side) and the expected output model (right-hand side).

Let us consider the version of the transformation of Listing 1 using the commented code. Figure 3 (b) presents the target model we obtain when applying this version of the transformation on the input model of Fig. 2. We can see that it is different from the expected target model (a) in three different locations, as highlighted but red dots in Fig. 3 (b). The columns `name` and `firstName` are missing from the tables `Family` and `Person`, respectively. Also, we can see that the second key of the table `Family_members` is named `PersonId` instead of `membersId`. Therefore, according to this test case, the version of Listing 1 using the commented code presents a faulty transformation containing semantic errors. Note that the AnATLyser [8] tool did not detect any syntactic errors in this erroneous version of the transformation.



(a) Expected output model (b) Obtained output model

Fig. 3 Differences between the expected output model (a) and the output model obtained with the transformation of Listing 1 using the commented code (b)

2.3 Repair Patches

Program repair can be defined by *the transformation of an unacceptable behavior of a program into an acceptable one according to a specification* [25].

We call a *patch* a sequence of edit operations which modifies a transformation's source code. A patch is considered good if it modifies a transformation to conform to a given specification. In our case, if a patch modifies the transformation so that the obtained output models are equivalent to the expected ones, then this patch is considered optimal to repair the transformation.

Table 1 presents a subset of the atomic edit operations for ATL transformations proposed by Cuadrado *et al.* [9]. This subset corresponds to the operations modifying elements in the transformation rules, as presented in [41]. We thus use these edit operations to compose the patches to repair faulty ATL transformations. We also employ these edit operations in our evolutionary algorithm to *mutate* proposed patches in Section 3.1.

The two operations *Binding creation* and *binding deletion* respectively add a new and remove an existing binding in a given rule. *Type of source pattern element* modifies the `from` part of a rule, while *Type of target pattern element* changes the `to` part of a rule. *Type of collection* modifies collection data types provided by OCL (e.g., Sequences, Set, Bag). *Type argument of operation* changes the arguments of type-testing operations such as `oclIsKindOf()` and `oclIsTypeof()`. *Navigation expression* and *Target of binding* respectively change a given binding's right-hand side and

left-hand side. Finally, the three operations *Collection operation call*, *Iterator call* and *Predefined operation call* change a call by another.

All these operations take parameters to define on which element it should be applied, as well as the modified values when applicable. For instance, the edit operation *Navigation expression* considers four parameters: the rule, the element of the rule, the old value and the new value which should replace it.

Table 1 Atomic edit operations to modify ATL transformation programs, taken from [9, 41].

Target	Type
Binding	Creation
Type of source pattern element Type of target pattern element Type of collection Type argument of operation	Type modification
Navigation expression (binding RHS) Target of binding (binding LHS)	Feature name modification
Predefined operation call Collection operation call Iterator call	Operation modification
Binding	Deletion

Figure 4 shows an example of a patch composed of three edit operations used to correct the transformation of Listing 1, i.e., when applied on the version using the commented code, the patch modifies it to the non-commented version. The first operation replaces the operation call `excluding()` by `union()` in the rule *Class2Table* (original: line 7, corrected: line 9). Similarly, the second operation replaces the operation call `collect()` by `select()` in the same rule (original: line 8, corrected: line 10). The third operation changes a binding right-hand side in the rule *MultiValuedClassAttribute2Column*: it replaces `a.type` by `a.name` (original: line 32, corrected: line 33). This patch therefore modifies the faulty transformation behavior, and the patched transformation produces the expected output model. This three-edit patch is thus considered optimal to repair the transformation with regards to the provided test case.

2.4 Formulating Repair Patches

Designing patches to repair semantic errors is a difficult endeavor which requires an expertise in the transformation language, the meta-models and the transformation itself. Input/output in test cases may reveal the presence of semantic errors, but do not provide a clear indication of what is causing the errors, nor the rules in which they may occur. Detecting and fixing errors related to transformations' behavior is even more difficult because of the declarative nature of transformation languages such as ATL. Moreover, gathering reusable knowledge about model transformation repair on which we could build automated or semi-automated approaches to assist experts in this task is tedious. In fact, transformations are very dissimilar (notably because most of the transformation depends on the meta-models) and there are few available repositories of them. In such situations, an alternative is to formulate the task as an *optimization problem*, where the goal is to automatically find optimal solutions in the space of all possible solutions.

Formulating transformation repair as an optimization problem, an optimal solution represents a patch fixing the errors of the transformation, and the space of solutions to be explored is thus equal to the set of all possible patches which could be applied on the faulty transformation. However, this space cannot be explored exhaustively. Indeed, each error in a transformation can potentially be repaired by choosing one or many edit operations, and each edit operation may involve any possible instance of elements in the input and output meta-models. Alternative methods are then necessary to efficiently explore this space.

2.5 Problem Formalization

Our approach requires the definition of a *fitness function* (Section 3.1) which calculates how well a patch fixes errors in the transformation. This section provides the formal basis to define the terms in that fitness function.

We begin by representing our transformation of interest by tr. Associated with this transformation tr is a *test suite*, where each test case in that suite consists of one input model and its corresponding output model (Section 2.2). Equation 1 formalizes this notion of a test suite T.

$$T_{tr} = \{(\text{in}_0, \text{out}_0), (\text{in}_1, \text{out}_1), \ldots, (\text{in}_n, \text{out}_n), \} \quad (1)$$

[edit-1] **OperationCall(** *rule*=Class2Table, *object*=out, *old*=excluding, *new*=union)	[edit-2] **OperationCall(** *rule*=Class2Table, *object*=out, *old*=collect, *new*=select)	[edit-3] **NavigationExpression(** *rule*=MultiValuedClassAttribute2Column, *object*=foreignKey, *old*=a.type, *new*=a.name)

Fig. 4 Example of a patch to repair the transformation of Listing 1 to conform to the behavior defined by the expected output model of Fig. 3

We are interested in how a transformation will transform the input models in a test suite, and how these *produced output models* differ from the *reference output models* in that test suite. That is, when we execute the transformation tr on the input model in_i to obtain $tr(\text{in}_i)$, how does this differ from the reference output model out_i? We define the function *errors* in Equation 2 which collects this set of differences. This will allow for the measuring of *how many* and *which* errors are fixed by patching a transformation. These differences are produced through the *diff* function, which is provided by EMFCompare [5] in our implementation.

$$\text{errors}(T_{tr}) = \{\forall i \in T_{tr} : \text{diff}(t_r(\text{in}_i), \text{out}_i)\} \quad (2)$$

We define a patch p in Equation 3 which is a sequence of *edit operations* as shown in Figure 4.

$$p = (e_0, e_1, \ldots, e_n) \quad (3)$$

We then define a patching function $\text{patch}(tr, p)$ which applies the patch p to the transformation tr to produce the patched transformation tr_p.

From these constructs, we can then i) collect the errors in a transformation's test suite (Equation 2), and ii) patch a transformation to attempt to fix it. We will combine i) and ii) in our approach to repeatedly patch transformations and determine the fitness of that patch in terms of the errors fixed.

3 Evolutionary Algorithm Patch Creation

This section provides the background on evolutionary algorithms (EAs) required for the next section. We discuss EAs as applied to transformation repair, in the context of the multi-objective algorithm introduced by our previous work [41]. The first objective of this algorithm focuses on scoring patches depending on the provided test cases. We show how we use the expected output models of these test cases to retrieve more precise information regarding the transformation's errors and refine the fitness score. A second objective is also discussed which prevents the patches from growing unnecessarily large during the search, an issue known as *bloating*.

3.1 Evolutionary Algorithms

Evolutionary algorithms (EAs) are search methods used to solve a wide range of optimization problems by efficiently exploring the search space. Their search strategy is inspired by the evolutionary theory: EAs maintain a population of *candidate solutions* which undergo an evolution process through several generations. At each generation, some solutions are *mutated* (i.e., we use an existing solution to create a slightly different solution) and other solutions are *bred* (i.e., several existing solutions are recombined to create new solutions).

The newly created solutions along with the previous solutions are then evaluated and a *fitness score* is associated to each one of them, which reflects how good the solution is to solve the considered problem. The solutions with the best scores have a higher probability to be retained in the population and to go through the next generation, while the others tend to be discarded. By keeping the best solutions at each generation and using them to create new solutions, each new generation should have a population of solutions better suited to fix the problem than the previous one, until an optimal solution is finally found.

Population-based evolutionary algorithms have been studied to find patches to repair general purpose programs [11, 13] and domain specific ones such as ATL [41, 42]. Adapting a problem such as tranformation repair to be solved with EAs revolves around three points: *defining a solution representation*, *genetic operators* and a *fitness function*. In the rest of the section, we discuss these three points and illustrate them on the problem of repairing ATL transformations.

3.1.1 Solution Representation

In EAs, solutions are the central artifacts which are modified, evaluated and retained through generations. Because this process is fully automated, choosing a way to represent solutions that ease their manipulation is essential for the approach to run smoothly. Early EA-based approaches to repair programs used to consider a whole program as a solution: the population included different versions of the program to be repaired (usually in the form of ASTs) and evolved these programs until a correct version was found. This could be costly in time and memory, and the evolution process was complex because it involved modifications on the AST. A more convenient way to represent solutions in these cases is to consider patches in the form of sequences of edit operations as represented in Fig. 4. Sequences are easy to represent and manipulate, especially during the evolution phase, as discussed hereafter.

In the case of ATL transformation repair, a population would gather a set of patches being sequences of variable size of atomic edit operations, as presented in Section 2.3.

3.1.2 Genetic operators

Genetic operators are at the core of the process of evolving solutions of each generation: they enable to obtain new candidate solutions based on the ones present in the population. EAs usually rely on two types of genetic operators: *mutation* and *crossover*. The mutation operator takes one solution as input and outputs a slightly modified solution. The crossover operator recombines two existing solutions (parents) to create two new solutions (children) composed of rearranged parts of their parents. Usually, the operators are applied randomly until the population of solutions doubles in size.

In the case of evolving patches to repair faulty ATL transformations, the *mutation operator* applies a mutation on one patch. The considered mutations here are 1) adding an edit operation, 2) removing an edit operation and or 3) modifying an edit operation. Fig. 5 presents an example of two mutations applied on the patch of Fig. 4. The first mutation replaces *[edit-1]* with another type of edit operation (target of binding) and the second mutation only modifies one parameter of *[edit-2]*.

The *crossover operator* takes two patches and outputs two new patches representing a recombination of the inputs. In other words, it cuts the two sequences of edit operations in several parts (sub-sequences) and recombines them to create new sequences. Representing solutions as sequences is thus convenient when performing crossover operations. In this work, we used a single-point crossover operation, which separates patches in two parts and exchanges their right parts.

Fig 6 represents a single point crossover on the patch of Fig. 4 and another arbitrary patch of two edit operations.

3.1.3 Fitness function

After the evolution phase, the fitness function is invoked on each solution to compute their fitness score. This score should reflect how good a solution is to solve the problem, and is used to rank the solutions. This ranking is then used to select the better half of the population, and discard the solutions with poor fitness.

As explained previously, a way to detect the presence of semantic errors in ATL transformations is by relying on input/output test cases. If a patch, when applied to the faulty transformation, modifies the later such that it produces the expected output models for all input models, then the patched transformation is semantically correct with regard to the provided behavior specification, and the patch is thus considered optimal. In this case, the fitness function could associate to each patch a score corresponding to the number of passing test cases. At each generation, the fitness function would thus favor the patches passing the most test cases, until finding one passing them all.

3.2 Multi-Objective Evolutionary Algorithm

Multi-objective optimization problems introduce the idea that the fitness of candidate solutions may be evaluated based on several objectives, which may conflict with each other. Evolutionary algorithms are hence designed to find a set of near-optimal solutions, called non-dominated solutions (or Pareto front). These non-dominated solutions provide a suitable compromise between all objectives without degrading any of them.

[edit-1] **TargetOfBinding**(*rule*=Class2Table, *object*=out, *old* =col, *new*=key)	[edit-2] **OperationCall**(*rule*=Class2Table, *object*=out, *old*=collect, *new*=union)	[edit-3] **NavigationExpression**(*rule*=MultiValuedClassAttribute2Column, *object*=foreignKey, *old*=a.type, *new*=a.name)

Fig. 5 Examples of two mutations of the patch of Fig. 4

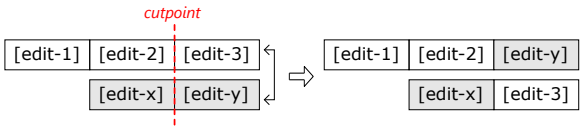


Fig. 6 Examples of a single point crossover operation

Thus, non-dominated solutions are not comparable and can be considered equally good. In this paper, we use NSGA-II [10], a well-known fast multi-objective genetic algorithm, that is suitable to the kind of problem we are solving [1].

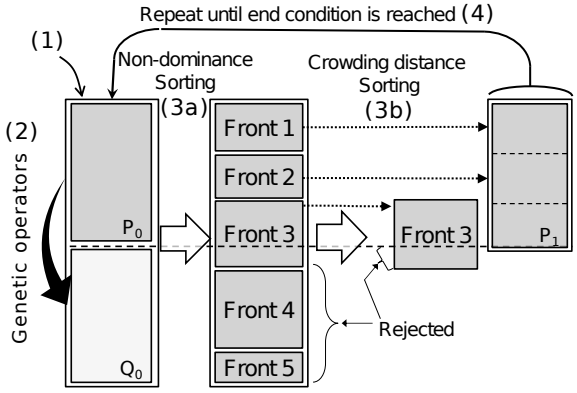


Fig. 7 NSGA-II Algorithm [10]

Figure 7 presents the four main steps of NSGA-II. The first step in NSGA-II is to create randomly a population P_0 of $N/2$ solutions (Fig. 7 (1)). Then, genetic operators are applied on the solutions of the population P_0 to create a child population Q_0 of the same size (2). Both populations are then merged into an initial population of size N. The populations are sorted into dominance fronts according to the dominance principle (3a). A solution s_1 dominates a solution s_2 for a set of objectives $\{O_i\}$ if $\forall i, O_i(s_1) > O_i(s_2)$ and $\exists j | O_j(s_1) > O_j(s_2)$. The first front includes the non-dominated solutions. The second front contains the solutions that are dominated only by the solutions of the first front, and so on and so forth. 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 (see Section 4) in the solutions [10] (3b). This process is repeated (4) until an optimal solution is found in Pareto set or a stop criterion is reached, e.g., a number of iterations or one or more objectives greater than a certain threshold.

3.2.1 Objective 1: Fixing as Many Errors as Possible

The first objective of our algorithm (retained from our previous work [41]) targets the main goal of the approach (i.e., repairing transformations): it scores patches depending on their capability to fix errors in the faulty transformations. As stated before, test cases are traditionally used to estimate the utility of a patch: the more test cases pass, the better the patch. In the case of ATL transformations, test cases are pairs of input/output models: provided with the input models, a correct transformation should output the expected models. To assess a patch, first the sequence of edit operations is applied on the faulty transformation to obtain a patched transformation. Then, the input model of the test case is given to the patched transformation to produce an output model: if the obtained model is equivalent to the expected output, then the test case passes. If the obtained model is different from the expected one, the test fails.

Failing test cases do not usually provide information regarding why they fail, or how close they were to pass. However, working with test cases based on model comparison gives us the opportunity to refine the fitness score by considering the differences between the two output models. The idea is that even if a patch does not correct all the errors and does not pass all the tests, a partial solution should lead to less discrepancies between the output models and the expected ones compared to a random solution.

For this objective, we compare the output models generated by the patched transformation with the expected ones provided by the test cases to compute the number of differences between these models. We rely on EMFCompare [5], a tool which, given two models, outputs a list of differences between them, in a similar manner than the differences presented in Fig. 3. The higher the number of differences, the less fit the patch is considered, and it thus receives a bad score. An optimal patch is a patch for which the number of differences is zero. When such a patch is found, the process stops.

Recalling the formalization from Section 2.5, this objective is stated in Equation 4. Here, we simply minimize the number of errors present in the test suite for a patched transformation.

$$\arg\min_{p} f_1(p) = \left|\text{errors}(T_{tr_p})\right| \quad (4)$$

The patch presented in Fig. 4 is optimal because it produces the expected output model: it fixes the three differences indicated in Fig. 3. Non-optimal patches could either (a) introduce new differences, (b) do not change the output models or (c) correct some differences but not all. Figure 8 shows an example of scores given to partial patches inspired from the optimal patch of Fig. 4. The unmodified faulty transformation (no patch) produces an output model with three differences. If we consider a patch composed of the two first edit operations of Fig 4, it fixes differences 1 and 2, but not 3. A patch composed only of the third operation *[edit-3]* fixes the difference 3 but not the differences 1 and 2. This patch can be thus not considered as good as the previous one, because it fixes one less difference. However, it is still better than no patch at all. To properly assess the fitness of patches and compare patches, it is best to consider several examples to approximate the expected behavior of a transformation.

Even if this score is more fine-grained than relying on the number of passing test cases, it is not enough to prevent *fitness plateaus* [41]. These fitness plateaus are addressed in the current work in Section 4.

In cases where several errors needed to be corrected in one transformation, many of them needed to be corrected at the same time to see an improvement in the output models. If two errors

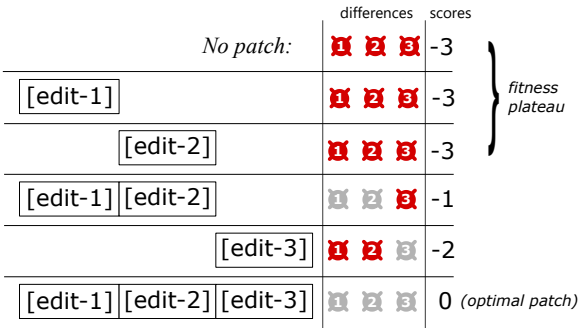


Fig. 8 Assessing patch fitness depending on model differences

disturb similar parts of the output models, correcting only one of them would not improve the output models, thus both of them need to be corrected at the same time to notice any improvement, hence hindering the detection of partial solutions, as shown in Fig. 8. [edit-1] and [edit-2] both modify the same binding (lines 7-8) in the rule Class2Table. To notice an improvement in the fitness scores, they both need to be present in the patch, otherwise, the score is as good as for no patch at all. However, the fitness plateau is smaller than the one shown in Fig. 9, when only the numbers of passing test cases were used to assess the fitness of a patch.

3.2.2 Objective 2: Controlling the Size of the Generated Patches

Bloating is a known issue in EAs where the solutions considered during a run grow in size and become larger than necessary to represent good solutions. This is unpleasant because it slows down the search by increasing manipulation and evaluation time, and find good solutions which are unnecessary large and complex. In multi-objective EAs, dedicating an objective to give better scores to solutions of small size (*Parsimony Pressure*) appeared to be effective to prevent bloating. Thus, we use a second objective (Equation 5), also retained from our earlier work [41], which represents the number of operations in the patch. This objective thus favors patches of small size to avoid generating candidate patches using too many edit operations.

$$\arg\min_{p} f_2(p) = |p| \quad (5)$$

4 Social Diversity Repair Approach

In this section, we propose an extension of our previous approach for automated repair of semantic errors in transformations [41]. The goal of the proposed approach is to overcome the *single fitness peak* and *fitness plateaus* limitations faced by the earlier test-based EA approach when repairing transformations with several errors.

Our hypothesis in this paper is that deliberately maintaining diversity in the population would not only help avoiding single fitness peak but also escaping fitness plateaus, hence increasing the effectiveness and efficiency of test-based EAs approaches. The core of our approach is thus the use of several objectives to guide the search, including one to promote *social diversity*.

Here, we provide a comprehensive overview of our approach. We then present the third objective in our algorithm, which is designed to preserve diversity in the population. We discuss why we think that preserving diversity would help prevent both single fitness peak and fitness plateaus.

4.1 Issues with Convergence

For a given problem, different fitness functions can be designed to achieve the same goal. Carefully designing the fitness function is essential and may impact both the approach's efficiency (time to converge toward an optimal solution) and efficacy (whether it converges towards optimal solution or not). Indeed, the fitness score plays a central role in the search strategy of EAs, because selecting which solutions to retain or discard through the successive generations is what is *guiding the search* by defining which parts of the solution space are explored or not. Relying on test cases to assess the fitness of repair patches can lead to convergence issues of EA repair approaches: we highlight two of them that we target in this paper.

4.1.1 Single Fitness Peak

Groups of similar solutions may have similar fitness scores. Because EAs sift solutions with the highest fitness, it may promote groups of similar solutions if they have a high fitness score. Mutations and crossovers, when applied on these solutions, will mostly produce similar solutions again, with high fitness as well. Such groups may quickly overpower other solutions, leading to a loss of diversity in the population and a premature convergence toward a local optima. This issue is known as *single fitness peak*.

In the case where the fitness function relies on test cases, EAs will tend to promote patches correcting most of the errors. We can end up in a situation where the population is mostly constituted of similar patches correcting the same errors and passing most of the test cases. However, the other solutions that could target the remaining errors are quickly discarded in favor of these patches having a high score, and the necessary material to cover all errors and pass all tests is lost to their profits. Sustaining a certain level of diversity within the population, i.e., ensuring that individuals are scattered in different regions of the search space, increases the chances to find good solutions efficiently.

4.1.2 Fitness Plateaus

Using test cases to define fitness functions may lead to another issue hindering convergence: *partial patches*, i.e., correcting only a part of the defect, are associated with bad fitness score because test cases do not detect and reflect their value. For instance, the patch presented in Fig.4 modifies the illustrative faulty transformation to pass the test case of Fig.3. However, sub-patches (or partial patches) such as {edit-1, edit-2} or {edit-3}, even though they correct part of the defect and are necessary to build the optimal patch, are not enough to pass the test, as illustrated in Figure 9.

These patches are thus indistinguishable from random patches which do not address at all the defects of the transformation, and are discarded early in the process. As a consequence, a lot of candidate solutions (partial or bad) have the same fitness score, thus creating *fitness plateaus*, i.e., large parts of the fitness landscape where all solutions have the same fitness score even though they are different from one another, and even though some of them are partial solutions [11, 36]. This makes some parts of the search space difficult to explore, making it as good as random search because the fitness scores are the same and thus cannot properly guide the search.

In our previous work [41], we used EAs to repair transformations by relying on test cases.

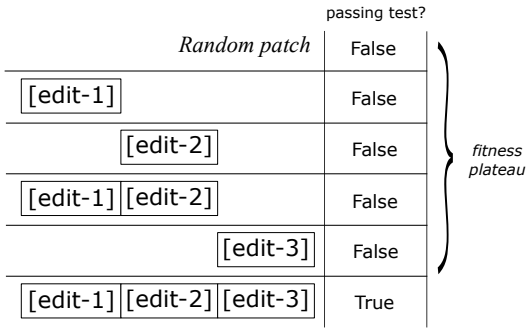


Fig. 9 Example of fitness plateau caused by a fitness evaluation based on passing test cases

We obtained good results for faulty transformations needing less than three edit operations to be repaired (i.e., with few errors): beyond this limit, our approach had trouble converging towards patches addressing all errors. We have analyzed in detail the process of our approach for these cases, notably how candidate patches were selected or discarded through the generations, to understand why the approach was not effective anymore. We found that partial patches (partial solutions) were quickly discarded in the process due to fitness plateaus. The more errors to correct, the larger the size of the plateaus and the less effective the search for an optimal patch.

4.2 Objective 3: Preserving Semantic Diversity

Our third objective addresses the above issue by focusing on promoting the diversity in the population. It is a new objective to the algorithm, as it was not found in our earlier work [41].

The literature recognizes two types of diversity. The first one, called *genotypic* or *syntactic* diversity, distinguishes individuals based on their structure. In our case, syntactic diversity would promote patches of variable size and using dissimilar edit operations.

The second type of diversity is called *phenotypic* or *semantic*. This time, it distinguishes individuals based on their behaviors without considering their structure. Patches having similar size and edit operations but modifying the transformations such that they result in different output models would then be considered semantically diverse.

When targeting semantic errors in transformations, maintaining diversity in transformations' behaviors is highly relevant. On the other hand, understanding the impact of syntactic diversity on the behavior of a program is quite complex [23]. We thus focus on semantic diversity, which is also known to be more efficient to prevent single fitness peak [4, 40].

4.2.1 Social Semantic Diversity

The aim of a social diversity measure is to assess a candidate solution not only by examining the solution alone, but also by considering the solution as a part of the population. When repairing transformations, a social diversity measure would consider that the value of a patch should not be restricted to the number of errors it corrects, but should also consider its capability to address errors which are infrequently covered by the other patches of the population. In other words, it aims at assessing the value a candidate patch brings to the entire population.

Batot et. al [4] proposed a social diversity measure giving higher scores to solutions which pass test cases frequently failed by the other solutions. A solution passing numerous test cases that the majority of the population also pass will receive a lower score than a solution passing less test cases but which are failed by a majority of the population. They show that considering this measure to score solutions allows to reduce single fitness peak when using test cases conformance to guide the search.

In this paper, we propose a *social diversity measure* relying, not on the number of passing test cases, but on the differences between the obtained output models and the expected ones. Because these differences give information about what part of the output models differ from what is expected, we are able to estimate which parts of the output models are impacted by each patch. We use this information to give higher scores to patches modifying parts of the output models which are less covered by the other patches of the population.

As discussed previously, correcting transformations with many errors increases the chances to have several errors impacting the same parts of the output models. These errors need to be fixed at the same time to notice a difference in the output model, and thus an improvement in the

fitness score. We think that bringing social diversity in our fitness function will help maintain a population of patches addressing different parts of the output models, thus increasing the chances to escape fitness plateaus caused by errors interactions. Moreover, using a social diversity measure as an objective would refine the fitness score by adding a new level of granularity, thus helping reduce the size of the plateaus.

4.3 Social Diversity Formalization

Social diversity is calculated for patches by determining the uniqueness of the errors they address, compared to the rest of the population. This is calculated by collecting the set of errors for a patched transformation, and comparing this set with the set of errors for the original (unpatched) transformation. Then, we determine for each error whether it is addressed by many patches, or a smaller set of patches. The patches which address a unique set of errors are then assigned a better score.

First, following Section 2.5, let $tr_p = \text{patch}(tr, p)$. That is, we apply a patch p to the original transformation tr to obtain tr_p. We then create the sets of errors for each transformation: errors(tr) and $errors(tr_p)$ following Equation 2 and utilizing EMFCompare for the *diff* function.

A matrix D is constructed to record which of the errors are addressed by each patch. The columns represent the errors $e_i \in \text{errors}(tr)$, while the rows are each candidate patch p_j. Each entry D_{ij} is assigned as 0 or 1 depending on whether the error e_i is still present in $errors(tr_{p_j})$ or not (Equation 6). That is, if a patch fixes an error, then a 1 will appear. If the error is not fixed by a patch, then a 0 appears[3].

$$D_{ij} = \begin{cases} 0 & \text{if } e_i \in errors(tr_{p_j}) \\ 1 & \text{if } e_i \notin errors(tr_{p_j}) \end{cases} \quad (6)$$

Table 2 demonstrates a small example table. In this example, there are three errors in the original transformation's test suite, and there are four candidate patches to assign a diversity score to. For patch p_0, when this patch is applied, then error e_0 is still present in the patch's test suite, represented by a 0 in entry D_{00}. However, patch p_0 fixes errors e_1 and e_2 which is recorded with 1's in those entries.

	e_0	e_1	e_2	**Patch Diversity**
p_0	0	1	1	0.50
p_1	0	1	0	0.00
p_2	1	1	0	0.75
p_3	0	1	1	0.50
Error Fix Rate	1/4	4/4	2/4	

Table 2 Calculation of diversity for patches.

The *fix rate* for each *error* is calculated at the bottom of this table. This score represents in how many patches the error was fixed. Precisely, the score r is the sum of column i divided by the number of patches in the table $|p|$, as defined in Equation 7. For example, the fix rate for e_0 is 1/4, as it was fixed by only 1 out of four candidate patches. On the other hand, e_1 has a fix rate score of 4/4 as it is fixed by each of the four patches.

$$r(e_i) = \Big(\sum_{j=0}^{|p|} D_{ij}\Big)/|p| \quad (7)$$

Finally, we can calculate the diversity score for each patch $s(p_i)$ as given by Equation 8. The sum is taken of terms for each entry across the row, where each of these terms is one minus the fix rate of the error r_{e_i}. This calculation penalizes patches who fix errors that other patches also address, while rewarding uncommon fixes.

$$s(p_i) = \sum_{i=0}^{|errors|} D_{ij} * (1 - r_{e_i}) \quad (8)$$

For an example, let us perform the calculation for three patches p_0, p_1, and p_2 from Table 2.
$s(p_0) = (0*(1-1/4))+(1*(1-4/4))+(1*(1-2/4)) = 0.50$
$s(p_1) = (0*(1-1/4))+(1*(1-4/4))+(0*(1-2/4)) = 0.00$
$s(p_2) = (1*(1-1/4))+(1*(1-4/4))+(0*(1-2/4)) = 0.75.$

Here $s(p_2) = 0.75$ is the highest score, reflecting how patch p_2 fixes the error e_0 which is not fixed by any other patch. The social diversity objective is then defined in Equation 9 where we select for the highest score.

[3]Note that there is a risk that a patch introduces new errors not seen in the original test suite. This situation is not covered in this objective, but instead by objective 2 (Section 3.2.2) which penalizes patches that create many errors.

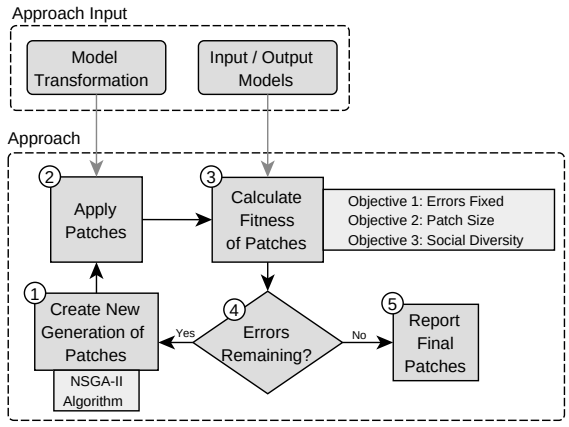


Fig. 10 An overview of our model transformation repair approach.

$$\arg\max_{p} f_3(p) = s(p) \tag{9}$$

4.4 Improved Approach Overview

Fig. 10 provides an overview of our approach to model transformation repair. The approach takes as input the model transformation under study and the test suite of input/output models.

The approach contains a loop of four steps. First, a new generation of patches is created by the NSGA-II algorithm, as denoted by the label "1" in Fig. 10. In the first iteration of the loop, the patches are randomly created. Otherwise, they are selected and mutated based on the fitness scores assigned to each patch.

In step 2, these patches are applied to the model transformation to form a set of patched transformations. These patched transformations are tested against the input/output models in step 3 to determine the fitness of the respective patches, as scored by three objectives.

In step 4, it is determined whether the errors in the transformation have been fixed. If errors remain, then another generation of patches is produced. If no errors remain, then the final set of patches is presented to the user for their inspection.

5 Experiments

We implemented our approach in a tool called *Automatix* and performed an empirical evaluation. This section reports on the evaluation of the impact of our multi-objective approach using social diversity on correcting several semantic errors. We perform our evaluation on four existing third-party transformations, where two were examined in our earlier work [41]: *Class2Table*, *PNML2PN*, *Bibtex2Docbook* and *UML2ER*. We formulate the following research questions:

RQ1: What is the impact of social diversity on the effectiveness of the approach (i.e., finding a patch correcting all the errors)?

RQ2: What is the impact of social diversity on the efficiency of the approach (i.e., the convergence time)?

RQ3: What is the impact of social diversity on the type of errors which are corrected?

5.1 Dataset

We performed our evaluation on existing faulty transformations from the literature. In previous work [41], we utilized 13 faulty versions of the Class2Rel transformation, and 18 faulty versions of the PNML2PN transformation. We reused these 31 faulty transformations versions in these experiments.

Then, we complete this dataset with transformations having three errors or more to assess the impact of social diversity in these cases[4]. Guerra *et al.* [16] introduced an approach for mutation testing in ATL transformations. They note that mutation testing process needs mutants coming from distinct error categories.

We retrieved the UML2ER mutants and the Bibtex2Docbook mutants from their paper. We tested each mutant with the AnAtlyser tool [8], which finds a wide range of syntactic errors (including type errors) in ATL transformations using static analysis. We only select mutants containing semantic errors and we discarded the mutants with syntactic errors. Out of the 800/354 mutants for Bibtex2Docbook/UML2ER studied in their paper, 101/48 of them were syntactically correct but presented semantic discrepancies with the original transformations. However, these

[4]Experimental data is available at https://github.com/jgalasso/faulty-ATL-transformations

mutants only have one semantic error. We reused the approach presented in [42] to merge several mutants with one error to obtain mutants with several errors. We applied this approach on UML2ER and Bibtex2Docbook mutants to create mutants with multiple semantic errors.

To identify semantic error types in faulty transformations, we determine how many atomic modifications need to be performed to correct it: the type of elements of the faulty transformation that should be modified determine the semantic error type. Types of semantic errors are thus strongly related to the edit operations used in this approach. We identified nine kinds of elements that could be modified by an atomic edit operation: a) the types of input/output patterns, b) the operation calls and their arguments, c) the types of collections, d) the properties of input/output object, and e) the bindings (missing bindings and extra bindings). Table 3 presents the 9 different classes of semantic errors, as well as their occurrences in the faulty transformations used in the experiments. We can see that each error type is well represented in our dataset.

Table 3 Classes of semantic errors found in our experiment ATL transformations.

Id	Type of semantic errors	Occurrences
TOP	Wrong type of output pattern	32
TIP	Wrong type of input pattern	13
OP	Wrong operation call	22
TA	Wrong type argument	19
CT	Wrong collection type	9
BL	Wrong property in binding LHS	29
BR	Wrong property in binding RHS	30
MB	Missing binding	29
EB	Extra binding	21

Previous work [42] showed that multi-objective genetic programming faces convergence issues to repair faulty transformations having three or more errors. To study the impact of social diversity on higher numbers of errors, we thus created four sets with respectively two to five mutants and then merged them in each set to form four faulty transformation mutants with two to five semantic errors. We ran this creation process five times for both UML2ER and Bibtex2Docbook mutants. In the end, we acquired 20 faulty versions of each transformation (5 * 4 mutants, each having 2 to 5 errors). Table 4 presents information characterizing the four transformations and their input/output metamodels.

For comparison, we examined the size (in terms of number of rules) of the 106 ATL transformations from the ATL Zoo and discovered out that a transformation has an average of 11 rules, with Q1 = 5, Q3 = 12 and the median being 9. The four selected transformations of our evaluation are thus representative of transformations found in the ATL Zoo.

5.2 Process

In this experiment, we aim at testing social diversity with two configurations separately: as a *crowding distance* and as an *objective*. We believe this helps assess the impact of social diversity on convergence from two different perspectives. Using a social diversity measure as a crowding distance will help hamper a loss of diversity without altering the fitness function. Thus, it would help understand how diversity in the population impacts the resolution of problems whose fitness landscapes contain large plateaus, and thus if social diversity can help escape such plateaus. Using a social diversity measure as an objective would refine the fitness score by adding a new level of granularity, thus helping reduce the size of the plateaus.

We thus adapt our approach to run with three different configurations: a) *without social diversity*, b) with *social diversity as a crowding distance*, and c) with *social diversity as an objective*. We run our approach on all transformation mutants (71 in total) with the three configurations to compare the results.

Note that in our earlier work [41], we only considered approach a), and applied it to two transformations. In this paper, we have added two transformations, so as our extension we are running the approach a) on the two new problems as well and the social diversity approaches b) and c) on all four problems.

We set a maximum number of generations to 50,000 as an arbitrary cut-off.If an optimal patch, which fixes all the semantic errors, is found before attaining the 50,000 generation, the program stops and the number of generations needed to find the patch is preserved. If no optimal patch is found at the end of the 50,000 generations, we retain the best patch found (i.e., the one with the best

Table 4 Transformations used in the evaluation. Cells with two values (X/Y) represent values from the input and output metamodel respectively.

# of	Class2Table	PNML2PN	Bibtex2Docbook	UML2ER
Lines of Code	136	91	232	79
Rules	8	5	9	8
Helpers	4	0	4	0
Classes	6/5	13/9	21/8	4/8
Attributes	3/1	4/3	9/2	87/2
Associations	11/8	28/20	21/9	7/10
Inheritance associations	5/3	14/8	18/4	3/7

fitness score) at the end of the last generation. Because EAs are probabilistic approaches, we run our process five times on each mutant and for each configuration to be able to compute averages. We thus run the 71 distinct mutants five times, for a total of 355 runs for each configuration.

To answer RQ1, we compare the effectiveness of each configuration, i.e., the number of times a run can find an optimal patch. To answer RQ2, we compare the efficiency of each configuration, i.e., the number of generations necessary for a run. To answer RQ3, we applied the obtained best patches on the faulty transformations, and we manually compared the patched transformations with their correct versions to identify the number of remaining errors (if the patch was not optimal) and their types.

5.3 Results

RQ1: What is the impact of social diversity on the effectiveness of the approach (i.e., finding a patch correcting all the errors)?

The percentages of runs that find an optimal patch for all four transformations are shown in Fig 11. The results show an improvement in finding optimal patches in configurations using social diversity in three problems out of four: *Class2Rel*, *Bibtex2DocBook* and *UML2ER*.

Class2Rel and *PNML2PN* mostly include mutants with one or two errors. As we have seen before, the patch generation approach without social diversity already worked effectively in these cases, which explains why social diversity does not introduce huge improvements. Note that among the transformations studied in our previous work [41], *Class2Rel* represented the the largest and more complex ones, which were the most difficult to handle with the approach without diversity. Even if the improvement is not important, it is still noticeable that injecting social diversity helped increase the effectiveness of these difficult cases.

PNML2PN is the only case in which social diversity does not increase the effectiveness of the initial approach. However, the percentages are so close that they are not really significant: we cannot conclude that social diversity reduces the effectiveness. *PNML2PN* is less complex than *Class2Rel* and contains few mutants with more than two errors. The configuration without social diversity already gives very good results on this case, where over 80% of the runs found an optimal patch. Thus, social diversity does not bring improvement in these easy cases.

For *Bibtex2DocBook* and *UML2ER*, we noticed sizable improvements for finding optimal patches when injecting social diversity in the process. In both cases, social diversity as an objective yields better results than as a crowding distance. Because these two cases mostly contain mutants with three errors or more, we expect their fitness landscape to contain more plateaus than the ones of *Class2Rel* and *PNML2PN*.

The results of social diversity as a crowding distance suggests that diversity indeed helps escape these plateaus in certain cases, improving the effectiveness from 34% to 44% for Bibtex2DocBook, and from 57% to 70% for UML2ER. But considering social diversity as an objective (which should reduce the size of plateaus) provides even better results, attaining an effectiveness of 83% and 85% for Bibtex2DocBook and UML2ER, respectively.

We can conclude that using **social diversity both as crowding distance and as objective improves the correction of larger number of errors at the same time**.

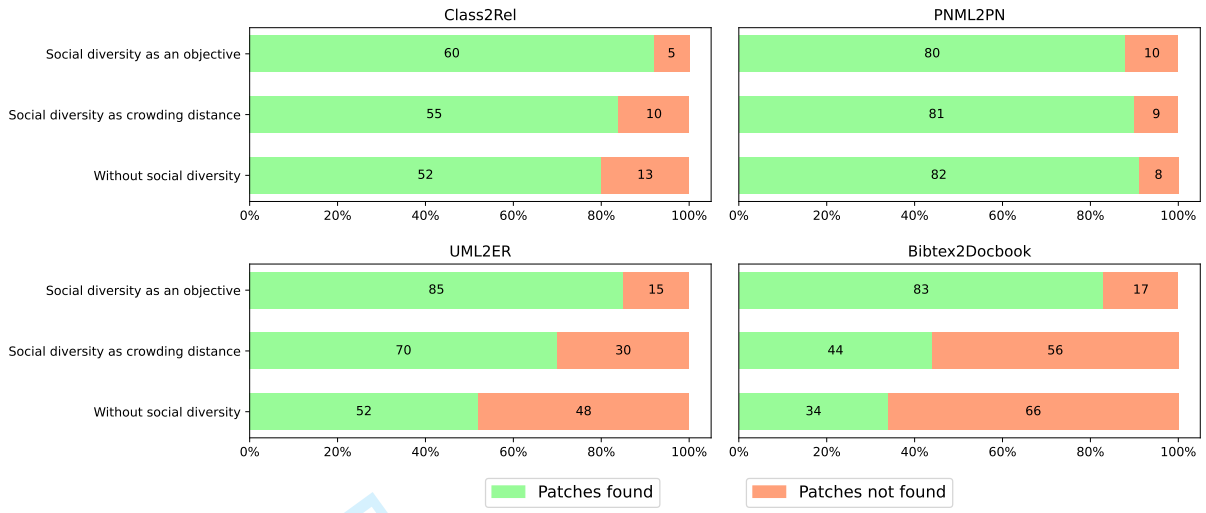


Fig. 11 Percentage of runs finding an optimal patch (RQ1)

RQ2: What is the impact of social diversity on the efficiency of the approach (i.e., the convergence time)?

Figure. 12 shows the average number of generations to obtain a solution for the four studied transformations, and depending on the three configurations.

Overall, the average number of generations required to find a good patch when injecting social diversity in the process is decreased for all four transformations. In RQ1, social diversity did not substantially increase the effectiveness of the approach for *Class2Rel* and *PNML2PN*, which represent transformations with a smaller number of errors. However, Fig. 12 shows that social diversity improves its efficiency. Here again, social diversity as an objective give better results than as a crowding distance for *Class2Rel*. For *PNML2PN*, social diversity as an objective or as crowding distance provided similar results, but they are both better than the initial configuration without social diversity.

For *Bibtex2Docbook* and *UML2ER*, even though the convergence time is better with both configurations including a social diversity measure, the one adding social diversity as an objective brings a higher improvement than the one using social diversity as a crowding distance. In fact, for these transformations with many errors, injecting social diversity through the crowding distance is more effective than the initial approach but the differences are not that important. This suggests that injecting diversity without altering the fitness function increases the chances to find optimal patches (see RQ1), but the exploration is still difficult and the convergence takes time. Reducing the plateaus' size by introducing the diversity measure as an additional objective, however, seems to ease the exploration process, leading to a fastest convergence and a better effectiveness.

Thus, we conclude that using **a social diversity measure helps the approach find the optimal solutions faster**.

5.4 RQ3: What is the impact of social diversity on the type of errors which are corrected?

To answer RQ3, we first retrieve the number and type of errors present in all mutants. For each type of semantic error, we computed their occurrences in the studied mutants. Since we run each mutant five times for each configuration in our approach, we multiply the total number of errors five times to correctly calculate the ratio of corrected/remaining errors. Finally, we counted the total number of errors that are corrected/not corrected by the best patch found at each run. We repeated this process for the three configurations. At the end, we obtained, for each error type, the total number

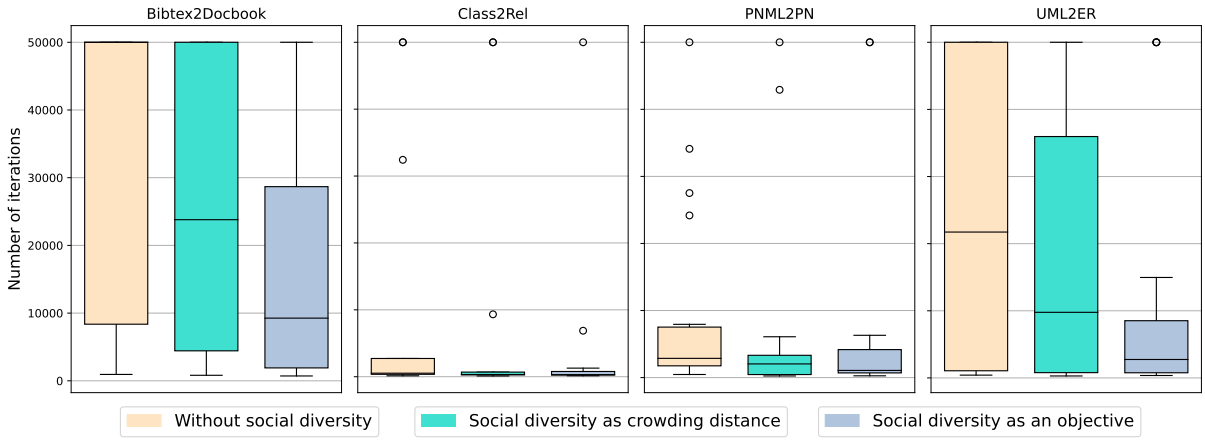


Fig. 12 Number of generations (convergence time) to find a solution

of their occurrences in the mutants and the percentage of corrected errors for each configuration, as shown in Fig. 13.

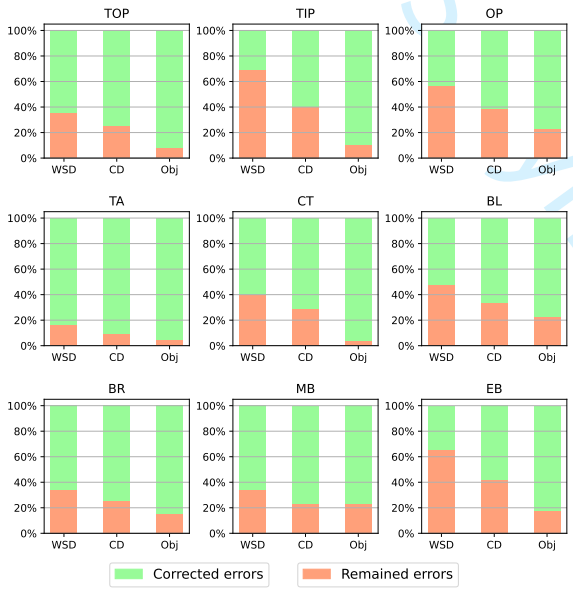


Fig. 13 Percentage of corrected/remained errors for each type of semantic error in faulty transformations, without social diversity (WSD), with social diversity as crowding distance (CD) and with social diversity as an objective (Obj).

We can see that **injecting social diversity, both as a crowding distance and as an objective, increases the correction rate for all types of errors**. For example, the correction rate of semantic errors related to a wrong type argument (*TA*) increased from 83.15% (without SD) to 90.53% (social diversity as a crowding distance) and to 94.74% (social diversity as objective). Errors of type EB, OP and TIP are difficult to repair without social diversity: more than 50% of them remain after applying the best patch. Considering a social diversity measure in the fitness function allows to decrease this percentage to 20% or less for these three cases. Here again, social diversity as an objective provides better improvement than social diversity as a crowding distance.

Even if social diversity improves the correction rates of all types of errors, some of them remain more difficult to fix than other. Those include the three types which were the most difficult to handle without social diversity (EB, OP, TIP). We performed a behavior analysis of our automated approach, especially on the candidate patches which are discarded or kept at each iteration, to understand why some errors remain more difficult to correct than the others. We observed that combinations of these types of errors are more likely to cause interaction, i.e., impact the same parts of the output models and need to be fixed at the same time to see a improvement in the fitness score.

This evaluation shows that social diversity can help overcome the limitations caused by fitness plateaus, which occur when trying to find complex patches (i.e., correcting several errors) while

guiding the search with test cases. We showed that in our case, injecting social diversity in the population helps improve the effectiveness of the approach for repairing more than two errors. The convergence time is also reduced but remain high, suggesting that the exploration is still difficult. We also showed that refining the fitness function by adding social diversity as an objective improve both the effectiveness and the efficiency of the approach. It creates a smoother fitness landscape, more suited for the exploration process. Finally, this evaluation highlighted that some types of errors are more difficult to repair than other, because they are more likely to form fitness plateaus when combined with each other.

6 Discussion

This section will briefly discuss the benefits and limitations of our approach, and the threats to validity for the current work.

6.1 Approach Benefits

Our approach aims to improve the (semi-) automatic repair of model transformations primarily in terms of convergence. The innovation is to use the metric of *social diversity* (Section 4) which ensures that the patches produced are *diverse* throughout multiple evolutionary generations.

This social diversity metric fights fitness plateaus and peaks in the produced patches. Our results show that this leads to finding optimal solutions faster than our earlier work. As well, the retention of the patch size objective means that patches are evolved to be minimal, providing performance enhancements and the exact changes that must be applied to the transformation.

6.2 Approach Limitations

A limitation of our approach is that we are fixing ATL transformation rules, not the helpers. This will require the definition of further edit operations to can modify helpers in ATL transformations.

Another limitation is that due to the evolutionary algorithm-based nature of our approach, it is not possible to ensure that the optimal patches will be found. This is due to the probabilistic nature of evolutionary algorithms which may take a long time to search throughout the solution space to find the optimal solution. This limitation can be seen in our results where the approach occasionally cannot find the optimal solution even after tens of thousands of generations.

6.3 Threats to Validity

There are some threats to validity in our approach as follows. The main threat to validity is the input/output model examples to evaluate the behavior of a transformation, which may not cover all types of semantic errors in a transformation. This causes the incorrect transformations to produce expected output models. We used four different input/output examples to overcome this threat.

A threat to validity of our work is that we tested our approach only with the Atlas Transformation Language (ATL) but we believe that our approach can be generalized with other transformation languages using specific version of edit operations related to the targeted language.

The semantic errors in mutants used in the evaluation originated from mutations and not actually introduced by developers. We used this external data set since it is independent from our project and it covers a large spectrum of semantic errors.

Another threat to validity is the use of specific model transformations originating from the ATL zoo and other model transformation verification papers. These transformations may not be fully representative with real-world transformations in terms of size and complexity. However, we believe that the set of four model transformations used in our experiments is sufficiently representative to demonstrate the benefits of our approach.

7 Related Work

The work presented in this paper intersects three research areas: *program repair in general*, *repairing transformation programs*, and *social diversity*. For a broader examination of model transformation testing and debugging, we point the reader towards the recent survey of Troya *et al.* [39]

7.1 Model Repair

Ben Fadhel et al. [12] use a search-based algorithm to express high-level model changes in terms of

refactorings. Their approach takes a list of possible refactorings, an initial model and its revised version, and searches for a sequence of refactorings characterizing the changes made to obtain the revised model. After applying the sequence of refactorings on the initial model, the obtained model should be as close as possible as the provided revised model. Their approach finds a sequence of edit operations based on model differences but our method applies on transformations.

Puissant et al. [32] proposed an approach to resolve model inconsistencies. They use automated planning to generate one or more resolution plans to repair one error. A change-preserving model repair approach is proposed by Taentzer et al. [37], based on the theory of graph transformation. They consider the edit operations history to identify the inconsistent changes in a model, and complete them with number of possible repair actions to restore consistency. A rule-based repair of EMF models with user intervention is proposed by Nassar et al. [28]. Their approach repair models in a specific context but the efficiency of evolutionary algorithms, in which we used in our approach, is independent from the context. In [21], Kretschmer et al. present an automated approach to explore the space of possible repair values using validation trees to repair model inconsistencies. In comparison, in our approach we explore the space of possible patches using evolutionary algorithms, which is based on random choices and genetic operators, and leads to more diverse solutions. Bariga et al. [2] presented an automatic model repair method which uses reinforcement learning, in which used Markov Decision Process (MDP) and Q-learning algorithm, to repair broken models. Their goal is to generate sequences of edit operations to apply on the whole model, and not just specific errors.

7.2 Transformation Error Detection and Repair

Troya et al. [38] proposed a Spectrum-Based Fault Localization technique, which uses test cases to find the probability of transformation rules being faulty. Oakes et al. [29] presented an approach to statically verify the declarative subset of ATL model transformations. They translated the transformation into DSLTrans and used a symbolic-execution approach to produce representations of all possible executions to the transformation. They verify pre-/post-condition contracts on these representations to verify the transformation. These two works focus on detecting faulty rules in transformation programs and do not repair the faulty rules. Burgueño et al. [6] presented a static approach to check the correctness of transformation rules using matching functions, which used metamodel footprints to automatically generate the alignments between implementations and specifications. Cuadrado et al. [7] presented a combined method using a static analyzer and a constraint solver to detect errors in model transformations. They produced a witness model using constraint solving to make the transformation to execute the erroneous statement. These approaches could find the faulty rules in model transformation, but they cannot fix transformation errors. Cuadrado et al. [9] proposed a tool, Quick fix, to repair syntactic errors in ATL transformations using a static analyzer proposed in [7]. Their approach needs a user interaction to select a suitable repair, while our approach generates a candidate patch automatically. In a previous work [42], we relied on the static analyzer of [7] to automatically generate patches addressing syntactic errors in transformation programs.

Kessentini et al. [18] have implemented an evolutionary algorithm to modify a model transformation to conform to new versions of the metamodels. Their approach aims to adapt models to the new version of metamodels syntactically but not semantically. Rodriguez *et al.* proposed the Model Transformation TEst Specification (MoTES) approach to repair transformations for rule-based languages [33]. Their approach is based on a metric-based test oracle and they used input/output models to mark input/output pattern relationships as true positive, true negative, false positive or false negative. In our approach, we used input/output models as a measure of diversity to choose candidate patches which are less similar to the others for next generation.

7.3 Social Diversity

Soto [35] proposed a study of patch diversity as a means to increase the quality of generated patches through patch consolidation. Their approach focuses on improving patch quality for

general program repair. Ding et al. [11] used a search-based technique for program repair, which is successful when it produces short repairs. The fitness function relies on test cases, which are not enough to determine partially correct solutions and lead to a fitness plateaus. They proposed a novel fitness function using learned invariants over intermediate behavior. Their approach improved semantic diversity and fitness but not repair performance. This approach is similar to ours in the sense that they used the semantic diversity to optimize the fitness function. However, They used invariant-based semantic diversity but we used social diversity in different way. Their method applies on programming languages but ours applies on transformation languages.

Vanneschi et al. [40] divided semantic-aware methods into three categories. *Diversity methods*, that work with diversity, mostly at the population level [19]. *Indirect semantic methods*, that act on the syntax of the individuals and depend on criteria to indirectly promote a semantic behavior [31] [14] [15] [20] . *Direct semantic methods*, that act directly on the semantics of the individuals by using precise genetic operators [26]. All these approaches improve the power of genetic programming. Batot et al. [4] proposed injecting social diversity in multi-objective genetic programming to learn model well-formedness rules from examples and tackle the bloating and single fitness peak limitations. They presented an improvement in population's social diversity that was performed during the evolutionary computation and lead to efficient search strategy and convergence. They implemented the social semantic diversity in NSGA-II algorithm both as crowding distance and as an objective. The difference with our work are that we aim at fixing semantic errors in ATL transformations not learning model well-formedness rules from examples. Interestingly, they obtained better results when injecting social diversity in the crowding distance than as an additional objective. This could be explained by the fact that we target two different issues: they try to limit the loss of diversity to prevent single fitness peak while we try to overcome the issues caused by fitness plateaus.

8 Conclusion and Future Work

In this paper, we presented a novel automated approach to correct many semantic errors in model transformations. This approach is based on evolutionary algorithms and test cases in the form of input/output models to find suitable patches to fix the transformations.

We discuss two limitations of EAs, namely single fitness peak and fitness plateaus, which are known to hinder the convergence of EAs approaches in this case and which make it difficult to find patches fixing three errors or more. To overcome these limitations, our approach is formulated as a multi-objective optimization problem and we use several objectives to guide the search. We dedicate an objective which gives a score based on the notion of social diversity that we defined on model differences.

We performed experiments to assess the impact of our approach, and especially on injecting social diversity in the process, on the effectiveness and the efficiency of repair approaches based on EAs and test cases. Our results showed that injecting our social diversity measure in the search process improves both the effectiveness and the efficiency, and enables to find patches for transformations containing up to five errors.

Our future work will be to examine how the social diversity measure can be combined with objectives focusing on other quality attributes [43]. To improve the convergence of the approach, it may also be possible to better target faulty rules in the transformation through a *spectrum-based fault localization (SBFL) approach* [27, 30, 38]. That is, SBFL would produce a ranking of which rules are likely to be faulty. Then, our EA approach could prioritize patches which repair those rules.

Statements and Declarations

The authors have no competing interests to declare that are relevant to the content of this article.

References

[1] Ali S, Arcaini P, Yue T (2020) Do quality indicators prefer particular multi-objective search algorithms in search-based software engineering? In: the 12th Int. Symp. on Search Based Software Engineering (SSBSE), Springer, pp 25–41, https://doi.org/10.1007/978-3-030-59762-7_3

[2] Barriga A, Mandow L, Pérez-de-la-Cruz J, et al (2020) A comparative study of reinforcement learning techniques to repair models. In: the 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS), Companion Proceedings. ACM, pp 47:1–47:9, https://doi.org/10.1145/3417990.3421395

[3] Barroca B, Lúcio L, Amaral V, et al (2010) DSLTrans: A Turing incomplete transformation language. In: the 3rd Int. Conference on Software Language Engineering (SLE), Springer, pp 296–305, https://doi.org/10.1007/978-3-642-19440-5_19

[4] Batot E, Sahraoui H (2018) Injecting social diversity in multi-objective genetic programming: The case of model well-formedness rule learning. In: the 10th International Symposium on Search Based Software Engineering (SSBSE), Springer, pp 166–181, https://doi.org/10.1007/978-3-319-99241-9_8

[5] Brun C, Pierantonio A (2008) Model differences in the Eclipse modeling framework. UPGRADE, The European Journal for the Informatics Professional 9(2):29–34

[6] Burgueño L, Troya J, Wimmer M, et al (2015) Static fault localization in model transformations. IEEE Trans on Software Eng 41(5). https://doi.org/10.1109/TSE.2014.2375201

[7] Cuadrado JS, Guerra E, d. Lara J (2014) Uncovering errors in ATL model transformations using static analysis and constraint solving. In: the 25th Int. Symposium on Software Reliability Engineering (ISSRE), pp 34–44, https://doi.org/10.1109/ISSRE.2014.10

[8] Cuadrado JS, Guerra E, de Lara J (2018) AnATLyzer: An Advanced IDE for ATL Model Transformations. In: the 40th Int. Conference on Software Engineering (ICSE), Companion Proceedings, pp 85–88, https://doi.org/10.1145/3183440.3183479

[9] Cuadrado JS, Guerra E, de Lara J (2018) Quick fixing ATL transformations with speculative analysis. Software and Systems Modeling 17(3):779–813. https://doi.org/10.1007/s10270-016-0541-1

[10] Deb K, Agrawal S, Pratap A, et al (2000) A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In: Int. Conf. on Parallel Problem Solving from Nature

[11] Ding ZY, Lyu Y, Timperley C, et al (2019) Leveraging program invariants to promote population diversity in search-based automatic program repair. In: 2019 IEEE/ACM International Workshop on Genetic Improvement (GI), IEEE, pp 2–9

[12] ben Fadhel A, Kessentini M, Langer P, et al (2012) Search-based detection of high-level model changes. In: the 28th IEEE International Conference on Software Maintenance (ICSM), pp 212–221, https://doi.org/10.1109/ICSM.2012.6405274

[13] Forrest S, Nguyen T, Weimer W, et al (2009) A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp 947–954

[14] Galvan E, Trujillo L, McDermott J, et al (2013) Locality in continuous fitness-valued cases and genetic programming difficulty. In: EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation II. Springer, p 41–56

[15] Galván-López E, McDermott J, O'Neill M, et al (2011) Defining locality as a problem difficulty measure in genetic programming. Genetic Programming and Evolvable Machines 12(4):365–401

[16] Guerra E, Sánchez Cuadrado J, de Lara J (2019) Towards effective mutation testing for ATL. In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS), pp 78–88, https://doi.org/10.1109/MODELS.2019.00-13

[17] Jouault F, Allilaire F, Bézivin J, et al (2008) ATL: A model transformation tool. Sci Comput Program 72(1-2):31–39

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

[19] Koza J (1992) On the programming of computers by means of natural selection. Genetic programming

[20] Krawiec K, Lichocki P (2009) Approximating geometric crossover in semantic space. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp 987–994

[21] Kretschmer R, Khelladi DE, Egyed A (2018) An automated and instant discovery of concrete repairs for model inconsistencies. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. Association for Computing Machinery, New York, NY, USA, ICSE '18, p 298–299, https://doi.org/10.1145/3183440.3194979, URL https://doi.org/10.1145/3183440.3194979

[22] Lúcio L, Amrani M, Dingel J, et al (2016) Model transformation intents and their properties. Software & systems modeling 15:647–684

[23] McPhee NF, Hopper NJ, et al (1999) Analysis of genetic diversity through population history. In: Proceedings of the genetic and evolutionary computation conference, Citeseer, pp 1112–1120

[24] Mohagheghi P, Gilani W, Stefanescu A, et al (2013) An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empirical Software Engineering 18:89–116

[25] Monperrus M (2018) Automatic software repair: A bibliography. ACM Comput Surv 51(1):1–24

[26] Moraglio A, Poli R (2004) Topological interpretation of crossover. In: Genetic and Evolutionary Computation Conference, Springer, pp 1377–1388

[27] Muñoz P, Troya J, Wimmer M, et al (2022) Revisiting fault localization techniques for model transformations: Towards a hybrid approach. Journal of Object Technology 21(4)

[28] Nassar N, Radke H, Arendt T (2017) Rule-based repair of EMF models: An automated interactive approach. In: Guerra E, van den Brand M (eds) Theory and Practice of Model Transformation. Springer International Publishing, Cham, pp 171–181

[29] Oakes BJ, Troya J, Lúcio L, et al (2018) Full contract verification for ATL using symbolic execution. Softw Syst Model 17(3):815–849

[30] Oakes BJ, Troya J, Galasso J, et al (2023) Fault localization in model transformations by combining symbolic execution and spectrum-based analysis. Software and System Modeling To appear

[31] O'Reilly UM, Goldberg DE (1998) How fitness structure affects subsolution acquisition in genetic programming. In: Genetic Programming 1998: Proceedings of the Third Annual Conference, Citeseer, pp 269–277

[32] Puissant JP, Van Der Straeten R, Mens T (2015) Resolving model inconsistencies using automated regression planning. Software & Systems Modeling 14(1):461–481

[33] Rodriguez-Echeverria R, Macías F, Rutle A, et al (2021) Suggesting model transformation repairs for rule-based languages using a contract-based testing approach. Software and Systems Modeling pp 1–32

[34] Schmidt DC (2006) Model-driven engineering. Computer, IEEE Computer Society 39(2):25

[35] Soto M (2019) Improving patch quality by enhancing key components of automatic program repair. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 1230–1233, https://doi.org/10.1109/ASE.2019.00147

[36] de Souza EF, Goues CL, Camilo-Junior CG (2018) A novel fitness function for automated program repair based on source code checkpoints. In: in the th Genetic and Evolutionary Computation Conference (GECCO), pp 1443–1450

[37] Taentzer G, Ohrndorf M, Lamo Y, et al (2017) Change-preserving model repair. In: Huisman M, Rubin J (eds) Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 283–299

[38] Troya J, Segura S, Parejo JA, et al (2018) Spectrum-based fault localization in model transformations. ACM Trans Softw Eng Methodol 27

[39] Troya J, Segura S, Burgueño L, et al (2022) Model transformation testing and debugging: A survey. ACM Computing Surveys 55(4):1–39

[40] Vanneschi L, Castelli M, Silva S (2014) A survey of semantic methods in genetic programming. Genetic Programming and Evolvable Machines 15(2):195–214

[41] VaraminyBahnemiry Z, Galasso J, Belharbi K, et al (2021) Automated patch generation for fixing semantic errors in ATL transformation rules. In: 24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10-15, 2021. IEEE, pp 13–23, https://doi.org/10.1109/MODELS50736.2021.00011, URL https://doi.org/10.1109/MODELS50736.2021.00011

[42] Varaminybahnemiry Z, Galasso J, Sahraoui H (2021) Fixing multiple type errors in model transformations with alternative oracles to test cases. Journal of Object Technology 20(3):9:1–14. https://doi.org/10.5381/jot.2021.20.3.a9, URL http://www.jot.fm/contents/issue_2021_03/article9.html, the 17th European Conference on Modelling Foundations and Applications (ECMFA 2021)

[43] Wimmer M, Perez SM, Jouault F, et al (2012) A catalogue of refactorings for model-to-model transformations. J Object Technol 11(2):2–1