Social Diversity for ATL Repair

ZAHRA VARAMINYBAHNEMIRY, Université de Montréal, DIRO, Canada
JESSIE GALASSO, Université de Montréal, DIRO, Canada
HOUARI SAHRAOUI, Université de Montréal, DIRO, Canada

Model transformations play an essential role in the Model-Driven Engineering paradigm. Writing a correct transformation program requires to be proficient with the source and target modeling languages, to have a clear understanding of the mapping between the elements of the two, as well as to master the transformation language to properly describe the transformation. Transformation programs are thus complex and error-prone, and finding and fixing errors in such programs typically involve a tedious and time-consuming effort by developers. In this paper, we propose a novel search-based approach to automatically repair transformation programs containing many semantic errors. To prevent the fitness plateaus and the single fitness peak limitations, we leverage the notion of social diversity to promote repair patches tackling errors that are less covered by the other patches of the population. We evaluate our approach on 71 semantically incorrect transformation programs written in ATL, and containing up to five semantic errors simultaneously. The evaluation shows that integrating social diversity when searching for repair patches allows to improve the quality of those patches and to speed up the convergence even when up to five semantic errors are involved.

Additional Key Words and Phrases: Keywords

ACM Reference Format:
Zahra VaraminBaehnemiry, Jessie Galasso, and Houari Sahraoui. 2022. Social Diversity for ATL Repair. 1, 1 (August 2022), 23 pages. https://doi.org/10.1145/nnnnnn.nnnnnnn

1 INTRODUCTION

Model-driven engineering (MDE) has been encouraged for many years as an efficient approach to reduce software development complexity through increasing the abstraction level [24]. In this context, MDE sees models as first class artifacts. It combines domain specific modeling languages to capture specific aspects of the solution, and transformation engines in order to produce from these models low level artifacts such as source code, documentation and test suites [32]. Model transformations effect an essential role in the MDE approach. Model transformation programs depict how to transform elements of a model conforming to a source meta-model into elements of a model conforming to a target meta-model.

They can be written with general purpose programming languages or in dedicated transformation languages such as DSLTrans [3] or the ATLAS Transformation Language (ATL) [18]. Writing a correct model transformation program requires 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. Transformation programs are thus complex and error-prone, and finding and fixing errors in such programs typically involve a tedious and time-consuming effort by developers.

Several types of errors can affect a transformation program. 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 [10] propose predefined corrective patches to be applied on errors detected with syntactic analysis tools (e.g., AnATLyzer [9] for the ATL language). When the transformation program 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 the latters are different from the ones that were expected. Because semantic errors pertain to the transformation’s behavior and each faulty transformation needs tailored patches, predefined patches are not suited for semantic errors.

Population-based evolutionary algorithms (EAs) have been widely used to correct errors in programs [25], including both syntactic [38] and semantic [37] errors in transformation programs. Formulating program repair as an optimization problem enables such search-based approaches to find patches that will fix a given faulty program 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 program’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 a previous work [37], we used EAs with test cases to correct semantic errors in ATL transformation programs. We found that this approach usually finds patches to correct transformations having fewer errors, but either have trouble or take too much time to converge toward good patches in the presence of more errors. Preliminary analysis showed that using test cases to assess the fitness of the corrective patches make the search space difficult to explore efficiently due to fitness plateaus [11], an issue of EAs which impede 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, 11].

In this paper, we propose a new approach based on EAs to automatically find patches correcting many semantic errors which limits the problems related to convergence. To improve the efficiency and effectiveness of EAs using test cases, we propose to leverage the notion of social diversity to promote 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 a certain level of diversity and reduce 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. Our approach is implemented using the NSGA-II algorithm, a fast multi-objective EA [12].

We perform an evaluation on 71 semantically incorrect transformation programs written in ATL to assess the impact of social diversity on the convergence of EA-based repair. We reuse the faulty transformations from our previous work and also consider new transformations taken from the ATL zoo\(^1\) containing up to 5 semantic errors to thoroughly evaluate the impact of our approach on

\(^1\)https://www.eclipse.org/atl/atlTransformations/
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 transformation programs, even when they contain up to 5 semantic errors.

In Sect. 2, we briefly describe ATL transformation programs and give examples of defects in those transformations. We also provide examples of patches repairing such defects. Section 3 presents evolutionary algorithms and illustrate how to use them to repair semantic errors in transformation programs. We also discuss two limitations faced by this approach, related to the use of test cases. Section 4 describes our multi-objective approach using social diversity to automatically generate patches which attempt to overcome these limitations. The impact of breathing social diversity in EAs to improve convergence and repair many errors is evaluated in Section 5. Section 6 presents related work and Sect. 7 concludes the paper.

2 BACKGROUND

In this section, we first provide background about model-to-model transformation programs. We focus on programs written in the ATLAS Transformation Language (ATL) [18], but the approach presented in this paper is generic and can be adapted to other transformation languages. We then discuss the types of errors that can be found in such programs, including semantic errors, which are the target of this work. Finally, we show what is a patch to repair faulty transformation programs.

2.1 ATL Transformation Programs

In this section, we provide some background about model transformation programs written in ATL and illustrate the presented notions on an example inspired from the Class2Relational transformation of the ATL zoo.

Model transformation programs automate the process of transforming a source model into a target model. A transformation program relies on meta-models describing both the source and the target models. The source and target models can conform to the same meta-model (endogeneous transformation), but in the case of exogeneous transformations, transformation programs usually rely on two different meta-models. Thus, a given transformation program 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. For instance, the Class2Relational transformation program 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 as presented in the ATL Zoo Class2Relational transformation.

Listing 1 presents a simplified excerpt of the Class2Relational transformation program written in ATL. The two meta-models are identified on line 1: the source meta-model is Class (IN) and the target meta-model is Relational (OUT). A transformation typically consists of a set of rules defining how to transform elements of the input meta-model into elements of the output meta-model. A rule is introduced by the keyword rule (lines 3, 13, and 20) and has a name (for instance, Class2Table in line 3). Each rule has two parts: a from part outlining elements of the source meta-model, and a to parts defining how to transform elements of the from part into elements of the target meta-model. The from part defines a pattern in the source 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 14-16 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.

https://www.eclipse.org/atl/atlTransformations/#Class2Relational
Fig. 1. UML Class Diagram meta-model (left-hand side) and Relational Schema meta-model (right-hand side)

Listing 1. Excerpt of an ATL transformation program, from Class Diagram to Relational Schema

```
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     key <- Set {key}),
10    key: Relational!Column (  
11     name <- c.name + 'Id))
12
13 rule SingleValuedDataTypeAttribute2Column {  
14 from a: Class ! Attribute (  
15     a.type.oclIsKindOf(Class!DataType)  
16        and not a.multiValued)
17 to out: Relational!Column (  
18     name <- a.name)}
19
20 rule MultiValuedClassAttribute2Column {  
21 from a: Class!Attribute (  
22     a.type.oclIsKindOf(Class!Class)  
23        and a.multiValued)
24 to out: Relational!Table (  
25     name <- a.owner.name + '_' + a.name,  
26     col <- Sequence {id, foreignKey}),
27    foreignKey1: Relational!Column (  
28     name <- a.owner.name.firstToLower() + 'Id'),
29    foreignKey2: Relational!Column (  
30     name <- a.type + 'Id'})
```

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 17-18, a Column is created when an Attribute is matched) or several ones (e.g., in line 5 and line 10, 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 7, a Set in line 9) and may use iterator or operation calls in initialization (e.g., firstToLower() in line 28).

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.

2.2 Defects in Transformation Programs
Transformation programs 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 program. Tools such as AnATLyser [9], a static analyzer for the ATL language, can be used to check syntactic errors.

Semantic errors make a program behave in a way that differs from what is expected, i.e., the transformation program 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 program to produce the wrong outputs. As mentioned earlier, transformation programs are complex and difficult to debug, especially for declarative languages such as ATL. An easy way to outline the intended behavior of a program is to provide a set of inputs-outputs examples of this program, i.e., test cases. If provided with the source models, the program outputs the expected target models, no behavior deviations (indicating semantic errors) are detected. However, if the outputted target models are different from those of the provided examples, it shows that the transformation program is semantically incorrect with regards to the provided test cases. Figure 3 (a) presents the expected target model when providing the transformation program of Listing 1 with the source model of Figure 2 (left-hand side). We can see that it is different from the obtained target model in three different locations, as highlighted 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, Listing 1 presents a faulty transformation program containing semantic errors.

2.3 Repair Patches
Program repair can be defined by the transformation of an unacceptable behavior of a program into a acceptable one according to a specification [25].
We call a *patch* a sequence of edit operations which modifies a program’s source code. A patch is considered good if it modifies a program to conform to a given specification. In our case, we consider behavior specifications in the form of pairs of input/output models: if a patch modifies the transformation program so that the obtained output models are equivalent to the expected ones, then this patch is considered optimal to repair the transformation program.

Table 1 presents a subset of the atomic edit operations for ATL transformations proposed by Cuadrado et al. [10]. This subset corresponds to the operations modifying elements in the transformation rules, as presented in [37]. 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 changes 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 [10, 37].

| Target                        | Type                        |
|-------------------------------|-----------------------------|
| Binding                       | Creation                    |
| Type of source pattern element| Type modification            |
| Type of target pattern element|                            |
| Type of collection            |                            |
| Type argument of operation    | Feature name modification    |
| Navigation expression (binding RHS) |                      |
| Target of binding (binding LHS) |                          |
| Predefined operation call     | Operation modification      |
| Collection operation call     |                            |
| Iterator call                 |                            |
| Binding                       | Deletion                    |
These edit operations can be used to compose patches to repair faulty ATL transformation programs. Figure 4 shows an example of a patch composed of three edit operations applicable on the transformation program of Listing 1. The first operation replaces the operation call `excluding()` by `union()` in rule `Class2Table` (line 7). Similarly, the second operation replaces the operation call `collect()` by `select()` in the same rule (line 8). The third operation changes a binding right-hand side in the rule `MultiValuedClassAttribute2Column`: it replaces `a.type` by `a.name`. This patch, when applied to Listing 1, modifies the faulty transformation behavior: the obtained patched transformation now produces the expected output model. This 3-edits patch is thus considered optimal to repair the transformation program with regards to the provided test cases.

3 PROBLEM STATEMENT

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, transformation programs are very dissimilar (notably because most of the program depends on the meta-models) and there are few available data about 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 program, 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, for the large transformations with many errors, each error 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.

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 probabilities 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 [13, 14] and domain specific one such as ATL [37, 38]. In this paper, we focus on the use of EAs for repairing model transformation programs. Adapting a problem such as program 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 transformation programs.

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.

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.

![Diagram of single point crossover](image)

*Fig. 6. Examples of a single point crossover operation*

**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 is a solution 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 transformation program is by relying on input/output test cases. If a patch, when applied to the faulty program, 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 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 an 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 can lead to convergence issues of EA repair approaches: we highlight two of them that we target in this paper.

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.

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 7. These patches are thus indistinguishable from random patches which do not address at all the defects of the programs, 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, 13]. 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.

| Patch Configuration | Passing Test? |
|---------------------|--------------|
| Random patch        | False        |
| \{edit-1\}         | False        |
| \{edit-2\}         | False        |
| \{edit-1\} [edit-2]\| False    |
| \{edit-3\}         | False        |
| \{edit-1\} [edit-2] [edit-3]\| True  |

Fig. 7. Example of fitness plateau caused by a fitness evaluation based on passing test cases

In a previous work [37], we used EAs to repair transformation programs by relying on test cases. We obtained good results for faulty programs needing less than 3 edit operations to be repaired (i.e., with few errors): beyond this limit, our approach had trouble converging towards patches addressing all errors. We analyzed in details 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 bigger the size of the plateaus and the less effective the search for an optimal patch.

4 OUR APPROACH

In this section, we propose a new approach for automated repair of semantic errors in transformation programs. The goal of the proposed approach is to overcome the two aforementioned limitations faced by test-based EA approaches when repairing programs with several errors. A hypothesis we make 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 the use of several objectives to guide the search, including one to promote social diversity. In what follows, we first present the background related to multi-objective EAs. Then, we present the first objective, focusing 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 program’s errors and refine the fitness score. After that, we present the second objective 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. Finally, we present a third objective to prevent the patches to grow unnecessary large during the search, an issue known as *bloating*.

### 4.1 Multi-Objective Evolutionary Algorithm

Multi-objective optimization problems introduce the idea that 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). A non-dominated solution provides a suitable compromise between all objectives without degrading any of them. Thus, non-dominated solutions are not comparable and can be considered equally good. In this paper, we use NSGA-II [12], a well-known fast multi-objective genetic algorithm, that is suitable to the kind of problem we are solving [1].

![NSGA-II Algorithm](image)

Figure 8 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. 8 (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 in the solutions [12] (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.

### 4.2 Objective 1: Fixing as Many Errors as Possible

The first objective addresses the main goal of the approach (i.e., repairing programs): it scores patches depending on their capability to fix errors in the faulty transformation programs. As stated before, test cases are traditionally used to estimate the goodness of a patch: the more test cases pass, the better the patch. In the case of ATL transformation programs, 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 [6], a tool which, given two models, output 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 receive 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.

The patch presented in Fig. 4 is optimal because it produces the expected output model: it fixes the 3 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 9 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 3 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 [37]. It appears that in cases where several errors needed to be corrected in one program, many of them needed to be corrected at the same time to see
an improvement in the output models. If two errors 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. 9. [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. 7, when only the numbers of passing test cases were used to assess the fitness of a patch.

4.3 Objective 2: Preserving Semantic Diversity

Our second objective focuses on promoting the diversity in the population.

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 transformation programs such that they result in different output models would then be considered semantically diverse. When targeting semantic errors in programs, maintaining diversity in programs’ behaviors is highly relevant. On the other hand, understanding the impact of syntactic diversity on the programs’ behaviors is quite complex [23]. We thus focus on semantic diversity, which is also known to be more efficient to prevent single fitness peak [4, 36].

Social semantic diversity is particularly interesting in our case. 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 transformation programs, 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 programs 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.
4.4 Objective 3: 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 unpleasing 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 third objective, which represents the number of operations in the patch, to favor patches of small size to avoid to generate candidate patches using too many edit operations.

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, 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 transformation programs from the literature. In previous work [37], we utilized 13 faulty versions of the Class2Rel transformation, and 18 faulty versions of the PNML2PN transformation. We reused the 31 faulty transformations studied in this paper.

Then, we complete this dataset with transformations having three errors or more to assess the impact of social diversity in these cases. Guerra et al. [17] introduced an approach for mutation testing in ATL transformations. 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 [9], 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 mutants only have one semantic error. We reused the approach presented in [38] 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. Previous work [38] showed that multi-objective GP face convergence issues to repair faulty transformations having 3 or more errors. To study the impact of social diversity on higher numbers of errors, we thus created 4 sets with respectively 2 to 5 mutants and then merged them in each set to form 4 faulty transformations with 2 to 5 semantic errors. We ran this creation process 5 times for both UML2ER and Bibtex2Docbook mutants. In the end, we acquired 20 faulty versions of each transformation (5 * 4 faulty transformations having 2 to 5 errors). Table 2 presents information characterizing the 4 transformations and their input/output metamodels.

3 All experiment data and code are available at https://github.com/zahravaraminy
Table 2. Transformations used in the evaluation. Cells with two values represent input/output metamodels

|                  | Classe2Table | PNML2PN | Bibtex2Docbook | UML2ER |
|------------------|--------------|---------|----------------|--------|
| LoC              | 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      | 5/3          | 14/8    | 18/4           | 3/7    |

We studied on the size (in terms of number of rules) of the 106 ATL transformations from the ATL Zoo and found 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.

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 9 kinds of elements that could be modified by an atomic edit operation: the types of input/output patterns, the operation calls and their arguments, the types of collections, the properties of input/output object, 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 that can be found in ATL transformation programs.

| 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          |

5.2 Process

In this experiment, we aim at testing social diversity with two configurations (as a crowding distance and as an objective) separately, because we think that they can help assessing 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 impact 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. This time, we could see how social diversity impact the fitness landscape and if it makes it easier to explore.

We thus adapt our approach to run with three different configurations: without social diversity, with social diversity as a crowding distance and with social diversity as an objective. We run our approach on all faulty transformations (71 in total) with the three configurations to compare the results. We set a maximum number of generations to 50 000. 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 fitness score) at the end of the last generation. Because EAs are probabilistic approaches, we run our process 5 times on each faulty transformation and for each configuration to be able to compute averages. We thus run the 71 distinct faulty transformations 5 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 time 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 transformation programs are shown in Fig 10. The results shows an improvement in finding optimal patches in configurations using social diversity in three problems out of four: Class2Rel, Bibtex2DocBook and UML2ER.

![Fig. 10. Percentage of runs finding an optimal patch (RQ1)](image)

Class2Rel and PNML2PN mostly include transformations with one or two errors. As we have seen before, the patch generation approach without social diversity already worked effectively in these cases, which explain why social diversity does not introduce huge improvements. Note that among the transformations studied in our previous work [37], Class2Rel regrouped 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 problem 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 reduce the effectiveness. PNML2PN is less complex than Class2Rel and contains few transformations with more than 2 errors. The configuration without social diversity already gives very good results on this case (near 90% of the runs found an optimal patch): social diversity does not bring improvement in 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 transformations with 3 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 (resulting in reducing the size of plateaus) give 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 correcting larger number of errors at the same time.

RQ2: What is the impact of social diversity on the efficiency of the approach (i.e., the convergence time)? Figure 11 shows the average number of generations to obtain a solution for the four studied transformations, and depending on the three configurations.

![Fig. 11. Number of generations (convergence time) to find a solution](image)

Overall, the average number of generations required to find a good patch when injecting social diversity in the process is smaller for all four transformation programs. In RQ1, we saw that social diversity do not substantially increase the effectiveness of the approach for Class2Rel and PNML2PN, which gather transformations with small numbers of errors. However, Fig. 11 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 gives 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 social diversity 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 faulty transformations. For each type of semantic error, we computed their occurrences in the studied faulty transformations. Since we run each faulty transformation 5 times for each configuration in our approach, we multiply the total number of errors 5 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 of their occurrences in the faulty transformations and the percentage of corrected errors for each configuration, as shown in Fig. 12.

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 2 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.
5.5 Threat to Validity

There are some threats to validity in our approach as follows. A first and 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 produce expected output models. We used four different input/output examples to overcome this threat. Second threat to validity is that we are fixing ATL transformation rules not helpers. We think that if we define the edit operations, which can modify helpers in ATL transformations, then we can use them in our approach and also fix helpers. Another limitation 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 faulty transformations used in the evaluation originated from mutants 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.

6 RELATED WORK
The work presented in this paper intersects three research areas: program repair in general, repairing transformation programs and social diversity. In the following subsections, we discuss this three areas.

6.1 Model repair
Ben Fadhel et al. [5] 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. [30] 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. [34], 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. [27]. 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 [22], 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.

6.2 Transformation repair
Troya et al. [35] proposed a Spectrum-Based Fault Localization technique, which uses test cases to find the probability of transformation rules being faulty. Oakes et al. [28] 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. [7] 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. [8] 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. [10] proposed a tool, Quick fix, to repair syntactic errors in ATL transformations using a static analyzer proposed in [8]. Their approach needs a user interaction to select a suitable repair. On the other hand, our approach generates a candidate patch automatically. In a previous
work [38], we relied on the static analyzer of [8] to automatically generate patches addressing syntactic errors in transformation programs. An impressive work is presented by Kessentini et al. in [19] that 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. [31] proposed the Model Transformation TEst Specification (MoTES) approach to repair transformations for rule-based languages.

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.

6.3 Social diversity

Soto [33] 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. [13] 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. [36] divided semantic-aware methods into three categories. Diversity methods, that work with diversity, mostly at the population level [20]. Indirect semantic methods, that act on the syntax of the individuals and depend on criteria to indirectly promote a semantic behavior [29] [15] [16] [21]. Direct semantic methods, that act directly on the semantics of the individuals by using precise genetic operators [26]. All these approaches help improving 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.

7 CONCLUSION AND FUTURE WORK

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

REFERENCES

[1] Shaukat Ali, Paolo Arcaini, and Tao Yue. 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, 25–41. https://doi.org/10.1007/978-3-030-59762-7_3

[2] Angela Barriga, Lawrence Mandow, José-Luis Pérez-de-la-Cruz, Adrian Rutle, Rogardt Heldal, and Ludovico Iovino. 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, 47:1–47:9. https://doi.org/10.1145/3417990.3421395

[3] Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. 2010. Dsltrans: A turing incomplete transformation language. In the 3rd Int. Conference on Software Language Engineering (SLE). Springer, 296–305. https://doi.org/10.1007/978-3-642-19440-5_19

[4] Edouard Batot and Houari Sahraoui. 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 (SBSE), Springer, 166–181. https://doi.org/10.1007/978-3-319-99241-9_8

[5] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. 2012. Search-based detection of high-level model changes. In the 28th IEEE International Conference on Software Maintenance (ICSME), 212–221. https://doi.org/10.1109/ICSM.2012.6405274

[6] Cédric Brun and Alfonso Pierantonio. 2008. Model differences in the eclipse modeling framework. UPGRAGE, The European Journal for the Informatics Professional 9, 2 (2008), 29–34.

[7] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. 2015. Static Fault Localization in Model Transformations. IEEE Trans. on Software Eng., 41, 5 (2015). https://doi.org/10.1109/TSE.2014.2375201

[8] J. S. Cuadrado, E. Guerra, and J. d. Lara. 2014. Uncovering Errors in ATL Model Transformations Using Static Analysis and Constraint Solving. In the 25th Int. Symposium on Software Reliability Engineering (ISSRE). 34–44. https://doi.org/10.1109/ISSRE.2014.10

[9] J. S. Cuadrado, E. Guerra, and J. de Lara. 2018. AnATLyzer: An Advanced IDE for ATL Model Transformations. In the 40th Int. Conference on Software Engineering (ICSE), Companion Proceedings. 85–88. https://doi.org/10.1109/3183440.3183479

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

[11] Eduardo Faria de Souza, Claire Le Goues, and Celso Gonçalves Camilo-Junior. 2018. A novel fitness function for automated program repair based on source code checkpoints. In in the 7th Genetic and Evolutionary Computation Conference (GECCO). 1443–1450.

[12] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2000. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimisation: NSGA-II. In Int. Conf. on Parallel Problem Solving from Nature.

[13] Zhen Yu Ding, Yiwei Lyu, Christopher Timperley, and Claire Le Goues. 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, 2–9.

[14] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A genetic programming approach to automated software repair. In Proceedings of the 11th Annual conference on Genetic and evolutionary computation. 947–954.

[15] Edgar Galvan, Leonardo Trujillo, James McDermott, and Ahmed Kattan. 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, 41–56.

[16] Edgar Galván-López, James McDermott, Michael O’Neill, and Anthony Brabazon. 2011. Defining locality as a problem difficulty measure in genetic programming. Genetic Programming and Evolvable Machines 12, 4 (2011), 365–401.

[17] Esther Guerra, Jesús Sánchez Cuadrado, and Juan de Lara. 2019. Towards Effective Mutation Testing for ATL. In 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS). 78–88. https://doi.org/10.1109/MODELS.2019.00-13

[18] Frédéric Jouault, Freddy Alliloue, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A Model Transformation Tool. Sci. Comput. Program. 72, 1-2 (June 2008), 31–39.
[19] Wael Kessentini, Houari Sahraoui, and Manuel Wimmer. 2018. Automated Co-evolution of Metamodels and Transformation Rules: A Search-Based Approach. In Search-Based Soft. Eng. Springer, 229–245.

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

[21] Krzysztof Krawiec and Pawel Lichocki. 2009. Approximating geometric crossover in semantic space. In Proceedings of the 11th Annual conference on Genetic and evoloutional computation. 987–994.

[22] Roland Kretschmer, Djamel Eddine Khellali, and Alexander Egyed. 2018. An Automated and Instant Discovery of Concrete Repairs for Model Inconsistencies. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (Gothenburg, Sweden) (ICSE ’18). Association for Computing Machinery, New York, NY, USA, 298–299. https://doi.org/10.1145/3183440.3194979

[23] Nicholas Freitag McPhee, Nicholas J Hopper, et al. 1999. Analysis of genetic diversity through population history. In Proceedings of the genetic and evolutionary computation conference, Vol. 2. Citeseer, 1112–1120.

[24] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel Fernandez. 2013. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. Empirical Software Engineering 18 (2013), 89–116.

[25] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. ACM Comput. Surv. 51, 1 (2018), 1–24.

[26] Alberto Moraglio and Riccardo Poli. 2004. Topological interpretation of crossover. In Genetic and Evolutionary Computation Conference. Springer, 1377–1388.

[27] Nebras Nassar, Hendrik Radke, and Thorsten Arendt. 2017. Rule-Based Repair of EMF Models: An Automated Interactive Approach. In Theory and Practice of Model Transformation, Esther Guerra and Mark van den Brand (Eds.). Springer International Publishing, Cham, 171–181.

[28] James Bentley Oakes, Javier Troya, Levi Lucio, and Manuel Wimmer. 2018. Full contract verification for ATL using symbolic execution. Softw. Syst. Model. 17, 3 (2018), 815–849.

[29] Una-May O’Reilly and David E Goldberg. 1998. How fitness structure affects subsolution acquisition in genetic programming. In Genetic Programming 1998: Proceedings of the Third Annual Conference. Citeseer, 269–277.

[30] Jorge Pinna Puissant, Ragnhild Van Der Straeten, and Tom Mens. 2015. Resolving model inconsistencies using automated regression planning. Software & Systems Modeling 14, 1 (2015), 461–481.

[31] Roberto Rodriguez-Echeverria, Fernando Macias, Adrian Rutle, and José M Conejero. 2021. Suggesting model transformation repairs for rule-based languages using a contract-based testing approach. Software and Systems Modeling (2021), 1–32.

[32] Douglas C Schmidt. 2006. Model-driven engineering. Computer, IEEE Computer Society 39, 2 (2006), 25.

[33] Mauricio Soto. 2019. Improving Patch Quality by Enhancing Key Components of Automatic Program Repair. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 1230–1233. https://doi.org/10.1109/ASE.2019.00147

[34] Gabriele Taentzer, Manuel Ohrndorf, Yngve Lamo, and Adrian Rutle. 2017. Change-Preserving Model Repair. In Fundamental Approaches to Software Engineering, Marieke Huisman and Julia Rubin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 283–299.

[35] Javier Troya, Sergio Segura, Jose Antonio Parejo, and Antonio Ruiz-Cortés. 2018. Spectrum-Based Fault Localization in Model Transformations. ACM Trans. Softw. Eng. Methodol. 27, Article 13 (2018), 50 pages.

[36] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. 2014. A survey of semantic methods in genetic programming. Genetic Programming and Evolvable Machines 15, 2 (2014), 195–214.

[37] Zahra Varaminibahmehmiry, Jessie Galasso, Khalid Belharbi, and Houari A. Sahraoui. 2021. Automated Patch Generation for Fixing Semantic Errors in ATL Transformation Rules. In 24th International Conference on Model Driven Engineering Languages and Systems, MODELS, Fukuoka, Japan, October 10-15, 2021. IEEE, 13–23. https://doi.org/10.1109/MODELS50736.2021.00011

[38] Zahra Varaminibahmehmiry, Jessie Galasso, and Houari A. Sahraoui. 2020. Fixing Multiple Type Errors in Model Transformations with Alternative Oracle to Test Cases. CoRR abs/2012.07953 (2020). arXiv:2012.07953 https://arxiv.org/abs/2012.07953