Improving Repair of Semantic ATL Errors using a Social Diversity Metric

Journal:	*Software and Systems Modeling*
Manuscript ID	SOSYM-23-00004621
Manuscript Type:	Regular Paper
Keyword:	model-driven engineering, model transformations, ATL, evolutionary algorithms, social diversity

December 14, 2023

SOSYM-23-00004493
(Title) **Improving Repair of Semantic ATL Errors using a Social Diversity Metric**
(Authors) Zahra VaraminyBahnemiry, Jessie Galasso, Bentley Oakes and Houari Sahraoui

Dear Dr. Jeff Gray,

Please find enclosed our revised version of our previous submission entitled "Improving Repair of Semantic ATL Errors using a Social Diversity Metric" with manuscript number *SOSYM-23-00004493*. We would like to thank the reviewers for their remarks.

In this revision, we have carefully addressed the reviewers' comments, and another grammar and spelling pass of the paper was also performed. An overview of the modifications and a detailed point-by-point response to the comments are given below.

Overview of Modifications

- Clarified concepts (mutation operators, model equivalency, automation of technique)
- Restructured approach overview to come earlier in the paper and clarify the improvement over the earlier paper.
- Added new limitations and threats to validity.
- Added further related work.

Review 1

Comment 1 *Patches like those shown in the example depicted in Fig. 4 look very transformation-specific, especially concerning the parameters of the considered operations. In particular, the solution space is defined by the type of operations and the input parameters. Indeed, the number of operations is limited, whereas the values of the input parameters can be potentially unbounded.*

Following my previous comment, edit operations have to be adequately formalised in Section 2.5 as done for the other elements of the considered problem. Edit operations given in Equation (3) have to be further detailed to cover all the defining aspects, including the input parameters.

Fig. 5 shows examples of path mutations, and it is unclear where identifiers like "collect" and "union" come from. Indeed, these are ATL operators; thus, it is a specific dimension the algorithm has to explore to generate mutations. Can you elaborate on how this works?

When introducing the edit operations, we now provide details regarding their parameters and their values. Values for each parameter are bounded by the transformation (list of rules and of elements in the rules), the meta-models (defined types) and the available operations in ATL (as defined in the documentation). In this way, we now indicate the specific dimensions (which remain relatively small) the algorithm has to explore to instantiate edit operations and generate mutations.

> **Section 2.3 - Repair Patches** All these operations take parameters to define on which element of the transformation it should be applied, as well as the modified values when applicable. First, they all have a parameter indicating which rule is to be modified. All edit operations also indicate the

modified element, except for *Type of source pattern element* which applies on the unique source pattern element of the rule. The modified element is one of the `to` part of the rule, and some edit operations (e.g. *Type argument of operation*) can target elements in the `to` part as well. *Binding deletion* has a parameter whose value is taken in the list of bindings of the corresponding rule and element. All edit operations which are not creating or deleting bindings modify a datatype, a feature name or an operation call: they have thus two additional parameters indicating the old and the new values. The new values are bounded by the kind of edit operations. *Type of source* and *target pattern elements* take their values in the list of types defined in the source and target meta-models, respectively. *Type argument of operation* can take values from types defined in both meta-models. *Type of collection* relies on the collection types defined in OCL. *Navigation expression* and *Target of binding* take a new value in the attributes of the indicated element. The last three edit operations modify operation calls, which depend on the datatype on which the call is applied, as presented in the ATL documentation[a]. For simplicity, we refer to these three edit operations as *Operation call*.

[a] `https://eclipse.dev/atl/documentation/`

Comment 2 *In section 3.2.1, the authors write, "if the obtained model is equivalent to the expected output...". At this point of the paper, it is necessary to define the notion of equivalence. Later in the paper, it becomes clear that EMFCompare is exploited, and the equivalence notion is related to the number of differences calculated by EMFCompare.*

We have clarified the notion of equivalency in the paper, and moved the paragraph explaining the use of EMFCompare to directly after the paragraph in question.

Then, the input model of the test case is given to the patched transformation to obtain an output model. If the model produced by the transformation is *equivalent* to the expected output, then the test case passes. If the obtained model is different from the expected one, the test fails.
For this objective, we compare the output models generated by the patched transformation with the expected ones provided by the test cases to detect equivalency. We rely on EMFCompare, a tool which, given two models, outputs a list of differences between them, in a similar manner than the differences presented in Fig. 3. An optimal patch is a patch where the generated and test models are *equivalent*. That is, the number of differences between them is zero. When such a patch is found, the process stops. Otherwise, as the number of differences grows, the patch is considered less fit and it thus receives a poor fitness score.

Comment 3 *As discussed in Sect. 4.1 defying fitness functions is a critical task that needs to be done carefully. Considering such complexity, the paper should present a qualitative discussion to elaborate on the extent of the approach being more convenient than fixing transformations by hand. Related to the previous points, what can the authors say about the generalizability/reusability of the edit operations and fitness functions? Can they be defined in a knowledge base and considered to fix any ATL transformations potentially?*

Repairing model transformations and ATL transformations is a tedious task, as we mention in the introduction. We consider this common knowledge in the field, as there have been many efforts to reduce this burden [Cuadrado2018, Troya2022]. This paper presents our investigation of another automated approach to provide patches for the user to examine and apply.

[Cuadrado2018] Cuadrado, J. S., Guerra, E., & de Lara, J. (2018, May). AnATLyzer: an advanced IDE for ATL model transformations. In Proceedings of the 40th international conference on software engineering: Companion proceeedings (pp. 85-88).

[Troya2022] Troya, J., Segura, S., Burgueño, L., & Wimmer, M. (2022). Model transformation testing and debugging: A survey. ACM Computing Surveys, 55(4), 1-39.

We thank the reviewer for this comment about the re-usability of the approach. We have added a paragraph on this topic:

> **Section 6.1 - Approach Benefits**
> Our approach is also applicable to a broad range of ATL transformations. In particular, the fitness metrics defined here, the patch representation, and our use of a diversity score can be reused for other genetic approaches on ATL transformations. As the edit operations (Sec. 3.1) have been defined in prior work as primitive ATL edit operations[], they are also broadly applicable in search-based ATL approaches. To apply our approach to a new ATL transformation, the user must only produce an appropriate test suite of input and output models, possibly assisted from techniques from the literature[].

Comment 4 *Figure 10 comes too late in the paper. I would move it earlier by making graphically explicit what components are new or changed with respect to the conference version of the paper.*

Fig. 10 (now Fig. 9) has been moved early in the paper, to present an overview of the previous approach in Section 3. The extensions presented in this paper are highlighted in the figure and detailed in Section 4.

Comment 5 *The description of Table 3 (second paragraph of pag. 15 of the paper) should be fixed as follows:*

- ... their arguments .... -> ... their argument types?
- "operation call" should be mentioned in the text.

We fixed "their arguments" by "their argument types". We added a sentence about "operation call" earlier in the paper (when presenting the different edit operations) to explain which edit operations were regrouped under this term.

Comment 6 *In the last paragraph of section 5.2, the authors mention that comparing the patched transformations with their correct versions is performed manually. If I'm not wrong, such a comparison is done by EMFCompare, isn't it?*

We have clarified in the text that this comparison is on the transformation code, not the models output by the transformation:

> To answer RQ3, we applied the obtained best patches on the faulty transformations. We then manually compared the patched transformation code with their correct versions to count and classify the remaining errors.

Comment 7 *At the beginning of section 6.1, the authors write that the approach aims to improve the *semi-automatic* repair of model transformations. Thus, the system is not fully automated as readers may understand by looking at Fig. 10. I suggest refining the figure and the corresponding text by clearly showing the automated phases and those that instead require human intervention.*

The text describing the Figure has been refined as follows:

All steps of the approach for generating repair patches as presented in Fig. 9 are fully automated. Selecting and applying the generated patches to perform the repair (after step 6) is a task that requires the experts' intervention.

Review 2

Comment 8 *Listing 1 has a poor layout (different font sizes and types, broken lines), which complicates understanding, but – please correct me if I'm wrong – I'd say it has a few problems, in connection to the meta-models in Fig 1:*

1. It has small syntactic mistakes, like "multiValued" instead of "multivalued"

2. Line 28 "col <- Sequence id , foreignKey)" refers to two missing variables, id and foreignKey

3. It produces wrong target models, since, according to the ER meta-model in Fig 1., a Column object needs to be linked to exactly one Type object. However, rules creating Columns do not create or link these objects to Type objects.

Fig 2 is too small. The PKs of Family and Person is "objId", but they should rather be "FamilyId" and "PersonId". Same with Fig 3.

We thank the reviewer for these detailed comments. We have fixed the layout, syntactic mistakes and the names of the missing variables on line 28.

Because we omitted elements related to Type in our example for the sake of brevity, we removed the class Type from the ER meta-model in Fig. 1.

We fixed the size of Fig. 2 and the transformation of listing 1 so the PKs of Family and Person are "objId".

Comment 9 *The authors use full fledged input-output examples as test cases, which are heavyweight, tedious to create and error prone. Why not using partial oracles, like Tracts [A] or Pamomo patterns [B]?*

[A] Burgueno et al: TractsTool: Testing Model Transformations based on Contracts. MoDELS (Demos/Posters/StudentResearch) 2013: 76-80

[B] Guerra et al: Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. 20(1): 5-46 (2013)

We thank the reviewer for this interesting insight. We have added these two references to the related work, along with a brief discussion of the partial oracle approach compared to our own.

Section 7.2 - Transformation Error Detection and Repair While these partial oracle approaches would reduce the size of the test suite to be created and maintained, it is unclear how to integrate these approaches with our own. This is due to the genetic algorithm underlying our approach, which requires a granular fitness function to guide the patches towards the correct ones. In the current approach, this fitness function is based on the number of model differences between the transformed model and the oracle model (Section 3.2.1). We thus currently require full models in the test suite to provide a fitness value. Future work will determine if partial oracles are sufficiently sensitive enough to guide the search.

Comment 10 *- Fig 4, third edit of the patch: Where is foreignKey in rule MultiValued...?*

This has been fixed with our response to Comment 8, as foreignKey was one of the missing variables.

Comment 11 *Eqn 2 looks odd to me. The current formulation (∀ i...) inside the set would return a boolean value, which is not what you want (probably, something like diff*$(tr(in_i), out_i)|(in_i, out_i) \in Tr)$

We have reformulated Eq. 2 as follows: $\text{errors}(T_{tr}) = \{\text{diff}(t_r(\text{in}_i), \text{out}_i)|i \in |T_{tr}|\}$

Comment 12 *Sec 3.2, paragraph after Fig 7: Explain better what is* O_i*. The second part of the condition for dominance (∃ j...) seems to be implied by the first part (∀i...) if the set of objectives is not empty. Is this really the condition you want to express?*

We have added a sentence to introduce O_i:

> Let us consider a set of m objectives $\{O_i, i \in \{1, 2, ..., m\}\}$ and their corresponding fitness functions $O_i(s)$ mapping a solution s to a value, usually between 0 and 1, reflecting how well the solution s meets the objective O_i.

The first part of the definition of dominance also had a typo: we corrected $>$ to $\geq$.

Comment 13 *What's the time it took to find an optimal patch? This data is necessary to understand if the approach is practical.*

Our evaluation specifically centers on measuring the improvements in terms of efficacy and efficiency resulting from the inclusion of social diversity into our EA repair approach. The focus here is on the running of the search strategy rather than the usability of the approach by end-users. We acknowledge that exploring the latter is a crucial future work, but we consider it currently beyond the scope of this paper's contribution. Additionally, providing specific time metrics at this stage could be misleading, as the time to identify a set of potential solutions strongly depends on the implementation and the computing resources utilized in the search process. We made sure to clearly state the goal of our evaluation and reformulated the beginning of the corresponding section:

> **Section 5:**
> This section reports on the evaluation of the impact of our extended multi-objective approach leveraging social diversity on correcting several semantic errors. More specifically, we aim at determining whether the proposed approach successfully addresses the convergence issues detailed earlier which hindered the generation of patches correcting several errors. This evaluation does not focus on the practicality of the approach in a real-world context, but instead explores the benefits of introducing mechanisms preserving social diversity on the efficacy and efficiency on our search-based repair approaches. We implemented our approach in a tool called *Automatix*, and 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:

Comment 14 *What do patched transformations look like? Do patched look like those that would be produced by human developers? Please show a few examples. I think the patched transformations are not available in the github repository.*

Listing 1 shows the transformation before and after the patch. In the commented-out code, represented by – and the green colouring, this is the code with the defects. The code following those comments is the patched version.

We have added examples of patched transformations to the GitHub repository[1].

[1] https://github.com/jgalasso/faulty-ATL-transformations

Comment 15 *Sec 6.3: Using 4 test cases per transformation looks like a weak test suite, which may really invalidate the results. Did you use mutation analysis to understand the quality of the test cases?*

Indeed, the Section 5.1 Dataset of our experiments lacked a clear explanation regarding the aquisition of the test suites. We added a paragraph at the end of Section 5.1 (Dataset) explaining how we sourced the test suites for each transformation.

> **End of Section 5.1:**
> To generate patches for repairing faulty transformations, we need test cases in the form of correct input / output models. We opted to reused the four test cases for Class2Rel and four test cases for PNML2PN from prior research work. Each test suite comprises the example input model available in ATL Zoo, along with three additional input models sourced from online tutorials. For UML2ER and Bibtex2Docbook, we leveraged four input models sourced from Guerra et al.'s mutation testing approach. To obtain the expected output models, we executed each input model with the correct version of the transformation. Manual verification ensured that each test suite covered all rules associated with its respective transformation.

Comment 16 *Another threat would be that you only used 4 transformation as a base. What was the criterion for choosing them?*

We thank the reviewer for this comment. We have added sentences to the evaluation and threats sections on why we chose to re-use these transformations:

> **Section 5.1 - Dataset** We performed our evaluation on existing faulty transformations from the literature. This was done to aid comparisons to earlier works, and to reuse faulty transformations and test models.

> **Section 6.3 - Threats to Validity**
> We also aimed to reuse the transformations and test suites used in earlier transformation verification work to both aid comparisons between approaches, and to leverage the existence of the transformations, their faulty versions, and the test suites.

Comment 17 *Are the transformation you cannot fix? You mention you do not patch helpers, but do you patch OCL expressions (which may appear in bindings or rule filters)? Choosing only mutants for which you can produce patches would probably be another threat.*

We thank the reviewer for mentioning this limitation and threat. We have now added sentences to specify this.

> **Section 6.2 - Approach Limitations**
> A limitation of our approach is that we are fixing ATL transformation rules only, not the helpers. This will require the definition of further edit operations that can modify helpers in ATL transformations. We also do not consider all constructs of OCL, restricting the range of errors that our approach can fix.
> **Section 6.3 - Threats to Validity**
> 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. However, we also note that the mutations we have selected do not cover possible errors in all possible constructs of OCL. Thus, our approach will not be able to repair these transformations.

Comment 18 *Are 5 errors a hard limit for your approach? Can you go beyond?*

We have added a paragraph in the threats to validity to explain that the choice of five is not a hard limit:

Section 6.3 - Threats to Validity
We have also only tested our approach on transformations containing up to five errors. This number was selected to improve upon the results of our earlier work[], which suffered poor performance after three semantic errors. While we claim our approach can effectively find patches for a transformation with five errors, this is not a hard limit. We expect that adding more errors in the transformation would increase the difficulty for the genetic algorithm to find the correct patch. Our approach would thus require more time and computation to fix an increasing number of errors.

Comment 19 *Related work: I miss here MDE-based evolutionary approaches, like MoMoT [C], or the approach in [D]*

[C] Bill et al: A local and global tour on MOMoT. Softw. Syst. Model. 18(2): 1017-1046 (2019)

[D] Burdusel et al. Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering. Softw. Syst. Model. 20(6): 1857-1887 (2021)

We thank the reviewer for mentioning these works.

While the MOMoT tool is certainly an example of applying search-based techniques to model transformations, our understanding is that MOMoT is targeted towards the *scheduling* of the transformation rules to satisfy the optimization criteria. As we are studying the *repair* of the transformation rules themselves, we (kindly) consider the MOMoT paper to be too distantly related to fit within our related work categories.

We have added a paragraph relating the work of Burdusel *et al.* to the set of ATL mutations used in this work:

Section 6.2 - Limitations
Our current set of mutation operators is at the level of modifying ATL primitives, based on previous literature. The efficiency of the approach would be improved by also taking the meta-models of the transformation into account. For example, Burdusel *et al.* define *multiplicity-preserving search operators* (MPSOs) which combine multiple edit operations[]. When these MPSOs are applied instead of the primitive edit operations, the resulting model will always conform to the meta-model, ensuring that only correct solutions are produced.

Comment 20 *Related work: How does your approach compare with EA approaches for GPLs? Are techniques similar? Are they equally effective?*

We do acknowledge that our approach shares similarities with EA approaches for GPLs. However, we (respectfully) consider a direct comparison of EAs for GPLs to be out-of-scope for the current work. That said, we have added some text in Section 3.1.1 - Solution Representation relating model transformation EAs to GPL EAs:

> 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 [Weimer2009]. 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 [...]

[Weimer 2009] Weimer W, Nguyen T, Le Goues C, et al (2009) Automatically finding patches using genetic programming. In: 2009 IEEE 31st International Conference on Software Engineering, IEEE, pp 364–374

Comment 21 *Minors/Typos*

- Abstract: Are 5 errors "many"?

- Section 1: 2nd par of "play an an", "meta-model. by producing"

- Sec 2.1: I miss an explanation of how ATL's binding resolution mechanism for bindings works (for example, as you use in line 10)

- Sec 2.2 "inputs-outputs example" -> "input-output examples"

- Sec 2.3 "edit operations which modifies" -> modify

- Eqn (1) has an extra ","

- Sec 3.2, after Fig 7: You mention N/2 without mentioning what is N at this point.

- The equations with arg min (4, 5, 9) are a bit confusing: what is f_1? what is f_2? A perhaps more clear notation would be arg $min_p(|error(T_{tr_p})|)$ and so on.

- Sec 4.1.1. "sift" -> "shift"

- Eqn 8: $r_{e_i} -> r(e_i)$

- Sec 5.2. "cut-off.If"

- The colours of Figs 11 – 13 look similar in B/W. I'd suggest to use colours + patterns to make the distinction clearer.

- Related work: Is really the tool proposed in [9] called "Quick fix"? That seems rather a general term to refer to an automated recipy for fixing a type of error.

- Listing 1 uses different font sizes

We thank the reviewer for identifying these issues. They have been corrected.
- We have changed "many" for "multiple" in the abstract.
- In eq 8 (and the text above), we have changed r_{e_i} by $r(e_i)$.
- We added an explanation for the binding resolution mechanism:

> When a binding references elements from the source model to initialize an element of the target model, the source model elements must undergo a transformation into elements compatible with the target model. In such cases, a binding resolution mechanism comes into play, tasked with identifying a rule capable of executing this transformation. This involves finding a rule with a `from` part corresponding to the type of the source model element and a `to` part corresponding to the type of the target model element.

- Eq 4, 5, and 9 now define the score returned by the objectives:
$O_1(p) = \left|\text{errors}(T_{tr_p})\right|$ (to be minimized)
$O_2(p) = |p|$ (to be minimized)
$O_3(p) = s(p)$ (to be maximized)
The 3 equations were rewritten accordingly.

Review 3

Comment 22 *My only concern about the structure is the the presentation of the third objective, Social Diversity, in section 4.2, instead of along with the other objectives. I didn't understand why diversity was also presented as a crowding distance. I have the impression that it only brings confusion to the paper.*

We reworked Sections 3 and 4 such that it is clearer why the 3rd objective is presented in a separate section. Section 3 introduces the previous approach which used 2 objectives. Section 4 presents an analysis of the convergence issues of this previous approach, discusses the notion of semantic diversity and its potential to overcome the identified issues, and finally formalizes how to compute a semantic diversity score, which can be leveraged both as a 3rd objective and as a crowding distance. The new approach is an extension which promotes semantic diversity and is evaluated in Section 5.

In Section 1: In Section 2, we describe ATL transformations, provide examples of defects and of patches repairing such defects. Section 3 presents our previous approach relying on EAs to find patches repairing semantic errors in transformations. We show how we formalized patch generation as an optimization problem and how we adapted multi-objective EAs to find patches meeting several objectives. Then, Section 4 presents an analysis of the limits of the previous approach and why we believe that introducing diversity would help mitigate these issues. We present an updated version of our approach from [47], now including a social diversity measure in the process. Our new approach is evaluated in Section 5 ...

First paragraph of Section 3: This section introduces an approach presented in our previous work [41] which relies on *evolutionary algorithms* (EAs) to automatically find repair patches for ATL transformations having semantic errors. Section 3.1 first presents the important concepts of EAs (i.e., solution representation, genetic operators and fitness function) and how we adapted them to find patches automatically. A key aspect of our previous approach was to find patches optimizing two different objectives: Section 3.2 then presents *multi-objective EAs* which enable to exploit several fitness functions. We explain the implementation of the two fitness functions we used to find patches repairing the most errors possible while preventing them from growing unnecessarily large. Finally, Section 3.3 combines the presented concepts to provide an overview of the approach.

Transition paragraph in Section 3: To sum up, we used EAs to automatically evolve a population of patches over several generations until finding optimal ones. We discussed a strategy to assess generated patches based on the objective of repairing a transformation. However, a key aspect of our automated repair approach [41] was to find patches meeting several objectives.

First paragraph of Section 4: The approach presented in Section 3 faced difficulties to find good patches when the faulty transformation presented 2 errors or more [41]. In this section, we propose an extension of our previous approach which improves both its efficacy and efficiency. We first identify and discuss two convergence issues faced by the previous approach, namely *single fitness peak* and *fitness plateaus* in Section 4.1. Then, Section 4.2 introduces the notion of social diversity and our hypothesis that maintaining diversity in the population of our approach could help with the aforementioned issues. Finally, we propose two ways to integrate social diversity in our approach, i.e., in the form of a third objective or as a crowding distance, in Section 4.3.

Comment 23 *First, I wonder if the authors considered using performance as an objective of the NSGA-II algorithm, since poorly written OCL/ATL expressions may have an important impact on bigger transformations.*

We thank the reviewer for this comment. We have not (yet) considered transformation performance as an objective. As mentioned below in our response to Comment 27, our focus is on *repairing* the transformation, and not on *improving quality* or *optimization*. We have added precision to our future work on this:

> One aspect of our future work is to combine the social diversity measure with objectives focusing on other quality attributes, such as transformation execution time or reducing overall complexity[Wimmer2012, Alkhazi2020].

[Wimmer2012] Wimmer et al. "A catalogue of refactorings for model-to-model transformations"
[Alkhazi] Alkhazi et al. "On the value of quality attributes for refactoring ATL model transformations: A multi-objective approach"

Comment 24 *Second, I wonder if it is possible to reduce the "area" of code that can be patched, excluding for instance, rules that have nothing to do with the detected differences in the output model.*

We thank the reviewer for this insight. We have added a paragraph to the future work section, mentioning related work on slicing transformations:

> Another approach to improve the efficiency of our approach is to 'slice' the transformation to just the rules which have an observable effect on the output model [Cheng2018]. This would restrict patch creation to only relevant rules, improving the convergence of our genetic algorithm approach.

[Cheng2018] Cheng Z, Tisi M (2018) Slicing ATL model transformations for scalable deductive verification and fault localization. International Journal on Software Tools for Technology Transfer 20:645–663

Comment 25 *The ATL Zoo has interesting transformation examples, but they are toy examples that use small input models. I'm not convinced that it reflects real-world transformations and that the good results found can be extrapolated to non-toy transformations.*

Please see our answer to Comment 16.

Comment 26 *Moreover, the experiments do not evaluate two aspects that could prevent the approach's applicability: the execution time and the quality of the patches. More precisely, if the approach takes too much time to provide a patch or if the proposed patch is too complex, the approach's applicability would be reduced.*

Please see our answer to Comment 13 and Comment 27.

Comment 27 *Did you consider asking an ATL expert to validate the proposed patches? Or use ATL quality metrics, similar to the work of Alkhazi et al. [Models 2016 and IEEE Transactions on Software Engineering 2017], who use a search-based approach to suggest ATL code refactorings.*

We thank the reviewer for this comment, and the pointer to these works. For this paper, we consider patches to be optimal if they are minimal, and if they produce the correct output model. Thus, we have not considered the qualitative aspects of these patches, such as preference by an expert.
We have added this reference to the related work section:

Section 7.2 - Transformation Error Detection and Repair
In a broader sense of improving transformations, Alkhazi *et al.* consider optimizing transformations to improve qualities such as rule complexity, cohesion, and coupling [Alkhazi2020]. A genetic algorithm is used, with operators such as moving rules between modules. While similar to our approach, we are instead concerned with the semantic *correctness* of rule elements, not rule refactoring to improve *quality*.

[Alkhazi2020] Alkhazi B, Abid C, Kessentini M, et al (2020) On the value of quality attributes for refactoring ATL model transformations: A multi-objective approach. Information and Software Technology 120:106243

Comment 28 *Could you add hyperlinks to references? It's helpful when reading the PDF version.*

We apologize for the inconvenience. On our machines, the first submitted PDF contained clickable citation markers. We have now explicitly added the `hyperref` package in the preamble, in an attempt to correct this issue for the reviewer.

Improving Repair of Semantic ATL Errors using a Social Diversity Metric

Zahra VaraminyBahnemiry[1], Jessie Galasso[2], Bentley Oakes[3], Houari Sahraoui[1*]

[1]Département d'Informatique et de Recherche Opérationnelle (DIRO), Université de Montréal, 3150 Jean Brillant St, Montréal, H3T 1N8, Quebec, Canada.
[2]Department of Electrical and Computer Engineering (ECE), McGill University, 3480 University Street, Montréal, H3A 0E9, Quebec, Canada.
[3]Département de Génie Informatique et Génie Logiciel (GIGL), Polytechnique Montréal, 2700 Tour Rd, Montréal, H3T 1J4, Quebec, Canada.

*Corresponding author(s). E-mail(s): sahraouh@iro.umontreal.ca;
Contributing authors: varaminz@iro.umontreal.ca; jessie.galasso-carbonnel@mcgill.ca; bentley.oakes@polymtl.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 *multiple 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 by increasing the level of abstraction [28]. In this context, MDE sees models as first-class artifacts where domain-specific modeling languages capture specific aspects of the solution.

Model transformations then play 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. They can also be used to produce from these models low-level artifacts such as source code, documentation, and test suites [39], or to optimize or simulate the model itself [26].

Model transformations can be written with either general-purpose programming languages or in dedicated transformation languages such as DSLTrans [4] or the ATLAS Transformation Language (ATL) [21].

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 [44].

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.* [12] propose predefined corrective patches to be applied on errors detected with syntactic analysis tools (e.g., AnATLyzer [11] 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 [29], as well as both syntactic [48] and semantic [47] 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 a space of 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 [47], 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* [41], 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 [5, 41].

Contributions and Structure

In this paper, we extend our EA-based approach from [47] 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* [5]. 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* [37], where solutions must optimize several

objectives including social diversity. Our approach is implemented using the NSGA-II algorithm, a fast multi-objective EA [13].

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 several 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 describe ATL transformations, provide examples of defects and of patches repairing such defects. Section 3 presents our previous approach [47] relying on EAs to find patches repairing semantic errors in transformations. We show the formalization of patch generation as an optimization problem and how we adapted multi-objective EAs to find patches meeting several objectives. Then, Section 4 presents an analysis of the limits of the previous approach and why we believe that introducing diversity would help mitigate these issues. We present an updated version of our approach from [47], now including a social diversity measure in the process. Our new 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) [21], 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 [21] is a well-known textual transformation language which is studied in the MDE literature [11, 12, 20, 33]. 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.

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* (`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

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

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

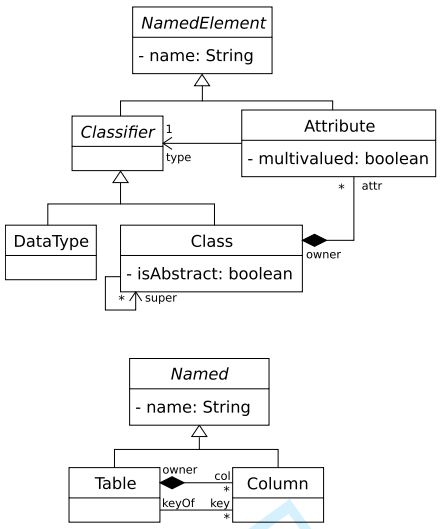


Fig. 1 Excerpt of a UML Class Diagram meta-model (top) and Relational Schema meta-model (bottom)

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). When a binding references elements from the source model to initialize an element of the target model, the source model elements must undergo a transformation into elements compatible with the target model. In such cases, a binding resolution mechanism comes into play, tasked with identifying a rule capable of executing this transformation. This involves finding a rule with a `from` part corresponding to the type of the source model element and a `to` part corresponding to the type of the target model element.

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

```
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 <- "objId")}
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       id: Relational!Column (
30          name <- a.owner.name.firstToLower() + "
            Id"),
31       foreignKey: Relational!Column (
32          -- name <- a.type + "Id")}
33          name <- a.name + "Id")}
```

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

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

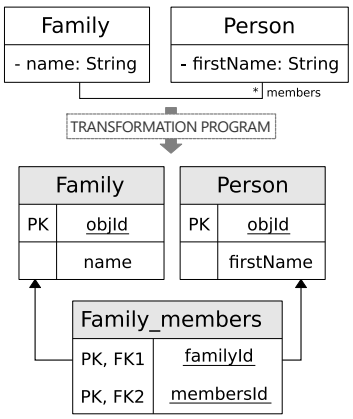
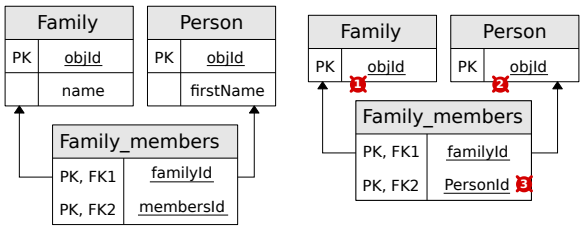


Fig. 2 Relational Schema (output model, bottom) obtained when applying the transformation of Listing 1 to the class diagram (input model, top)

as AnATLyser [11], 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 input-output 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 produced 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 (top) and the expected output model (bottom).

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 [11] 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* [29].

We call a *patch* a sequence of edit operations which modify 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.* [12]. This subset corresponds to the operations modifying elements in the transformation rules, as presented in [47]. 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 operation call* and *Predefined operation call* change a call by another.

All these operations take parameters to define on which element of the transformation it should be applied, as well as the modified values when applicable. First, they all have a parameter indicating which rule is to be modified. All edit operations also indicate the modified element, except for *Type of source pattern element* which applies on the unique source pattern element of the rule. The modified element is one of the `to` part of the rule, and some edit operations (e.g., *Type argument of operation*) can target elements in the `to` part as well. *Binding deletion* has a parameter whose value is taken in the list of bindings of the corresponding rule and element. All edit operations which are not creating or deleting bindings modify a datatype, a feature name or an operation call: they have thus two additional parameters indicating the old and the new values. The new values are bounded by the kind of edit operations. *Type of source* and *target pattern elements* take their values in the list of types defined in the source and target meta-models, respectively. *Type argument of operation* can take values from types defined in both meta-models. *Type of collection* relies on the collection types defined in OCL. *Navigation expression* and *Target of binding* take a new value in the attributes of the indicated element. The last three edit operations modify operation calls, which depend on the datatype on which the call is applied, as presented in the ATL documentation[3]. For simplicity, we refer to these three edit operations as *Operation call*.

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.

[3] https://eclipse.dev/atl/documentation/

Table 1 Atomic edit operations to modify ATL transformation programs, taken from [12, 47].

Target	Type
Binding	Creation
Binding	Deletion
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 operation call	Operation modification

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

[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

as an *optimization problem*, where the goal is to automatically find optimal solutions in a space of 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 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.

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

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 [6] in our implementation.

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

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 introduces an approach presented in our previous work [47] which relies on *evolutionary algorithms* (EAs) to automatically find repair patches for ATL transformations having semantic errors. Section 3.1 first presents the important concepts of EAs (i.e., solution representation, genetic operators and fitness function) and how we adapted them to find patches automatically. A key aspect of our previous approach was to find patches optimizing two different objectives: Section 3.2 then presents *multi-objective EAs* which enable to exploit several fitness functions. We explain the implementation of the two fitness functions we used to find patches repairing the most errors possible while preventing them from growing unnecessarily large. Finally, Section 3.3 combines the presented concepts to provide an overview of the approach.

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 [14, 16] and domain specific ones such as ATL [47, 48]. 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 [49]. 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

[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

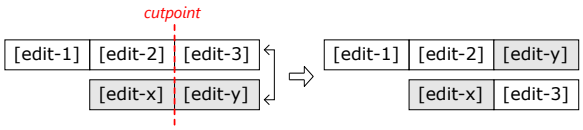


Fig. 6 Examples of a single point crossover operation

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.

In our case, the objective is to generate patches repairing a transformation. 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.

To sum up, we used EAs to automatically evolve a population of patches over several generations until finding optimal ones. We discussed a strategy to assess generated patches based on the objective of repairing a transformation. However, a key aspect of our automated repair approach [47] was to find patches meeting several objectives.

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. Thus, non-dominated solutions are not comparable and can be considered equally good. In this paper, we use NSGA-II [13], a well-known fast multi-objective genetic algorithm suitable for the kind of problem we are solving [1].

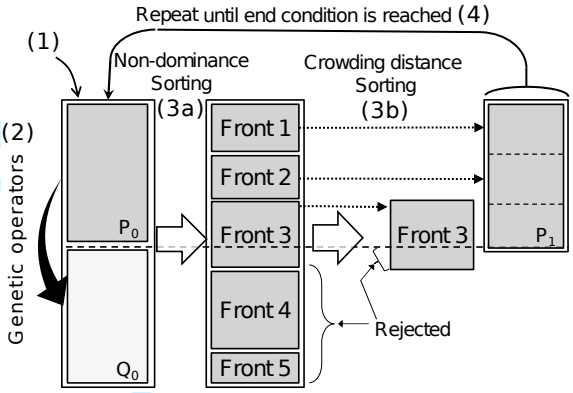


Fig. 7 NSGA-II Algorithm [13]

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)), with N being the size of the population. 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. Then, the resulting population is sorted into dominance fronts according to the dominance principle (3a). Let us consider a set of m objectives

$\{O_i, i \in \{1, 2, ..., m\}\}$ and their corresponding fitness functions $O_i(s)$ mapping a solution s to a value reflecting how well the solution s meets the objective O_i. A solution s_1 dominates a solution s_2 for a set of objectives $\{O_i\}$ if $\forall i, O_i(s_1) \geq O_i(s_2)$ 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 [13] (3b). This process is repeated (4) until an optimal solution is found in the first front or a stop criterion is reached, e.g., a number of iterations or one or more objectives greater than a certain threshold.

In our previous approach [47], we defined two objectives to evaluate our generated patches. We detail those objectives here:

3.2.1 Objective 1: Fixing as Many Errors as Possible

The first objective of our algorithm 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 p, first the sequence of edit operations is applied on the faulty transformation tr to obtain a patched transformation tr_p. Then, the input model of the test case is given to the patched transformation to obtain an output model $tr_p(in_i)$. If the model produced by the transformation is *equivalent* to the expected output out_i, then the test case passes. If the obtained model is different from the expected one, the test fails.

For this objective, we compare the output models generated by the patched transformation with the expected ones provided by the test cases to detect equivalency. We rely on EMFCompare [6], a tool which, given two models, outputs a list of differences between them, in a similar manner than the differences presented in Fig. 3. An optimal patch is a patch where the generated and test models are *equivalent*. That is, the number of differences between them is zero. When such a patch is found, the process stops. Otherwise, as the number of differences grows, the patch is considered less fit and it thus receives a poor fitness score.

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 provides 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 fewer discrepancies between the output models and the expected ones compared to a random solution.

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.

$$O_1(p) = \left|\text{errors}(T_{tr_p})\right| \tag{4}$$

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.

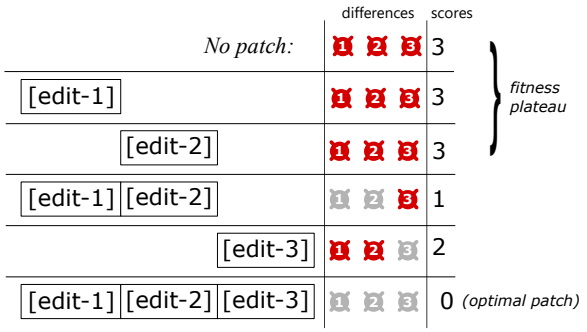


Fig. 8 Assessing patch fitness depending on model differences

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) which represents the number of operations in the patch. This objective, which should be minimized, thus favors patches of small size to avoid generating candidate patches using too many edit operations.

$$O_2(p) = |p| \tag{5}$$

3.3 Approach Overview

Fig. 9 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 five steps. First, a new generation of patches is created by the NSGA-II algorithm, as denoted by the label "1" in Fig. 9. 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

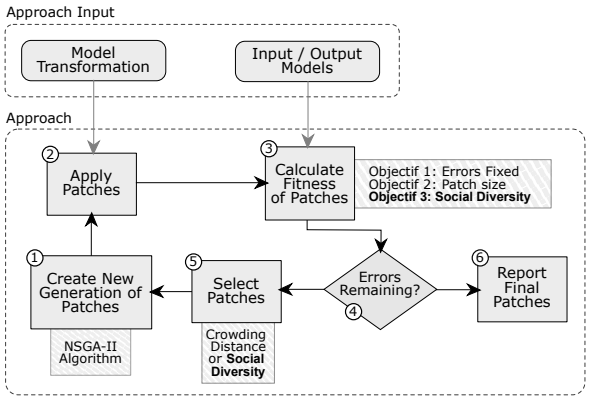


Fig. 9 An overview of our model transformation repair approach.

are tested against the input/output models in step 3 to determine the fitness of the respective patches, as scored by several objectives.

In step 4, it is determined whether the errors in the transformation have been fixed. If errors remain, then the best patches are selected in step 5 and another generation of patches is produced. If no errors remain, then the final set of patches is presented to the user for their inspection (step 6).

The parts about Social Diversity appearing in bold font in Fig. 9 are extensions we added in this paper and are addressed in the next section.

All steps of the approach for generating repair patches as presented in Fig. 9 are fully automated. Selecting and applying the generated patches to perform the repair (after step 6) is a task that requires the experts' intervention.

4 Social Diversity Repair Approach

The approach presented in Section 3 faced difficulties to find good patches when the faulty transformation presented 2 errors or more [47]. In this section, we propose an extension of our previous approach which improves both its efficacy and efficiency. We first identify and discuss two convergence issues faced by the previous approach, namely *single fitness peak* and *fitness plateaus* in Section 4.1. Then, Section 4.2 introduces the notion of social diversity and our hypothesis that maintaining diversity in the population of our approach could help with the aforementioned issues. Finally, we propose two ways to integrate

social diversity in our approach, i.e., in the form of a third objective or as a crowding distance, in Section 4.3.

4.1 Issues with Convergence

In our previous work [47], we used EAs to repair transformations by relying on 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) which are essential to build optimal patches were quickly discarded in the process, because the fitness function fail to properly reflect their value.

For a given problem, different fitness functions can be designed to achieve the same objective. 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. Having to rely 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 filter and retain 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 10.

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 [14, 41]. 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 approach, 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 disturb similar parts of the output models, correcting only one of them

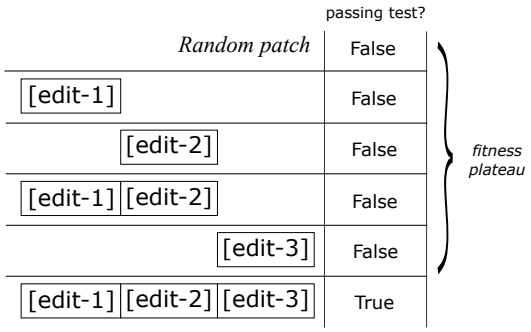


Fig. 10 Example of fitness plateau caused by a fitness evaluation based on passing test cases

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. Using model differences rather that number of passing test cases helps reduce fitness plateaus because it provides a more fine-grained score. In Fig. 8, the scores of the same patches as Fig. 10 creates a smaller plateau.

The more errors to correct in a transformation, the larger the size of the plateaus and the less effective the search for an optimal patch. This explains why our approach faced limitations with transformations having several errors.

4.2 Social Semantic Diversity

Our hypothesis in this paper is that deliberately maintaining diversity in the population would not only help avoiding single fitness peak but also reducing fitness plateaus, hence increasing the effectiveness and efficiency of test-based EAs approaches.

4.2.1 Definitions

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 [27]. We thus focus on semantic diversity, which is also known to be more efficient to prevent single fitness peak [5, 46].

4.2.2 Social Diversity for Repair Patches

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 [5] 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 and reducing premature convergence of single fitness peak. 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 Objective 3: Preserving Semantic Diversity

We propose to add a third objective to address the above convergence issues 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 [47].

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[4].

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

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.

[4]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.

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

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 O_3 is then defined in Equation 9 where we select for the highest score.

$$O_3(p) = s(p) \quad (9)$$

Crowding distance: In multi-objective EAs, when the best solutions of a generation are selected to start the next generation, it may be necessary to select a sub-set of solutions from a front which is, by definition, constituted of non comparable solutions. In this case, a crowding distance is used to select solutions from a front while favoring diversity. Given a set of patches from the same front, the proposed semantic diversity score can be reused as a crowding distance as well.

5 Experiments

This section reports on the evaluation of the impact of our extended multi-objective approach leveraging social diversity on correcting several semantic errors. More specifically, we aim at determining whether the proposed approach successfully addresses the convergence issues detailed earlier which hindered the generation of patches correcting several errors. This evaluation does not focus on the practicality of the approach in a real-world context, but instead explores the benefits of introducing mechanisms preserving social diversity on the efficacy and efficiency on our search-based repair approaches. We implemented our approach in a tool called *Automatix*, and perform our evaluation on four existing third-party transformations, where two were examined in our earlier work [47]: *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. This was done to aid comparisons to earlier works, and to reuse faulty transformations and test models. In previous work [47], 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[5]. Guerra *et al.* [20] 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 [11], 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 [48] 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

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

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 argument types, 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 nine 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 [48] 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, such as the number of rules and the composition of their input and output metamodels.

For comparison, we have examined the size (in terms of number of rules) of the 106 ATL transformations from the ATL Zoo and recorded that these transformations have an average of 11 rules, with $Q1 = 5$, $Q3 = 12$ and the median being 9. The four selected transformations of our evaluation (with 8, 5, 9, and 8 rules) are thus representative of transformations found in the ATL Zoo.

To generate patches for repairing faulty transformations, we need test cases in the form of correct input / output models. We opted to reused the four test cases for Class2Rel and four test cases for PNML2PN from prior research work [47]. Each test suite comprises the example input model available in ATL Zoo, along with three additional input models sourced from online tutorials. For UML2ER and Bibtex2Docbook, we leveraged four input models sourced from Guerra et al.'s mutation testing approach. To obtain the expected output models, we executed each input model with the correct version of the transformation. Manual verification ensured that each test suite covered all rules associated with its respective transformation.

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 [47], 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,

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

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 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. We then manually compared the patched transformation code with their correct versions to count and classify the remaining errors.

5.3 Results

This section will examine each of our three research questions and discuss the results.

5.3.1 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 [47], *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

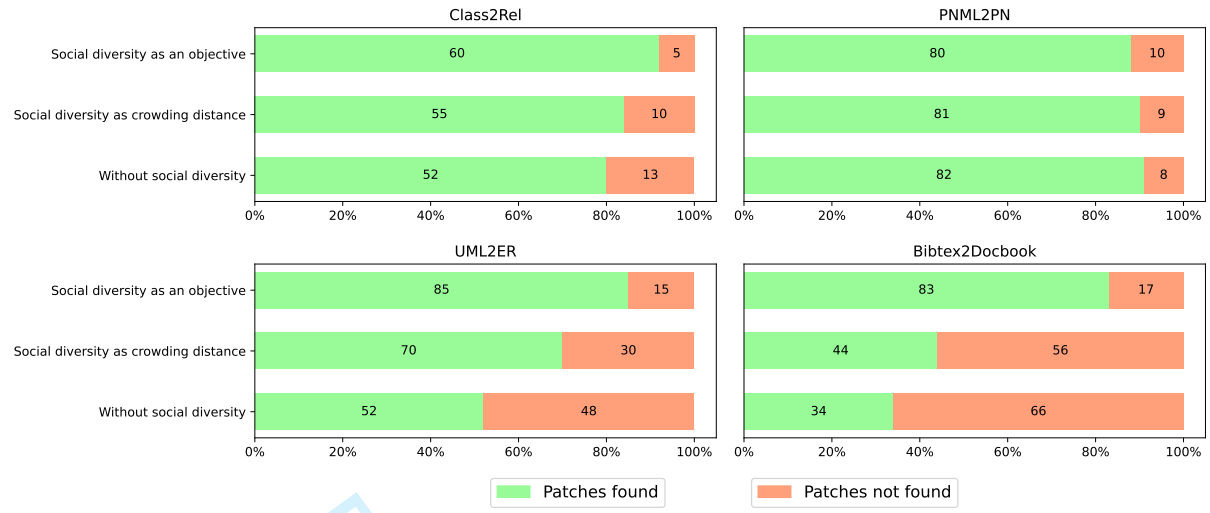


Fig. 11 Percentage of runs finding an optimal patch (RQ1)

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**.

5.3.2 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**.

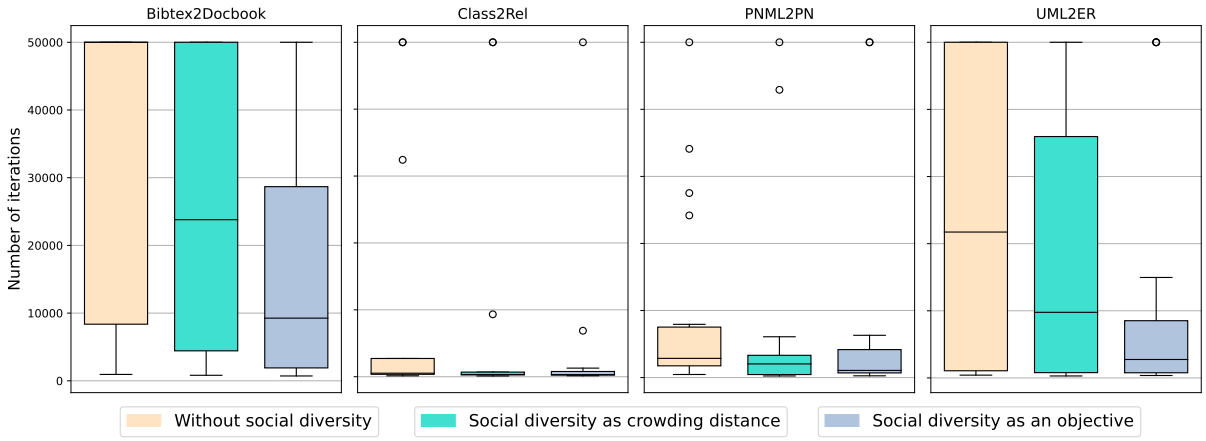


Fig. 12 Number of generations (convergence time) to find a solution

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

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

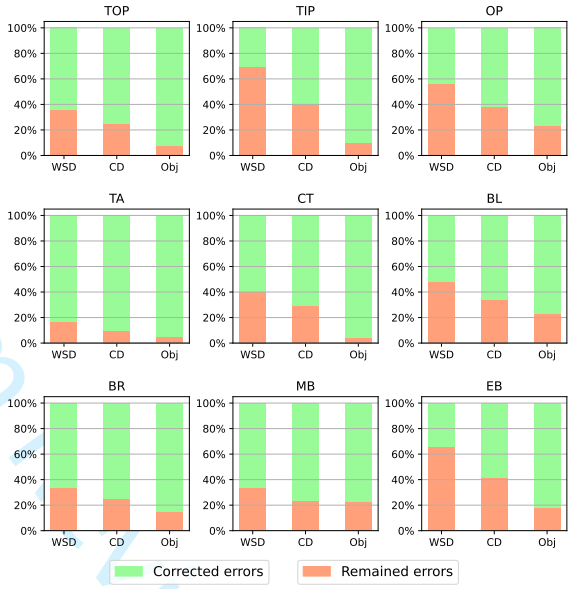


Fig. 13 Percentage of corrected/remained errors for each type of semantic error in faulty transformations, without social diversity (WSD), with social diversity as crowding distance (CD) and with social diversity as an objective (Obj).

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 thus assists with fighting 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.

Our approach is also applicable to a broad range of ATL transformations. In particular, the fitness metrics defined here, the patch representation, and our use of a diversity score can be reused for other genetic approaches on ATL transformations. As the edit operations (Sec. 3.1) have been defined in prior work as primitive ATL edit operations [12, 47], they are also broadly applicable in search-based ATL approaches. To apply our approach to a new ATL transformation, the user must only produce an appropriate test suite of input and output models, possibly assisted from techniques from the literature [20].

6.2 Approach Limitations

A limitation of our approach is that we are fixing ATL transformation rules only, not the helpers. This will require the definition of further edit operations that can modify helpers in ATL transformations. We also do not consider all constructs of OCL, restricting the range of errors that our approach can fix.

Our current set of mutation operators is at the level of modifying ATL primitives, based on previous literature. The efficiency of the approach would be improved by also taking the meta-models of the transformation into account. For example, Burdusel *et al.* define *multiplicity-preserving search operators* (MPSOs) which combine multiple edit operations [7]. When these MPSOs are applied instead of the primitive edit operations, the resulting model will always conform to the meta-model, ensuring that only correct solutions are produced.

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. However, we also note that the mutations we have selected do not cover possible errors in all possible constructs of OCL. Thus, our approach will not be able to repair these transformations.

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. We also aimed to reuse the transformations and test suites used in earlier transformation verification work to both aid comparisons between approaches, and to leverage the existence of the transformations, their faulty versions, and the test suites.

We have also only tested our approach on transformations containing up to five errors. This number was selected to improve upon the results of our earlier work [47], which suffered poor performance after three semantic errors. While we claim our approach can effectively find patches for a transformation with five errors, this is not a hard limit. We expect that adding more errors in the transformation would increase the difficulty for the genetic algorithm to find the correct patch. Our approach would thus require more time and computation to fix an increasing number of errors.

7 Related Work

The work presented in this paper intersects three research areas: *model repair*, *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.* [44].

7.1 Model Repair

Ben Fadhel et al. [15] 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. [36] 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. [42], 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. [32]. Their approach repairs models in a specific context but the efficiency of evolutionary algorithms, in which we used in our approach, is independent from the context. In [25], 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. [3] 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

Oakes et al. [33] presented an approach to statically verify the declarative subset of ATL model transformations. They translated the ATL transformation into DSLTrans using a higher-order transformation. Due to the limited expressiveness of DSLTrans, a symbolic-execution approach can then produce representations of all possible executions to the transformation. The transformation is verified through the matching of pre-/post-condition *contracts* on these representations, which resemble the visual contract language of Guerra *et al.* [19]. This produces evidence that the transformation is working correctly.

Similar work by Vallecillo *et al.* describes the definition of *Tracts* to be specified on model transformations [45]. Tracts define sets of constraints i) on the source and target meta-models, and ii) on the source-target constraints. Tracts also define a test suite, a collection of source models satisfying the source constraints. A TractsTool can then automatically transform source models into the target meta-model, and verify that the source/target model pairs satisfy the constraints.

While these partial oracle approaches would reduce the size of the test suite to be created and maintained, it is unclear how to integrate these approaches with our own. This is due to the genetic algorithm underlying our approach, which requires a granular fitness function to guide the patches towards the correct ones. In the current approach, this fitness function is based on the number of model differences between the transformed model and the oracle model (Section 3.2.1). We thus currently require full models in the test suite to provide a fitness value. Future work will determine if partial oracles are sufficiently sensitive enough to guide the search.

Troya et al. [43] proposed a Spectrum-Based Fault Localization (SBFL) technique, which uses test cases to find the probability of transformation rules being faulty. This symbolic execution approach was then combined with the SBFL technique in [34]. These works focus on detecting faulty rules in transformation programs and do not propose rule patches or repair the faulty rules.

Burgueño et al. [8] 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. [10] 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. [12] proposed quick fixes to repair syntactic errors in ATL transformations using a static analyzer proposed in [10]. Their approach needs a user interaction to select a suitable repair, while our approach generates a candidate patch automatically. In a previous work [48], we relied on the static analyzer of [10] to automatically generate patches addressing syntactic errors in transformation programs.

Kessentini et al. [22] 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 [38]. 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.

In a broader sense of improving transformations, Alkhazi *et al.* consider optimizing transformations to improve qualities such as rule complexity, cohesion, and coupling [2]. A genetic algorithm is used, with operators such as moving rules between modules. While similar to our approach, we are instead concerned with the semantic *correctness* of rule elements, not rule refactoring to improve *quality*.

7.3 Social Diversity

Soto [40] 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. [14] 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. [46] divided semantic-aware methods into three categories. *Diversity methods*, that work with diversity, mostly at the population level [23]. *Indirect semantic methods*, that act on the syntax of the individuals and depend on criteria to indirectly promote a semantic behavior [35] [17] [18] [24] . *Direct semantic methods*, that act directly on the semantics of the individuals by using precise genetic operators [30]. All these approaches improve the power of genetic programming. Batot et al. [5] 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 present our 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.

One aspect of our future work is to combine the social diversity measure with objectives focusing on other quality attributes, such as transformation execution time or reducing overall complexity [1, 50]. 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* [31, 34, 43]. 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.

Another approach to improve the efficiency of our approach is to 'slice' the transformation to just the rules which have an observable effect on the output model [9]. This would restrict patch creation to only relevant rules, improving the convergence of our genetic algorithm approach.

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] Alkhazi B, Abid C, Kessentini M, et al (2020) On the value of quality attributes for refactoring ATL model transformations: A multi-objective approach. Information and Software Technology 120:106243

[3] 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

[4] 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

[5] 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

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

[7] Burdusel A, Zschaler S, John S (2021) Automatic generation of atomic multiplicity-preserving search operators for search-based model engineering. Software and Systems Modeling 20(6):1857–1887

[8] 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

[9] Cheng Z, Tisi M (2018) Slicing ATL model transformations for scalable deductive verification and fault localization. International Journal on Software Tools for Technology Transfer 20:645–663

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] Guerra E, de Lara J, Wimmer M, et al (2013) Automated verification of model transformations based on visual contracts. Automated Software Engineering 20(1):5–46

[20] 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

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

[22] 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

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

[24] 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

[25] 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

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

[27] 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

[28] 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

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

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

[31] 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)

[32] 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

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

[34] Oakes BJ, Troya J, Galasso J, et al (2023) Fault localization in DSLTrans model transformations by combining symbolic execution and spectrum-based analysis. Software and Systems Modeling https://doi.org/10.1007/s10270-023-01123-3

[35] 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

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

[37] Ramirez A, Romero JR, Ventura S (2019) A survey of many-objective optimisation in search-based software engineering. Journal of Systems and Software 149:382–395

[38] 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

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

[40] 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

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

[42] 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

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

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

[45] Vallecillo A, Gogolla M, Burgueno L, et al (2012) Formal specification and testing of model transformations. In: International School on Formal Methods for the Design of Computer, Communication and Software Systems. Springer, p 399–437

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

[47] 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). IEEE, pp 13–23, https://doi.org/10.1109/MODELS50736.2021.00011

[48] 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)

[49] Weimer W, Nguyen T, Le Goues C, et al (2009) Automatically finding patches using genetic programming. In: 2009 IEEE 31st International Conference on Software Engineering, IEEE, pp 364–374

[50] 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