AnATLyzer: An Advanced IDE for ATL Model Transformations

Jesús Sánchez Cuadrado
Universidad de Murcia
jesusc@um.es

Esther Guerra
Universidad Autónoma de Madrid
esther.guerra@uam.es

Juan de Lara
Universidad Autónoma de Madrid
juan.delara@uam.es

ABSTRACT

Model transformations (MTs) are key in model-driven engineering as they automate model manipulation. Their early verification is essential because a bug in a MT may affect many projects using it. Still, there is a lack of analysis tools applicable to non-toy transformations developed with practical MT languages.

To alleviate this problem, this paper presents AnATLyzer: a static analysis tool for ATL MTs. The tool is able to detect a wide range of non-trivial problems in ATL transformations by using constraint solving to improve the analysis precision. It provides a live environment integrated into Eclipse which allows checking and fixing problems as the transformation is written. The environment is highly configurable and provides facilities like quick fixes, visualizations, navigation shortcuts and problem explanations. We have evaluated the tool over third-party MTs, obtaining good results.

The tool website is http://anatlyzer.github.io, and a video showcasing its features is at https://youtu.be/bFpbZht7bqY

CCS CONCEPTS

• **Software and its engineering → Software development techniques**; **Software verification and validation**;

KEYWORDS

Model-driven engineering, Model transformation, Static analysis, Verification, ATL

ACM Reference Format:
Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2018. AnATLyzer: An Advanced IDE for ATL Model Transformations. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3183440.3183479

1 INTRODUCTION

Model transformation (MT) is the main enabler of automation in model-driven engineering (MDE) [19]. It is used to define all sorts of model manipulations, like language conversions, model refinements, refactorings, simulators, and code generators. Given the prominent role of MTs, they need to be thoroughly tested to guarantee the reliability of any MDE-based solution [15].

MTs are typically encoded using dedicated rule-based transformation languages to describe the mappings between the input and output model elements, or model rewritings. While there are many MT languages [7, 9, 12, 14], their development environments hardly ever support the static analysis of transformations. Thus, languages like ATL [9] and ETL [12] are popular to define MTs, but assessing the correctness of these transformations is difficult. MT verification is an active area of research [15], and much effort has been spent in verifying transformations defined with formal languages (e.g., based on graph transformation [7]). However, few verification tools can be applied on non-toy transformations defined with transformation languages widely used in practice. Compared to the IDEs of mainstream programming languages, MT environments are still rudimentary in terms of facilities to detect and fix errors, and understanding transformation programs.

To address this shortcoming, we have developed AnATLyzer: a static analysis tool for ATL transformations. ATL is one of the most widely used MT languages, backed by a mature and efficient execution runtime [9]. In contrast to other verification approaches [4, 6], AnATLyzer targets the full ATL language with the aim of being a truly practical tool. Its main features are the following: (1) its static analysis identifies more than 50 types of problems, is highly configurable, and is powered by constraint solving, including live and batch analysis; (2) integration with Eclipse and the standard ATL editor; (3) catalogue of quick fixes, including speculative ones; (4) different visualizations to help understanding the transformation and its errors; (5) programmatic API and extension capabilities. This paper presents AnATLyzer from the transformation developer perspective, and evaluates its analysis capabilities. Technical details of the static analysis and quick fixes can be found at [16, 17].

2 ATL MODEL TRANSFORMATIONS

This paper deals with model-to-model transformations, a special case of MT to transform an input model conforming to a source meta-model, into an output model conforming to a target meta-model. Such MTs are useful in different scenarios, like model analysis (e.g., transform a process model into a Petri net for verification), model refinement (e.g., create a Java model out of a UML model), model abstraction (e.g., in reverse engineering, to create models from code), or to bridge technological spaces (e.g., create a BPMN model from a UML activity diagram).

As a concrete scenario, we focus on a reverse engineering process transforming a KDM [11] model into a UML class diagram. The Knowledge Discovery Metamodel (KDM) is an OMG standard for representing legacy code in a neutral way. Listing 1 shows an excerpt of an ATL MT, part of the Modisco project [2], performing this task. While the listing only shows 3 simplified rules (slightly modified for illustration), the complete transformation has 31 rules (579 LOC). Rule ModuleToPackage (lines 1–3) creates a UML package for every KDM module. ClassUnitToClass (lines 5–12) creates a UML Class for every KDM ClassUnit. MemberUnitToProperty (lines 14–16) creates a UML Property object (e.g., an attribute) and its lower and upper cardinality objects for each KDM MemberUnit.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3183479

```
1  rule ModuleToPackage {
2   from src : kdm!Module
3   to tgt : uml!Package ( packagedElement ← src.codeElement ) }
4
5  rule ClassUnitToClass  {
6   from src : kdm!ClassUnit (
7      not src.reflmmediateComposite().oclIsTypeOf(kdm!StorableUnit) )
8   to tgt : uml!Class (
9    name ← src.model.name+src.name,
10   visibility ← src→getVisibility(),
11   ownedAttribute ← src.codeElement→select(e | e.oclIsKindOf(kdm!DataElement))
12 ) }
13
14 rule MemberUnitToProperty {
15   from src : kdm!MemberUnit
16   to tgt : uml!Property ( ), //... bindings and other object creations omitted }
```

Listing 1: Excerpt of KDM to UML transformation. Issues underlined according to their type (Error, Style warning)

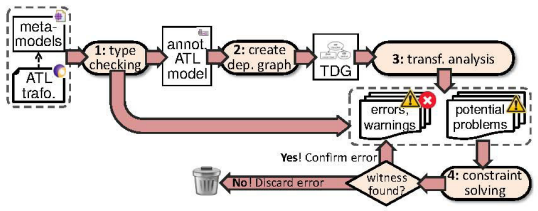

Figure 1: Static analysis process

Each rule declares a matching pattern with one or more source objects ("from" clause), and creates objects in the output model ("to" clause). Rules may define filter conditions to exclude some matches. For example, line 7 restricts the applicability of the rule to ClassUnit objects not contained in any StorableUnit (i.e., avoid transforming inner classes). The features of the created target objects are initialized via rule bindings. For features with primitive type, these are just assignments (e.g., line 10). For references (e.g., line 3), an implicit resolution mechanism assigns to the reference the target objects resulting from the transformation of the source objects selected by the right-hand side of the binding. Typically, non-primitive bindings involve navigating the source model, selecting objects using OCL expressions. ATL supports other kinds of rules, like lazy rules which must be called explicitly, and auxiliary operations called helpers. Rules need to be exclusive; otherwise, if two rules match the same source object, a runtime error is raised.

The listing has some problems discussed in the rest of the paper, but the standard ATL environment reports none. In general, MTs should meet the following quality attributes: *Q1:* is the transformation correctly typed w.r.t. to the source meta-model? *Q2*: do the generated models conform to the target meta-model (including its invariants)? *Q3*: are there conflicting or missing rules? The standard ATL environment does not help in answering Q2 and Q3, and gives poor support for Q1. The next section shows how AnATLyzer improves this situation.

3 ANATLYZER

AnATLyzer is an Eclipse plug-in that extends the basic ATL editor with new features, including automatic reporting of errors detected by static analysis. Fig. 1 depicts its analysis process.

Table 1: Some of the problems detected by AnATLyzer.

Error description	Kind	Time	Solver	Qu.
1: Feature or operation not found	typing	live	no	Q1
2: Incoherent variable declaration	typing	live	no	Q1
3: Access over undefined receptor	navigation	live	maybe	Q1
4: No binding for compulsory target feature	tgt. integrity	live	no	Q2
5: Binding resolved by rule with invalid target	tgt. integrity	live	maybe	Q2
6: Unresolved binding	rules	live	maybe	Q3
7: Rule conflict	rules	batch	maybe	Q3
8: Target invariant violation	tgt. integrity	batch	yes	Q2

Given an ATL MT, the first step is its type checking based on a type inference engine for OCL, which annotates each node of the abstract syntax tree with the inferred type. From this, a transformation dependence graph (TDG) is built. A TDG is similar to a program dependence graph but including links between bindings and their resolving rules. At this point, the analysis outputs two kinds of results: (i) actual errors and warnings, which are reported to the user; (ii) "potential problems" or smells that cannot be statically confirmed to be errors (e.g., problem in line 3 of Listing 1) but that can be verified using a model finder (a constraint solver over models). Hence, for each potential problem, we compute its OCL path condition (i.e., an OCL expression stating the requirements for an input model to make the MT fail at the problem's location) and use it as input of the finder. If a model is found, the problem is confirmed and reported to the user; otherwise, we discard the problem. Interestingly, we use a similar approach to refine the TDG deleting impossible binding-rule links. This improves the accuracy of the visualizations and program navigation actions.

Next, we review the main features of the tool.

Detecting errors. AnATLyzer identifies more than 50 problems types. Table 1 shows some of the most relevant ones. Problems #1, #2 and #3 report typing errors and OCL navigation issues, and hence contribute to answer *Q1*. Notably, checking error #3 sometimes requires using constraint solving. As an example, AnATLyzer reports an error #3 in line 9 of Listing 1 because property model is optional, and hence, the expression will fail if model is not initialized. Problems #4, #5 and #8 aim to answer *Q2* by analysing whether the MT always produces models conformant to the target meta-model. For instance, an error #5 is signalled in line 3 of the listing because the type of packagedElement is Package, but if src.codeElement contains a MemberUnit, packagedElement will be incorrectly assigned a Property (i.e., the target object in which MemberUnit was transformed by rule MemberUnitToProperty). Problems #6 and #7 signal rule errors, which requires analysing how related rules cover different fragments of the input model (i.e., *Q3*). For instance, AnATLyzer reports a potential error #6 in line 11 as there may be objects in the right-hand side of the binding not handled by any rule. This is a smell of incompleteness of the transformation. The problem can be confirmed or discarded using the solver to check whether the OCL expressions in the rule's filter and the binding prevent this possibility. Moreover, the analysis takes into account pre-conditions expressed in OCL, either declared in the transformation or encoded in an Ecore meta-model. See [17] for details on detected errors.

Fig. 2 shows a screenshot of AnATLyzer. The detected problems are highlighted in the ATL editor (label 1, underlined and with error markers) and classified in the Analysis View (label 2).

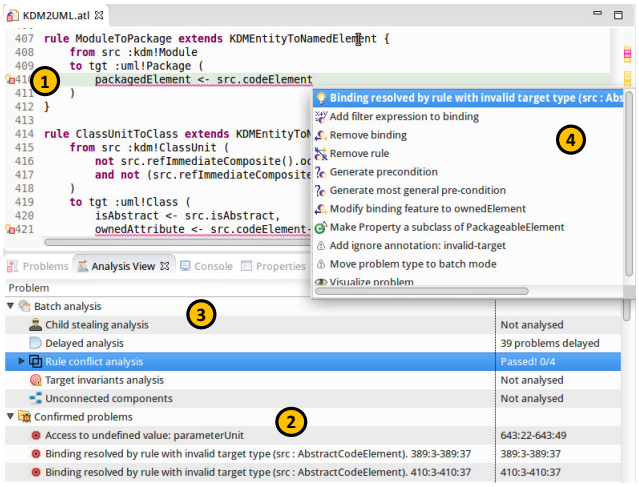

Figure 2: Screenshot of AnATLyzer.

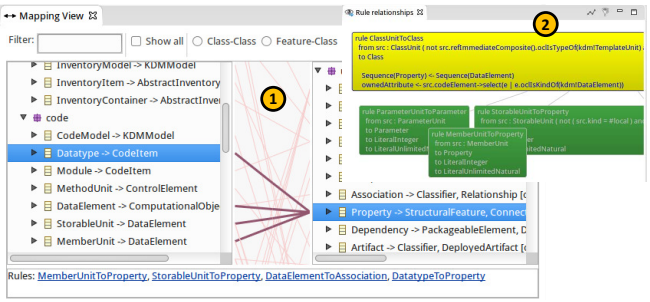

Figure 3: Screenshot of visualization facilities.

Live analysis. The static analysis performed by AnATLyzer is fast enough to be executed whenever a transformation file is saved (see Sect. 4). The constraint solving phase is also fast due to an internal meta-model pruning optimization, but its performance is less predictable and may interrupt the developer workflow. To address this issue, we launch the solver in a background process with low priority to avoid interrupting the editing process, and classify the potential problems in the Analysis View into *Running* (currently being checked by the solver), *Confirmed* (the solver has confirmed the problem), and *Discarded* (spurious problem according to the solver).

Problem checking is incremental upon transformation changes, as the solver is launched to check a potential problem only if the changes may affect the result of the last check. This is done by computing a "problem hash" based on the problem's path condition.

Finally, users can configure the solver by defining a search scope (minimum and maximum number of model objects).

Batch analysis. Some kinds of analysis may take more than a few milliseconds to complete as they heavily rely on the solver. These analyses must be performed on demand by double-clicking on their icon (label 3). There are 4 such analyses. *Rule conflict analysis* checks that a transformation does not contain overlapping rules, i.e., its rules will not match the same input objects. *Target invariant analysis* checks whether any possible output model of a transformation will fulfil a set of provided post-conditions, target meta-model invariants and transformation contracts expressed in OCL. This analysis is based on rewriting target constraints as source constraints [18]. *Child stealing analysis* checks that no object changes its container at runtime. *Unconnected components* reports whether the transformation generates disconnected subgraphs, which is a smell of bugs caused by the lack of initialization of some target references.

Fixing problems. AnATLyzer provides a comprehensive catalogue of more than 100 quick fixes to help developers easily correct the detected problems [16]. Given a problem, the user can access the available quick fixes with CTRL+1 (Fig. 2; label 4). When the best strategy to fix an error is not straightforward, the user can query a dedicated dialog in which quick fixes are executed speculatively to show the state of the transformation if a given quick fix was applied. Moreover, the IDE provides an explanation for each error, in order to help less experienced developers understand its causes.

Transformation comprehension. For any moderately complex MT, it can be difficult to grasp the accepted input model configurations, the relations between its rules, and the connections between the objects produced and consumed by the rules. This issue is exacerbated in ATL due to its implicit rule resolution mechanism.

To help comprehend a MT, AnATLyzer can render it as a graph where rules are nodes and bindings are edges, and where the rules a binding resolves are explicit (Fig. 3; label 2). At the program level, the shortcut CTRL+B allows jumping from a binding to its resolving rules, and vice versa. This navigation is very precise as it is derived from a filtered version of the TDG using constraint solving.

Another recurrent task involves understanding the relations between source and target elements. Hence, AnATLyzer provides a view in which source and target meta-model elements are connected, offering a high-level view of the mappings (Fig. 3; label 1).

To improve error comprehension, given a reported error, AnATLyzer can provide a source model (i.e., a witness) that would make the error happen at runtime. Such a witness is generated by constraint solving, can be visualized, and can be exported into XMI to use it as input to the transformation for testing.

Configuration. Reporting too many errors and warnings may overwhelm some users. For this reason, AnATLyzer permits a fine-grained configuration of the problems to be checked and reported in live mode, and which ones defer to the batch mode. It also provides pre-defined profiles like: "check all problems", "do not check warnings", "delay to batch mode if require constraint solving", etc. The problems scheduled to be executed in batch mode can be checked by clicking on the *Delayed analysis* button (Fig. 2; label 3).

4 EVALUATION

In [17], we evaluated the accuracy and performance of AnATLyzer on 101 MTs gathered from the ATL Zoo [17], and others generated via mutation. The results showed that: a) there were very few false positives and negatives due to the use of constraint solving, b) 84% of the potential problems were successfully validated by the constraint solver (i.e., AnATLyzer has good support for a variety of ATL constructs) and c) performance was generally good. However, a threat to the validity of the results is that the ATL Zoo might not be representative of the MTs used in industry, as some of them have been developed by non-experienced people.

Table 2: Setup and results of evaluation.

	T1: Java2Kdm	T2: Kdm2Uml	T3: Syml2Modelica	Total
Rules/helpers/SLOCs	109/7/2003	31/20/579	37/74/988	-
Source/target classes	146/320	320/263	378/47	-
Q1: source errors	48	22	190	260
Q2: target errors	48	25	29	102
Q3: rule behaviour	48	13	1	62
Other problems	5	0	2	7
Total	149	60	222	431
Solver executions	342	48	142	532
Successful executions	146	26	8	180
Timeouts (2.5 secs)	53	0	0	53
Errors confirmed	89	18	5	112
Parsing/analysis time [secs.]	1.4/0.7	0.4/0.2	1.1/0.7	-
Solver t. (total/median) [secs.]	180.0/0.08	4.5/0.02	32.7/0.02	-

Hence, in this paper, we evaluate AnATLyzer with 3 large third-party industrial MTs: *T1: Java2Kdm* and *T2: Kdm2Uml* are part of MoDisco, an Eclipse project for the modernization of legacy systems, and *T3: SysML2Modelica* is an implementation of an OMG standard to integrate SysML and Modelica [20]. Table 2 shows the transformation size and the evaluation results. AnATLyzer uncovers many errors (431) which were unnoticed by the developers, 112 of them discovered by constraint solving and involving mostly rule-related errors. As these MTs are supposed to have been thoroughly tested, the results evidence a remarkable usefulness of AnATLyzer for industrial practice. Regarding the ability to map complex programs to the input format of the solver, AnATLyzer provides a reasonable mapping for *T1* and *T2* up to some limitations of the solver to handle string operations; however, for *T3*, we found a limitation regarding the translation of UML profiles, whose support is future work.

Regarding performance, the main bottleneck is the parser, which is based on the standard ATL parser (1 second). The analysis is relatively fast (average 0.5 seconds), even for the largest MT. The total time spent in checking problems with the solver is fairly large for *T1* due to the amount of problems to check. However, the median (0.02 seconds) indicates that solving is normally fast. Moreover, it is done in the background and can be cancelled by the user.

5 RELATED WORK

While static analysis has been applied in graph transformation, e.g., to detect rule conflicts [7], its use to detect errors for MT languages closer to programming languages, like ATL, is uncommon. As an exception, VIATRA2 [21] performs static type checking to ensure the parameters of rules are well-typed w.r.t. a meta-model. In our case, we type-check ATL which is more challenging as it relies on OCL, and produce witness models and fixes for the errors. Specific to ATL, the static fault localization technique in [3] uses testing to identify the rules causing contract violations. This technique complements ours. In [22], the proposed Java Façade for ATL can be used to build static analyses. We opted for a new API to integrate explicit rule dependencies and error handling information.

Many works express MTs as transformation models [1] and analyse them using model finding. E.g., [4] uses this approach to check if an ATL transformation can yield models violating the output meta-model constraints. In [5], MT properties are defined in OCL and analysed with model finders. While these works assume correctly typed ATL transformations, we focus on typing errors.

Most approaches to derive and rank quick fixes come from the programming languages community. For instance, MintHint [10] analyses statistical correlation to identify likely expressions in patches, BugFix [8] learns from bugs previously fixed to delete new bugs, and [13] visualizes fix alternatives for buffer overflows in C code. However, few works propose quick fixes for MDE artefacts, and none tackles MTs that we are aware of.

6 CONCLUSIONS

This paper has presented AnATLyzer, an advanced IDE for ATL, which features a static analyser powered by constraint solving; and an Eclipse plug-in with facilities for live and batch validation, quick fixes, visualizations and navigation shortcuts. Our experiments show that the tool can deal with large and realistic transformations effectively, not being limited to "easy" subsets of ATL. AnATLyzer is free software, available under EPL license.

ACKNOWLEDGMENTS

Work funded by the Spanish MINECO TIN2014-52129-R and TIN2015-73968-JIN (AEI/FEDER/UE) and the R&D programme of Madrid (S2013/ICE-3006).

REFERENCES

[1] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow. 2006. Model transformations? Transformation models!. In *MODELS (LNCS)*, Vol. 4199. Springer, 440–453.
[2] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot. 2014. MoDisco: A model driven reverse engineering framework. *Inf. & Softw. Technology* 56, 8 (2014), 1012–1032.
[3] L. Burgueño, J. Troya, M. Wimmer, and A. Vallecillo. 2015. Static fault localization in model transformations. *IEEE TSE* 41, 5 (2015), 490–506.
[4] F. Büttner, M. Egea, J. Cabot, and M. Gogolla. 2012. Verification of ATL transformations using transformation models and model finders. In *ICFEM (LNCS)*, Vol. 7635. Springer, 198–213.
[5] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. 2010. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software* 83, 2 (2010), 283–302.
[6] Z. Cheng and M. Tisi. 2017. A deductive approach for fault localization in ATL model transformations. In *FASE (LNCS)*, Vol. 10202. Springer, 300–317.
[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. 2006. *Fundamentals of algebraic graph transformation.* Springer-Verlag.
[8] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. 2009. BugFix: A learning-based tool to assist developers in fixing bugs. In *ICPC*. IEEE Computer Society, 70–79.
[9] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. 2008. ATL: A model transformation tool. *Science of Computer Programming* 72, 1-2 (2008), 31 – 39.
[10] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. 2014. MintHint: automated synthesis of repair hints. In *ICSE*. ACM, 266–276.
[11] KDM. 2016. http://www.omg.org/spec/KDM/About-KDM/. (2016).
[12] D. Kolovos, R. Paige, and F. Polack. 2008. The Epsilon Transformation Language. In *ICMT (LNCS)*, Vol. 5063. Springer, 46–60.
[13] P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert. 2015. Automated generation of buffer overflow quick fixes using symbolic execution and SMT. In *SAFECOMP (LNCS)*, Vol. 9337. Springer, 441–456.
[14] QVT. 2016. http://www.omg.org/spec/QVT/. (2016).
[15] L. Rahim and J. Whittle. 2015. A survey of approaches for verifying model transformations. *SoSyM* 14, 2 (2015), 1003–1028.
[16] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. 2016. Quick fixing ATL transformations with speculative analysis. *SoSyM* in press (2016).
[17] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. 2017. Static analysis of model transformations. *IEEE TSE* 43, 9 (2017), 868–897.
[18] J. Sánchez Cuadrado, E. Guerra, J. de Lara, R. Clarisó, and J. Cabot. 2017. Translating target to source constraints in model-to-model transformations. In *MODELS*. IEEE Comp. Soc., 12–22.
[19] S. Sendall and W. Kozaczynski. 2003. Model transformation: The heart and soul of model-driven software development. *IEEE Software* 20, 5 (2003), 42–45.
[20] SysML-Modelica transformation. 2012. http://www.omg.org/spec/SyM. (2012).
[21] Z. Ujhelyi, Á. Horváth, and D. Varró. 2011. Static type checking of model transformation programs. *ECEASST* 38 (2011).
[22] A. Vieira and F. Ramalho. 2011. A static analyzer for model transformations. In *MtATL'11*.