THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

VICToRy: Visual Interactive Consistency Management in Tolerant Rule-based Systems

OPEN ACCESS

VICToRy: Visual Interactive Consistency Management in Tolerant Rule-based Systems

Nils Weidmann

Paderborn University
Paderborn, Germany

nils.weidmann@upb.de

Anthony Anjorin

IAV GmbH Ingenieurgesellschaft Auto und Verkehr
Berlin, Germany

anthony.anjorin@iav.de

James Cheney

University of Edinburgh
Edinburgh, United Kingdom

jcheney@inf.ed.ac.uk

In the field of Model-Driven Engineering, there exist numerous tools that support various consistency management operations including model transformation, synchronisation and consistency checking. The supported operations, however, typically run completely in the background with only input and output made visible to the user. We argue that this often reduces both understandability and controllability. As a step towards improving this situation, we present VICToRy, a debugger for model generation and transformation based on Triple Graph Grammars, a well-known rule-based approach to bidirectional transformation. In addition to a fine-grained, step-by-step, interactive visualisation, VICToRy enables the user to actively explore and choose between multiple valid rule applications thus improving control and understanding.

1 Introduction and Motivation

As an approach to model management, bidirectional transformation (bx) enables designers to derive multiple consistency management operations from a single specification. Concepts of bx for various research fields exist, including Model-Driven Engineering (MDE), databases, and programming languages. Among other formal approaches, Triple Graph Grammars (TGGs) are a rule-based approach to bx, in which the consistency relation between two models (interchangeably denoted as source and target) is expressed by a third correspondence model. A convenient feature of TGGs is that declarative rules are used as a common basis from which various operations such as unidirectional transformation, model synchronisation and consistency checking are derived. This can, however, be confusing for designers who are not aware of or used to the underlying derivation process. While some TGG tools provide basic debugging functionality for the transformation process (cf. Sect. 5), none of them enable the user to track let alone influence the choice of rule applications. In most cases, only the input and output models are visible to the user, whereas the transformation process runs in the background. Stevens [28] argues for more transparent and fault-tolerant model transformation approaches, such that the user should be involved in controversial decisions, i.e. decisions that cannot be made only based on the consistency relation specification. Even for uncontroversial transformations, we have observed that novice users are unable to fully understand how TGG tools - viewed as black-boxes - determine a specific result.

In this paper, we propose the *VICToRy* debugger[1] as an add-on component for supporting an interactive step-by-step visualisation of model transformations based on TGGs. It presents possible operational rules including their concrete application contexts to the user, as well as a history of the involved models as they evolve during the transformation process. Additionally, the user can inspect and choose a valid rule application at each time step, or decide to resume the automated process in the background. Up to now, the operations model generation and forward/backward transformation are supported, while

---

[1] https://github.com/eMoflon/emoflon-victory

model synchronisation and consistency checking are planned to be implemented soon. VICToRy is currently integrated into the eMoflon tool suite[2] but can be potentially connected to other Java-based TGG and even general graph transformation tools via the defined interfaces. This means that existing and future tools can be enriched with debugging facilities to increase user involvement and understanding in the transformation process, and thereby contributing to fault-tolerant model management. Previous work [7, 1] identified possible connections between bx and *provenance*, to which our approach makes a further practical contribution. By providing access to why-provenance (i.e. why is a given pair of models consistent?) and how-provenance (i.e. how it can be proven that two models are consistent?) that bx tools produce and maintain, we strive to improve the understandability of such tools. For TGG tools, it was identified that why-provenance is provided by the resulting correspondence model, and how-provenance by the underlying sequence of rule applications. While this knowledge helps to understand TGG tools conceptually, it is also important to provide interactive access to these data structures. VICToRy visualises both the correspondence model and the sequence of rule applications, allowing users to actively inspect and shape the emerging provenance structures.

The remainder of this paper is structured as follows: An overview of supported features is presented in Sect. 2 using a running example. The software architecture of VICToRy is sketched in Sect. 3. A brief overview of a first case study with VICToRy is provided in Sect. 4. In Sect. 5 related work is discussed, while Sect. 6 concludes the paper.

2   An Overview of Supported Features

This section provides an overview of features of VICToRy that can help novice users explore an unknown TGG. To identify these features, we conducted an explorative study with students who did not have any MDE-related experience at that point (cf. Sect. 4 for details).

2.1   Running Example

To demonstrate different features of VICToRy, the bx example *CompanyToIT* is used,[3] which maintains consistency between a simplified organisational structure of a company and its corresponding IT infrastructure. The two metamodels are depicted in Fig. 1. A `Company` consists of multiple `Admins`, `Employees`, and at most one `CEO`. Each `Admin` reports to the `CEO` of the `Company`, just like the other `Employees`. The corresponding `IT` involves a set of `Routers` and `Networks`; a `Router` is always assigned to a particular `Network`. The `Network` itself consists of a set of `PCs` and a set of `Laptops`. A `Company` and an `IT` correspond to each other; this is depicted by a dashed line connecting the two nodes in the metamodel. The same holds for an `Admin` who is responsible for a `Router` within the IT infrastructure. Finally, an `Employee` can either work
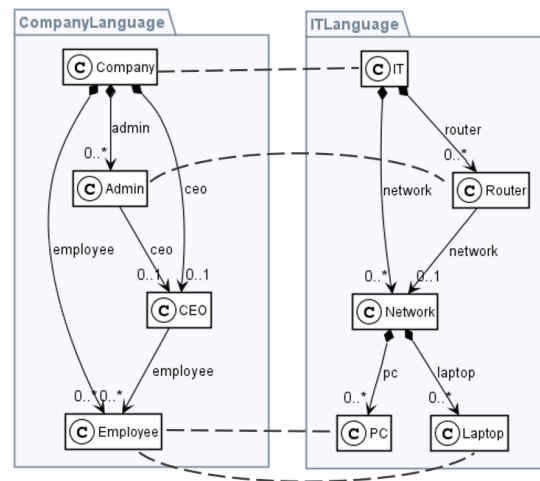

Figure 1: Metamodels: `CompanyToIT`

---

with a `PC` or a `Laptop`, leading to two correspondence links for the `Employee` in the metamodel.

2.2 Configurable Visualisation of Rules and Rule Applications

To understand the effects of a rule application on a concrete model, it is essential to visualise both the rule and the resulting model changes at runtime. The example rule `AdminToRouter`, which creates an `Admin` in the `Company` model, and relates them to a created `Router` in a created `Network` in the `IT` model, is depicted to the left of Fig. 2. Created (context) elements of the rule have a green (black) outline.
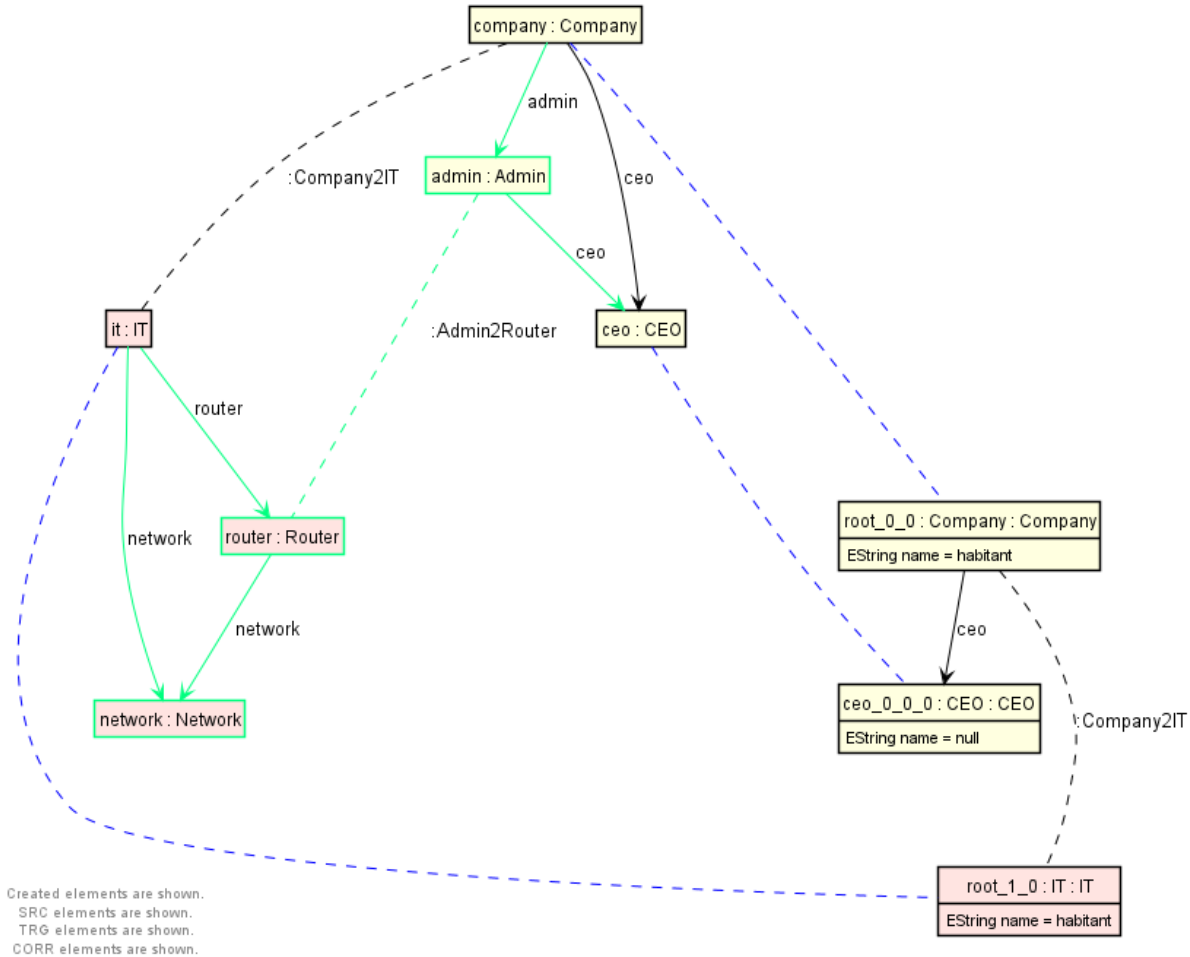
Figure 2: Visualising rules and matches

A specific match of this rule is depicted to the right of Fig. 2. Variables in the rule and corresponding matched elements in the model are connected with dashed purple lines. In both the rule and its match, the background colour of source model elements is peach, while target model elements have a rose background. Correspondences are represented as dashed black lines. The visualisation of rules and rule applications is based on PlantUML[4] and is generated automatically on rule and match selection (cf. Sect. 2.3). Editing rules is only possible in the underlying TGG tool, meaning that rules cannot be

---

[4]https://plantuml.com/

adapted at runtime. As source, target and correspondence models are depicted as a connected triple, the visualisation represents why-provenance for TGGs. To cope with a wide range of size and complexity of TGG rules, models' sizes, and the user's proficiency, it is crucial to be able to configure the visualisation:

Choice of displayed elements: For each domain (source, target, correspondence), the user can hide the respective elements. For rules, it is also possible to display only context elements and thus focus on the structure required for a match of that rule in the model instance.

Abbreviation of labels: For nodes, edges and correspondences, it is possible to display the labels completely, in an abbreviated form containing the first and last three letters, or not at all.

Neighbourhood of matches: As models of realistic size can become much too large to be completely displayed within the debugger, only the match of a selected rule application and a configurable neighbourhood of this match is displayed. The distance of a node to the match is defined as the shortest path from this node to any node contained in the match; nodes in the match itself are assigned a distance of 0. The k-neighbourhood of a match contains all nodes that are at most $k \in [0; 3]$ away.
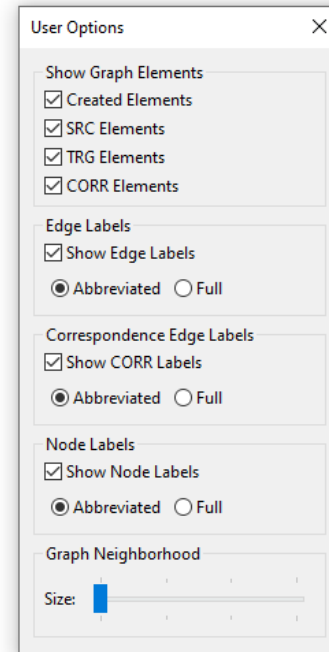

Figure 3: Configuration

2.3 Explorable and Interactive Overview of Applied Rules

The debugger's user interface (UI) provides an overview of all rules of the TGG. Our example TGG CompanyToIT consists of only four rules, which are depicted as a list. For each rule in the list, the number of available matches in the current model and the number of applied matches are displayed together with the name of the rule. Rules with a dark grey background are not applicable in the current state of the model, whereas rules with a white background have at least one applicable match. This provides a quick overview

Figure 4: Available rules and matches

and is useful for TGGs with a large number of rules. Furthermore, rules that have never been applicable are crossed out, providing a quick visual indication of rules that might be problematic. All matches of a rule can be viewed as sub-entries by expanding the corresponding rule entry in the list. The currently selected rule and match are visualised in the UI (cf. Sect. 2.2). To apply a rule, the user can either double-click on a particular match, select the match and press the apply button, or simply double-click the rule to apply a random match. This action is delegated to the connected TGG tool, which must handle the actual rule application (cf. Sect. 3). As soon as VICToRy receives a response, the UI is updated to reflect the new state of the model and available matches.
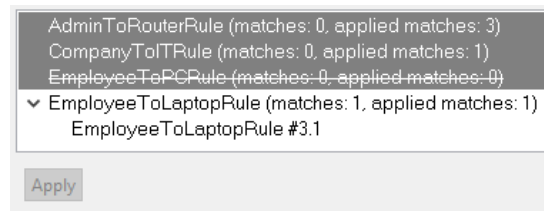
The VICToRy debugger provides traceability information by keeping track of all previous rule applications. This sequence of rule applications is referred to as the (transformation) protocol and represents how-provenance [1]. For each protocol entry, the name of the rule as well as a unique ID for the rule application is displayed. If a protocol entry is selected, the state of the model as created by all rule applications up to and including the selected one is displayed with a config-

Figure 5: The protocol section

urable neighbourhood; elements created by the selected rule application are highlighted green. It is also possible to select multiple entries: the respective rule applications are then combined into a single step and visualised accordingly.

3 Architecture

VICToRy can be connected to different Java-based TGG tools by implementing a simple interface for transferring data between the debugger and the respective tool. An overview of this interface is depicted in Fig. 6. The central component of this interface is the `Graph` class, which consists of `Nodes` and `Edges`. There exists a mapping from each `Edge` to a source and a target `Node`, reflecting the categorical approach to graph transformation. Both `Rules` and `Matches` are represented as `Graphs` consisting of rule and model elements, respectively.

Multiple `Matches` can be determined for the same `Rule`. Furthermore, a `RuleApplica-tion` object is created when a `Rule` is applied for a concrete `Match`. All this information is stored in a `DataPackage` transferred between the attached TGG tool and VIC-ToRy after each transformation step. Further information is stored in the `DataProvider`, including all current models and the current choice between debugging and background modi. As all information necessary to derive the current model state from the initial models and the given TGG rules are provided,
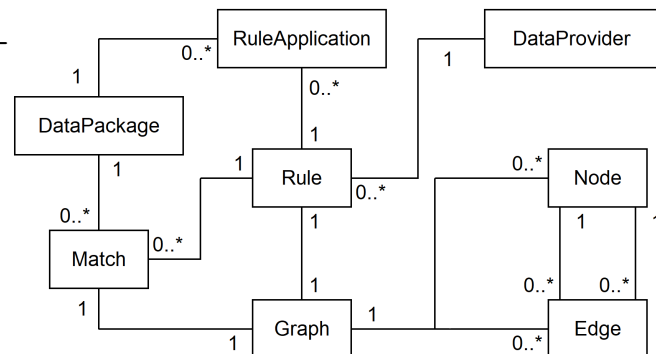
Figure 6: Data exchange with VICToRy

the interface represents a data exchange metamodel for why- and how-provenance between VICToRy and the attached TGG tool. The component diagram in Fig. 7 describes how the debugger has been embedded into the eMoflon tool suite. VICToRy is designed for - but not limited to - being connected to a TGG component of eMoflon. Currently, both eMoflon::IBeX [31] and eMoflon::Neo[5] implement the interface to the debugger. eMoflon::IBeX uses the incremental pattern matcher Democles [30] for matching patterns to graph instances which are loaded from XML metadata interchange (XMI) files into main memory, whereas eMoflon::Neo uses the external graph database Neo4J to collect matches and store models. Depending on the operation, both tools are able to construct a superset of possible rule applications, encode the graph problem into an optimisation problem and let an external Integer Linear

---

[5] https://github.com/eMoflon/emoflon-neo

Programming (ILP) solver determine the final set of rule applications. Interfaces to the solvers Gurobi[6] and SAT4J[7] are currently implemented, while other solvers can be connected to the tool suite as well.
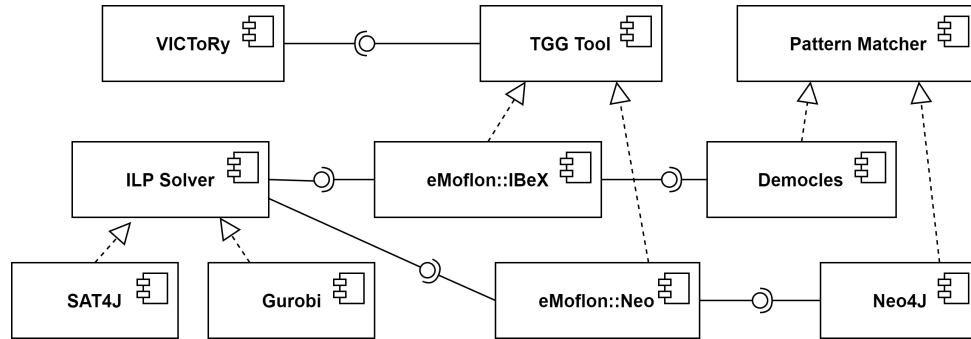

Figure 7: Integrating VICToRy into the eMoflon tool suite

While performing a consistency management task with VICToRy, the tool switches between two modi as depicted in Fig. 8. In the *background mode*, possible matches for rules are collected and one of them is chosen to be applied.

In case of multiple options, rule applications are chosen at random (or according to a configurable component) without user interaction, which is the usual workflow for model transformation tools. This procedure is repeated until no further matches can be found (leading to the termination of the process) or until a breakpoint is reached. In the latter case, the tool switches to the *debug mode*, where the VICToRy UI is visible and each rule application requires a user interaction. In contrast to the background mode, the user can choose between multiple options which are shown on the UI. To return to the background mode, the user resumes the automated choice of rule application by quitting the

Figure 8: State chart for debugging modi

debugger. This behaviour is similar to debugging concepts in contemporary integrated development environments (IDEs), but without the possibility of stepping *into* a rule application.

4   Case Study

In order to qualitatively evaluate the usefulness of VICToRy, we conducted a case study adapted from Blouin et al. [4]. The Architecture Analysis and Design Language (AADL) is a standard language in the aerospace domain, for which a textual editor (OSATE) and a graphical editor (Adele) exist. Blouin et al. [4] discuss the challenge of implementing a bx to synchronise models edited using the different editors. Our evaluation aims at answering two research questions related to motivational aspects for MDE debuggers:

1. Does VICToRy help to explore and understand a TGG of realistic size?
2. Does VICToRy help to identify bugs in rules or in input models?

4.1 The Adosate TGG

Blouin et al. implemented the original Adosate TGG that maintains consistency between Adele and OSATE models. The TGG consists of 60 rules specified with the model transformation tool MoTE [14], in which an extended TGG formalism is used that allows the designer to connect more than one element per model with a single correspondence. For the implementation with eMoflon, we created multiple correspondences for each pair of involved source and target nodes. While the eMoflon TGG is much larger than with MoTE, its rule refinement feature [2] was used to keep the rules manageable. The feature allowed us to define rules involving abstract node types (abstract rules) that can be refined by concrete types and enriched with additional elements in so-called concrete rules. These changes resulted in a semantically equivalent TGG (i.e., a TGG that generates the same language) with 49 abstract rules and 91 concrete rules, of which only the concrete rules are considered at runtime. An overview comparing the (concrete) rules required for the implementations with MoTE and eMoflon is provided in Table 1.

AADL Construct	Number of Rules		AADL Construct	Number of Rules	
	MoTE	eMoflon		MoTE	eMoflon
Package (axiom)	1	2	Component Type Features	10	19
Subcomponents	11	12	Feature Group Types	4	4
Component Types	2	13	Feature Group Type Features	10	9
Connections	20	21	Component Implementation	2	9
			Total	**60**	**91**

Table 1: Size of rule groups: MoTE and eMoflon

Compared to most bx benchmark examples[8] the number of rules is relatively large, while the average rule size is comparable to existing benchmarks. The mean of the number of nodes involved in abstract and concrete rules is 6.51 (2.69 created nodes, 3.81 context nodes), and the mean number of edges is 3.31 (2.46 created edges, 0.84 context edges) per rule.

4.2 Experiment and Results

In an experiment conducted at Paderborn University, Germany, with 15 computer science graduate students without substantial prior experience with MDE, we attempted to assess if and how VICToRy helps novice users understand a provided, non-trivial TGG. The first task was to identify relations between rules and model elements (which model elements are created/required by which rules?), as well as relations between the rules themselves (which rules depend on other rules?). The students were provided with the Adosate TGG and VICToRy, and asked to work independently on developing an understanding for the TGG. The first and second authors provided helpful material (tutorials, handbooks, and relevant papers), supervision (answering any basic questions), and held a feedback meeting after two weeks to check the students' understanding for the TGG and ask if VICToRy was helpful and in what ways it was used. In general, all students stated that the debugger indeed helped to get an overview of the entire TGG. It was especially helpful to identify which rules are applicable for an empty model (axioms) without clicking through all of them, and to determine which rules provide context for other rules (rule dependencies). Most students started exploring the TGG rules by generating consistent model triples and inspecting the resulting transformation protocol. In a second step, they then attempted to transform

---

[8]http://bx-community.wikidot.com/examples:home

smaller instances in forward and backward directions. As all models can be saved to disk at any point of time, it was easily possible to try out different alternatives starting from a common state of all models.

In a second task, the students were provided with the Adosate TGG, and with a test suite consisting of input models and expected output models for both forward and backward directions. One of the students was then asked to either make a change to a TGG rule, or to the supplied input models, resulting in both cases in a mismatch between TGG and test suite. The other students were then asked to determine and explain this mismatch. This task was carried out in a slightly more controlled manner, restricting the allotted time to a few hours, and asking the students not to perform a diff between the original and changed rules. The feedback from the students for this task (based again on a feedback meeting and discussion) was much less positive. The task turned out to be (1) much too difficult for novice users, and (2) VICToRy proved not to be of much help as it does not provide any information about why a rule is *not* applicable in a specific situation (even though the expectation is that it should be). While the possibility of selecting protocol entries and inspecting the resulting changes on the models was appreciated, many students stated that the opposite direction, i.e. selecting model elements in the visualisation and highlighting the "responsible" rule applications, would be indeed helpful as well and is currently missing. Furthermore, there was a clear need for the introduction of breakpoints, i.e. the transformation should start in the background mode and stop at a certain point defined by the user, e.g., when a specific rule is applicable for the first time. This enables users to skip irrelevant parts of the transformation that are already clear to them and set the focus on debugging problematic steps.

Revisiting our research questions, our initial exploratory experiment at least indicates that (1) VICToRy appears to help obtain an overview of a non-trivial TGG the user is not familiar with, but (2) leaves room for improvement regarding the detection of errors in either models or rules. An extension towards *why-not-debugging* facilities [1] seems to be necessary to properly address bug finding tasks. Our experiment is at best a pre-study for a more formal, controlled experiment and quantitative evaluation with multiple use cases, a larger group of test persons, and objective measures. While we cannot generalise our results, our goal was not to provide hard empirical evidence for the effectiveness of VICToRy but rather to explore the design space in a realistic setting and brainstorm together with novice users for promising features to guide future extensions of VICToRy.

5   Related Work

Several approaches to debugging in MDE have been proposed, including fundamental concepts, debugging Domain-Specific Language (DSL) code, and debuggers for non-deterministic approaches. Mierlo et al. describe a stepping semantics for debugging in MDE with four levels of different granularity [22]. The proposed approach is, however, conceptual and does not provide an implementation to the best of our knowledge. A debugger for Petri nets is based on Modelverse and supports basic functionality including breakpoints known from general purpose language (GPL) debuggers [23]. The prototype is planned to be extended to support model transformations as well. A wide range of facilities for DSL debugging is presented in previous work. Omniscient debugging - in contrast to stepwise execution - provides the user with enhanced navigation and exploration features such as reverting execution steps at runtime, impacting performance and scalability. Therefore, approaches are often tailored to rather small instances [8] or specific use cases, such as xDMSLs (a subset of DSLs) [5]. Lindeman et al. propose a declaratively defined debugger for DSLs [20]. The approach was integrated into the Spoofax language workbench and evaluated by case studies involving the textual DSLs StrategoTL and WebDSL. However, several limitations are mentioned for debugging modelling languages and model transformations.

Laurent et al. extended the Foundational UML (fUML) by debugging facilities [18]. While the approach is a tool-independent add-on, it considers only the execution of models complying to the fUML standard.

For debugging rule-based systems, Tichy et al. sketch how to execute debugging steps for graph transformation, taking the tool Henshin as an example [29]. In contrast to our approach, the debugging of rule applications is much more detailed and takes the matching process into account as well, whereas an implementation is not described. Similarly, Jukss et al. use graph transformations as an underlying formalism for a debugger integrated into AToMPM [15]. The approach focusses on a fine-grained inspection of the rule application process, whereas the user is not enabled to choose between multiple possible rule applications. For algebraic graph transformation, the tool AGG [26] provides a mode for stepwise execution of graph transformations. Rule and match can be chosen by the user in each step, while it is neither clear which rules are applicable in the current state, nor a protocol of previous rule applications is provided.

Furthermore, a substantial number of tools have emerged that implement bidirectional model transformations based on TGGs. While in Fujaba [12] and the TGG Interpreter [13], operational rules are directly interpreted, MoTE [14, 11], eMoflon [19, 31], HenshinTGG [9] and EMorF [16] compile rules into source code of a GPL to be executed at runtime. A concept for debugging TGGs at different levels was introduced to the TGG Interpreter by Rieke [25]. The debugging facilities are, however, tightly interwoven with the specific tool and several open challenges for practical use are mentioned. For MoTE, a monitor is implemented which allows to stepwise execute model transformations [11]. However, the user cannot influence the execution order, which is determined by order of correspondence nodes in a processing queue and their respective types. A debugging mode is implemented for EMorF as well, but both a detailed description and the tool itself are not currently available. For all other TGG-based tools, debugging functionality is missing to the best of our knowledge.

Besides these rule-based approaches, debugging plays an important role in other MDE-related fields as well. Proposed concepts include work on dynamic meta modelling [3], discrete event system specificationss (DEVSs) [21], and story diagrams [17], which are each tailored to a specific tool and use case, though. The tool TETRA Box is based on PaMoMo and involves white-box testing of transformation languages by symbolic execution of model transformations [27], which is independent of the underlying transformation language but not yet tested with realistic examples. SyVOLT localises errors in the input based on igraph and the T-Core framework [24], while the focus of debugging is set on detecting reasons for contract violations rather than on the transformation process. Ferdjoukh et al. localise faults in meta-model design based on static analyses and implemented their approach in TIWIZI and GRIMM [10], whereas model transformations are not taken into account.

6 Conclusion and Future Work

We presented the add-on component VICToRy for interactively visualising single steps of the model generation and transformation process. Besides the inspection of possible rule applications in the current state, the user can inspect the prior transformation process using a transformation protocol. The TGG-based tool is fully integrated into the eMoflon tool suite but can be used along with other applications via a defined interface. A concept for switching between background mode and debug mode via breakpoints is presented, whereas an implementation for breakpoints is left for future work. Besides model generation and transformation, other consistency management operations such as model synchronisation and consistency checking are planned to be supported as part of future work. To improve the tool's usability and effectiveness, support for easily defining different kinds of breakpoints should be implemented.

Furthermore - as an extension towards supporting why-not provenance - information about reasons for blocked rule applications should be presented to the user to support detecting logical faults in TGG rules, or a mismatch with expectations in provided input models and tests.

Acknowledgments

References

[1] Anthony Anjorin & James Cheney (2019): *Provenance Meets Bidirectional Transformations*. In: *11th International Workshop on Theory and Practice of Provenance (TaPP 2019)*, USENIX Association, Philadelphia, PA. Available at https://www.usenix.org/conference/tapp2019/presentation/anjorin.

[2] Anthony Anjorin, Karsten Saller, Malte Lochau & Andy Schürr (2014): *Modularizing Triple Graph Grammars Using Rule Refinement*. In Stefania Gnesi & Arend Rensink, editors: *FASE 2014*, *LNCS* 8411, Springer, pp. 340–354, doi:10.1007/978-3-642-54804-8_24.

[3] Nils Bandener, Christian Soltenborn & Gregor Engels (2010): *Extending DMM Behavior Specifications for Visual Execution and Debugging*. In Brian A. Malloy, Steffen Staab & Mark van den Brand, editors: *SLE 2010*, *LNCS* 6563, Springer, pp. 357–376, doi:10.1007/978-3-642-19440-5_24.

[4] Dominique Blouin, Alain Plantec, Pierre Dissaux, Frank Singhoff & Jean-Philippe Diguet (2014): *Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report*. In Davide Di Ruscio & Dániel Varró, editors: *ICMT 2014*, *LNCS* 8568, Springer, pp. 106–121, doi:10.1007/978-3-319-08789-4_8.

[5] Erwan Bousse, Jonathan Corley, Benoît Combemale, Jeffrey G. Gray & Benoit Baudry (2015): *Supporting efficient and advanced omniscient debugging for xDSMLs*. In Richard F. Paige, Davide Di Ruscio & Markus Völter, editors: *SLE 2015*, ACM, pp. 137–148. Available at https://dl.acm.org/doi/abs/10.1145/2814251.2814262.

[6] Loli Burgueño, Jonathan Corley, Nelly Bencomo, Peter J. Clarke, Philippe Collet, Michalis Famelis, Sudipto Ghosh, Martin Gogolla, Joel Greenyer, Esther Guerra, Sahar Kokaly, Alfonso Pierantonio, Julia Rubin & Davide Di Ruscio, editors (2017): *MODELS 2017 Satellite Event*. *CEUR Workshop Proceedings* 2019, CEUR-WS.org. Available at http://ceur-ws.org/Vol-2019/.

[7] James Cheney, Anthony Finkelstein, Bertram Ludäscher & Stijn Vansummeren (2012): *Principles of Provenance (Dagstuhl Seminar 12091)*. *Dagstuhl Reports* 2(2), pp. 84–113, doi:10.4230/DagRep.2.2.84.

[8] Jonathan Corley, Brian P. Eddy, Eugene Syriani & Jeff Gray (2017): *Efficient and scalable omniscient debugging for model transformations*. *Software Quality Journal* 25(1), pp. 7–48, doi:10.1007/s11219-015-9304-4.

[9] Claudia Ermel, Frank Hermann, Jürgen Gall & Daniel Binanzer (2012): *Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars*. *ECEASST* 54, doi:10.14279/tuj.eceasst.54.771.

[10] Adel Ferdjoukh & Jean-Marie Mottu (2018): *Towards an Automated Fault Localizer while Designing Metamodels*. In: *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018*, pp. 547–552. Available at http://ceur-ws.org/Vol-2245/mdebug_paper_4.pdf.

[11] Holger Giese, Stephan Hildebrandt & Leen Lambers (2014): *Bridging the Gap between Formal Semantics and Implementation of Triple Graph Grammars - Ensuring Conformance of Relational Model*

Transformation Specifications and Implementations. Software and System Modeling 13(1), pp. 273–299, doi:10.1007/s10270-012-0247-y.

[12] Holger Giese & Robert Wagner (2006): *Incremental Model Synchronization with Triple Graph Grammars*. In Oscar Nierstrasz, Jon Whittle, David Harel & Gianna Reggio, editors: *MoDELS 2006*, *LNCS* 4199, Springer, pp. 543–557, doi:10.1007/11880240_38.

[13] Joel Greenyer & Ekkart Kindler (2010): *Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars*. Software and System Modeling 9(1), pp. 21–46, doi:10.1007/s10270-009-0121-8.

[14] Stephan Hildebrandt, Leen Lambers, Holger Giese, Dominic Petrick & Ingo Richter (2011): *Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations*. In Andy Schürr, Dániel Varró & Gergely Varró, editors: *AGTIVE 2011*, *LNCS* 7233, Springer, pp. 238–253, doi:10.1007/978-3-642-34176-2_20.

[15] Maris Jukss, Clark Verbrugge & Hans Vangheluwe (2017): *Transformations Debugging Transformations*. In Burgueño et al. [6], pp. 449–454. Available at http://ceur-ws.org/Vol-2019/mdebug_1.pdf.

[16] Lilija Klassen & Robert Wagner (2012): *EMorF - A Tool for Model Transformations*. ECEASST 54, doi:10.14279/tuj.eceasst.54.768.

[17] A. Krasnogolowy, S. Hildebrandt & S. Wätzoldt (2012): *Flexible Debugging of Behavior Models*. In: *ICIT 2012, Proceedings*, pp. 331–336, doi:10.1109/ICIT.2012.6209959.

[18] Yoann Laurent, Reda Bendraou & Marie-Pierre Gervais (2013): *Executing and debugging UML models: an fUML extension*. In Sung Y. Shin & José Carlos Maldonado, editors: *SAC 2013*, ACM, pp. 1095–1102, doi:10.1145/2480362.2480569.

[19] Erhan Leblebici, Anthony Anjorin & Andy Schürr (2014): *Developing eMoflon with eMoflon*. In Davide Di Ruscio & Dániel Varró, editors: *ICMT 2014*, *LNCS* 8568, Springer, pp. 138–145, doi:10.1007/978-3-319-08789-4_10.

[20] Ricky T. Lindeman, Lennart C. L. Kats & Eelco Visser (2011): *Declaratively Defining Domain-Specific Language Debuggers*. In Ewen Denney & Ulrik Pagh Schultz, editors: *Proceedings of GPCE 2011, Portland, Oregon, USA, October 22-24, 2011*, ACM, pp. 127–136, doi:10.1145/2047862.2047885.

[21] Simon Van Mierlo, Yentl Van Tendeloo & Hans Vangheluwe (2017): *Debugging Parallel DEVS*. Simulation 93(4), pp. 285–306, doi:10.1177/0037549716658360.

[22] Simon Van Mierlo, Yentl Van Tendeloo & Hans Vangheluwe (2018): *A Generalized Stepping Semantics for Model Debugging*. In: *Proceedings of MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018*, pp. 541–546. Available at http://ceur-ws.org/Vol-2245/mdebug_paper_3.pdf.

[23] Simon Van Mierlo & Hans Vangheluwe (2017): *Debugging Non-determinism: a Petrinets Modelling, Analysis, and Debugging Tool*. In: *MODELS 2017 Satellite Event, Austin, TX, USA, September, 17, 2017*, pp. 460–462. Available at http://ceur-ws.org/Vol-2019/mdebug_4.pdf.

[24] Bentley James Oakes, Levi Lucio, Clark Verbrugge & Hans Vangheluwe (2018): *Debugging of Model Transformations and Contracts in SyVOLT*. In: *MODELS 2018 Workshops, Copenhagen, Denmark, October, 14, 2018*, pp. 532–537. Available at http://ceur-ws.org/Vol-2245/mdebug_paper_1.pdf.

[25] Jan Rieke (2015): *Model Consistency Management for Systems Engineering*. Ph.D. thesis, University of Paderborn. Available at https://digital.ub.uni-paderborn.de/ubpb/urn/urn:nbn:de:hbz:466:2-15597.

[26] Olga Runge, Claudia Ermel & Gabriele Taentzer (2011): *AGG 2.0 - New Features for Specifying and Analyzing Algebraic Graph Transformations*. In Andy Schürr, Dániel Varró & Gergely Varró, editors: *AGTIVE 2011, Budapest, Hungary, October 4-7, 2011*, *LNCS* 7233, Springer, pp. 81–88, doi:10.1007/978-3-642-34176-2_8.

[27] Johannes Schönböck, Gerti Kappel, Manuel Wimmer, Angelika Kusel, Werner Retschitzegger & Wieland Schwinger (2013): *TETRABox - A Generic White-Box Testing Framework for Model Transformations*. In: *APSEC 2013*, pp. 75–82, doi:10.1109/APSEC.2013.21.

[28] Perdita Stevens (2014): *Bidirectionally Tolerating Inconsistency: Partial Transformations*. In Stefania Gnesi & Arend Rensink, editors: *FASE 2014*, Springer, Berlin, Heidelberg, pp. 32–46, doi:10.1007/978-3-642-54804-8_3.

[29] Matthias Tichy, Luis Beaucamp & Stefan Kögel (2017): *Towards Debugging the Matching of Henshin Model Transformations Rules*. In Burgueño et al. [6], pp. 455–456. Available at http://ceur-ws.org/Vol-2019/mdebug_2.pdf.

[30] Gergely Varró, Anthony Anjorin & Andy Schürr (2012): *Unification of Compiled and Interpreter-Based Pattern Matching Techniques*. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle & Dimitrios S. Kolovos, editors: *ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, LNCS 7349, Springer, pp. 368–383, doi:10.1007/978-3-642-31491-9_28.

[31] Nils Weidmann, Anthony Anjorin, Lars Fritsche, Gergely Varró, Andy Schürr & Erhan Leblebici (2019): *Incremental Bidirectional Model Transformation with eMoflon::IBeX*. In James Cheney & Hsiang-Shang Ko, editors: *Bx@PLW 2019*, CEUR Workshop Proceedings 2355, CEUR-WS.org, pp. 45–55. Available at http://ceur-ws.org/Vol-2355/paper4.pdf.