JOT
JOURNAL OF
OBJECT TECHNOLOGY

Human-in-the-Loop Large-Scale Model Transformations with the VICToRy Debugger

Nils Weidmann[*], **Enes Yigitbas**[*], **Anthony Anjorin**[†], **Ankita Srivastava**[*], **and Jane Jose**[*]
[*]Paderborn University, Germany
[†]IAV GmbH Ingenieurgesellschaft Auto und Verkehr, Germany

ABSTRACT Model-Driven Engineering (MDE) is an approach to software engineering with a well-defined formal basis and rich tool-support. For MDE tools to be actually usable by practitioners in industry, adequate debugging facilities must be provided. This is especially true for model transformation tools, which are often declarative in the sense that they intentionally hide low-level details such as the exact order of pattern matching and rule selection. Building debuggers for such tools is challenging, as users require enough information to locate and understand problematic parts, but do not want to be swamped with the technical bits and pieces of a transformation possibly consisting of hundreds of steps. In this paper, we report on our continued research on the VICToRy debugger, a debugger for rule-based bidirectional model transformation tools. After presenting the general idea for VICToRy in prior work, we focus in this paper on closing two gaps we have observed in research: (i) exploring how best to support breakpoints as a means of efficiently debugging large-scale human-in-the-loop model transformations, and (ii) how to debug the propagation of concurrent updates between multiple models, i. e., concurrent model synchronisation. Both features allow the user to actively interact with the transformation tool: the former by switching between manual and automated selection and application of rules, the latter by involving the user in the choice of conflicting rule applications. We provide a qualitative evaluation of these extensions indicating that supporting breakpoints does indeed make the debugger more accessible to users, and that we addressed the most important requirements for debugging concurrent model synchronisation.

KEYWORDS model transformation, visual debugging, consistency management

1. Introduction and Motivation

Model-Driven Engineering (MDE) puts models at the centre of the software development process. This enables a wide range of domain experts, both with and without programming experience, to contribute and collaborate on a suitable level of abstraction. Building software systems from models involves different *consistency management* operations, including model transformation, synchronisation and consistency checks. When developing software systems of realistic complexity, practitioners in industry can only use MDE tools if these provide adequate debugging facilities for locating and understanding problems. This is even more the case for consistency management tools, which, often intentionally, shield users from low-level details such as the exact order of pattern matching and rule selection.

While development environments for General Purpose Languages (GPLs) usually provide mature debuggers for step-wise program execution and the definition of breakpoints as well as watchpoints, support for productive debugging provided by MDE tools in general and consistency management tools in particular, is often inadequate or even completely missing. State-of-the-art consistency management tools usually run their operations completely in the background with only input and output made visible to the user, arguably reducing both understandability and controllability. Even for uncontroversial transformations, we have observed that novice users are unable to fully understand how consistency management tools - viewed as black-boxes - determine a specific result. We argue

JOT reference format:
Nils Weidmann, Enes Yigitbas, Anthony Anjorin, Ankita Srivastava, and Jane Jose. *Human-in-the-Loop Large-Scale Model : Transformations with the VICToRy Debugger*. Journal of Object Technology. Vol. 21, No. 3, 2022.

http://dx.doi.org/10.5381/jot.2022.21.3.a8

that understanding and thus trust in the results of consistency management tools can be increased by actively involving the user i. e., an integration expert possessing knowledge about the involved domains, in key decisions made during the consistency restoration process.

To explore the solution space for debugging consistency management operations, the *VICToRy debugger*[1] was developed as an add-on component for consistency management tools. The rule-based approaches of (algebraic) graph transformation for unidirectional and Triple Graph Grammars (TGGs) (Schürr 1994) for Bidirectional Transformation (BX) form the formal basis for VICToRy. BX is an approach to consistency management, which makes it possible to derive different operations from a single specification of the consistency relation (Abou-Saleh et al. 2016). VICToRy is currently integrated into the eMoflon tool suite[2] but can be potentially connected to other Java-based TGG and graph transformation tools via provided interfaces. This means that existing and future tools can be enriched with debugging facilities to increase user involvement and understanding in the transformation process.

While a step corresponds to a single instruction for most GPL debuggers, this "smallest executable unit" is a rule application for the VICToRy debugger. Indeed, this is what distinguishes the VICToRy debugger from other MDE debuggers we are aware of, which tend to focus on making the actual pattern matching process and the generation of individual nodes and edges observable by the user. Our approach with VICToRy is to remain high-level and, instead of mimicking GPLs step-by-step debuggers, to research into features that can nonetheless provide just enough information for users to locate and understand problems with their transformations.

In this paper, we focus on two such high-level debugging strategies. Firstly, we explore how the general idea of breakpoints can be transferred to VICToRy without revealing the technical specifics of the underlying TGG tool. Secondly, we discuss how users can be involved in the process of conflict resolution. While transformation and consistency checking operations can, in principle, be solved without any user interaction, determining satisfactory solutions after concurrent updates is hardly possible without human-in-the-loop approaches involving an integration expert. This is because *conflicts* cannot be avoided in general and make it impossible to restore consistency and incorporate all changes at the same time. Existing approaches (Orejas et al. 2020; Fritsche et al. 2020; Weidmann, Fritsche, & Anjorin 2020) require the user to predefine a general resolution strategy but emphasize that more research in this direction is needed. Our approach with VICToRy is to involve the integration expert actively in the conflict resolution process via a debugger with suitable features for this task. As this is a largely unexplored solution space, we continuously consulted with multiple experts from the different and very diverse BX subfields to get their feedback and suggestions to further shape our ideas.

An initial version of the debugger was presented in prior work (Weidmann, Anjorin, & Cheney 2020), which is extended by different features in the current paper. As novel contributions, the current paper presents (i) a new breakpoint concept involving breakpoints of different types, (ii) a prototype for a concurrent synchronisation component, and (iii) an initial user study to gather requirements for and evaluate (i) and (ii). All concepts are demonstrated using a complex model transformation example, originating from an industrial case study from the railway domain (Salunkhe et al. 2021; Weidmann et al. 2021).

The remainder of this paper is structured as follows: The VICToRy debugger is compared to existing MDE debuggers in Sect. 2, before a motivating example is introduced in Sect. 3. An architectural overview is provided in Sect. 4, before Sect. 5 introduces our breakpoint concept. Section 6 provides an overview of the debugger's front-end. The User Interface (UI) prototype of a concurrent synchronisation component is presented in Sect. 7. The results of empirical studies with both MDE experts and novices are summarised in Sect. 8. Sect. 9 concludes the paper and provides some directions for future research.

2. Related MDE Debuggers

Several approaches for debugging in MDE have been proposed, including debugging Domain-Specific Language (DSL) code, executing state machine models, and debuggers for model transformation approaches. Table 1 provides an overview of related MDE debuggers, and in which MDE tools they are implemented. Aligned with this paper's contributions, we analyse whether the approaches support breakpoints (BP), model transformations (MT) or only model execution, and – if yes – whether the transformation approach is bidirectional (BX), being a prerequisite for implementing a component for synchronising concurrent updates.

Reference	Implementation	BP	MT	BX
(Bousse et al. 2015)	GEMOC Studio	✓	✗	✗
(Laurent et al. 2013)	unnamed	✓	✗	✗
(Wimmer et al. 2009)	TROPIC	✓	✓	(✓)
(Tichy et al. 2017)	?	✓	✓	✗
(Jukss et al. 2017)	AToMPM	✓	✓	✗
(Runge et al. 2011)	AGG	✗	✓	✗
(Mészáros et al. 2013)	VMTS	✓	✓	✗
(Rieke 2015)	TGG Interpreter	✓	✓	✓
(Giese et al. 2014)	MOTE	✗	✓	✓
(Klassen & Wagner 2012)	EMorF	✗	✓	✓
(Mierlo et al. 2017)	Python PDEVS	✓	✗	✗
(Krasnogolowy et al. 2012)	SD Interpreter	✓	✓	✗
(Cuadrado et al. 2018)	AnATLyzer	(✓)	✓	✗
(Schönböck et al. 2013)	$\text{TETRA}_{\text{Box}}$	✗	✓	(✓)
(Ujhelyi et al. 2012)	VIATRA2	✓	✓	✗
(Kolovos 2009)	Eclipse Epsilon	✓	✓	✗
(Oakes et al. 2018)	SyVOLT	✗	✗	✗

Table 1 Overview of related MDE debuggers

A wide range of facilities for DSL debugging is presented in existing 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

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

are often tailored to specific use cases, such as xDSMLs, a subset of DSLs (Bousse et al. 2015). Laurent et al. extended the foundational UML (fUML) by debugging facilities (Laurent et al. 2013). While the approach is a tool-independent add-on, it considers only the execution of models complying with the fUML standard. Wimmer et al. present a debugger for model transformations based on mediniQVT and coloured petri nets (Wimmer et al. 2009). Both models and metamodels can be inspected in each step of the model transformation, while the transformation itself cannot be directed by the user. Furthermore, the authors point out that the approach suffers from the ambiguous operational semantics of the QVT-R standard.

For debugging rule-based systems, Tichy et al. sketch how to execute debugging steps for graph transformations, taking the tool Henshin as an example (Tichy et al. 2017). 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 (Jukss et al. 2017). The approach focuses 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 (Runge et al. 2011) 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. Meszaros et al. present a visual model transformation debugger for step-by-step execution of model transformations based on graph transformations (Mészáros et al. 2013). Single steps of the transformation can be influenced by the user, being restricted to in-place model transformations, though.

Furthermore, multiple TGG tools (cf. (Leblebici et al. 2014) for an overview) have been extended by debugging facilities, which appear to be limited in several respects, though. A concept for debugging TGGs at different levels was introduced to the TGG Interpreter by Rieke (Rieke 2015). 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 (Giese et al. 2014). However, the user cannot influence the execution order, which is determined by the order of correspondence nodes in a processing queue and their respective types. A debugging mode is implemented for EMorF (Klassen & Wagner 2012) as well, but both a detailed description and the tool itself are currently not available.

Besides these rule-based approaches, debugging plays an important role in other MDE-related fields as well. Proposed concepts include work on Discrete Event System Specifications (DEVSs) (Mierlo et al. 2017) and story diagrams (Krasnogolowy et al. 2012), which are each tailored to a specific tool and use case, though. Cuadrado et al. use the AnATLyzer (Cuadrado et al. 2018) for static analyses of ATL model transformations, which is able to detect typing errors and conflicting rules at compile-time. The $\text{TETRA}_{\text{Box}}$ framework is based on the PaMoMo language and involves white-box testing of transformation languages by symbolic execution of model transformations (Schönböck et al. 2013), which is independent of the underlying transformation language but not yet tested with realistic examples. Generated traces of the underlying model transformation tools, i.e., VIATRA (Ujhelyi et al. 2012) and Epsilon (Kolovos 2009), to gain insights into the transformation process. SyVOLT localises errors in the input based on igraph and the T-Core framework (Oakes et al. 2018), while the focus of debugging is set on detecting reasons for contract violations rather than on the transformation process.

In total, no MDE debugger we are aware of fulfils the requirements of being tool-independent, offering a breakpoint concept, and supporting bidirectional transformations at the same time. For a detailed overview of debugging and testing model transformations, the interested reader is referred to a recent survey by Troya et al. (Troya et al. 2022).

3. Motivating Example

In order to motivate the necessity of both breakpoint facilities for MDE debuggers and user interaction when synchronising concurrent updates, we consider a practical case study from the railway domain presented in prior work (Salunkhe et al. 2021; Weidmann et al. 2021). For engineering railway systems, the well-known Systems Modeling Language (SysML) (Holt & Perry 2019) can be synergetically combined with the formal language Event-B (Abrial & Hallerstede 2007), with which safety properties can be verified. A BX between models of the two languages ensures that consistency is maintained throughout the development process.

This motivates us to take a closer look at the definition of the consistency relation. Suppose that a BX between the two languages has been defined, and the SysML state machine depicted in Fig. 1 results from a backward transformation of the Event-B code in Fig. 2. The Event-B machine can be considered as well-formed, i. e., it does not contain any syntactic errors. The SysML state machine, however, involves a dangling outgoing edge from the `START` state, which violates the syntax of the SysML language. It is non-trivial, though, to identify the source of the error: It can either originate from the Event-B model and be reproduced by the transformation engine, or the specification of the BX in use is incorrect, or both issues even occur at the same time.

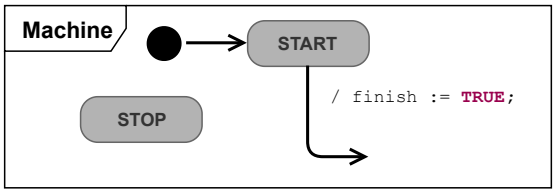

Figure 1 Faulty SysML model

In order to detect the source of the error, it seems desirable to step-wise execute the backward transformation. As Fig. 1 and 2 have shown the model instances in concrete syntax, the abstract syntax shall be briefly introduced at this point. Figure 3 depicts the relevant excerpt of the SysML metamodel for state

```
1  MACHINE Machine
2  VARIABLES START STOP finish
3  ...
4  EVENTS
5    INITIALISATION ≙
6    ...
7    COMPLETION ≙
8      WHEN
9        isin_START : START = TRUE
10     THEN
11       leave_START : START := FALSE;
12       act1 : finish := TRUE;
13     END
14 END
15
```

Figure 2 Equivalent Event-B code

machines to the left, the Event-B metamodel to the right. To improve readability, only multiplicities different from `1` for the source and `0..*` for the target of an association are depicted. For representing the consistency relation with TGGs, a third "correspondence" model is used that maps related language constructs of source and target. Visually, the mapping is represented in form of hexagons.

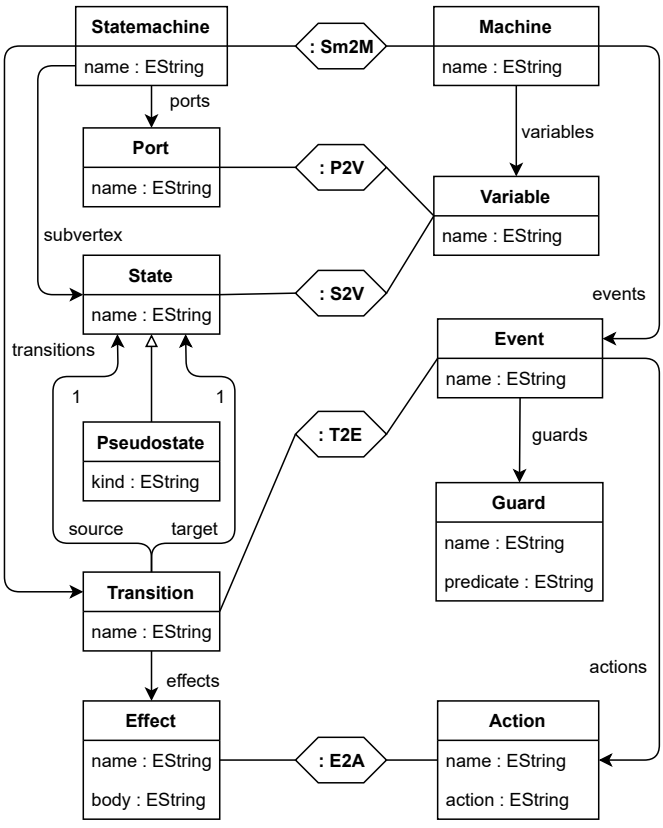

Figure 3 SysMLToEventB: Triple metamodel

The `Statemachine` class defines the primary behaviour of the modelled system and consists of `States`, `Transitions`, and `Ports`. The Event-B `Machine`, which consists of `Variables` and `Events`, has the same purpose, so these classes correspond to each other. The `States` of a SysML state machine ("START" and "STOP" in Fig. 1) correspond to `Variables` in the Event-B model (l. 2 in Fig. 2). The same holds for `Ports` in SysML models ("finish"), which are used for communication with external components. `Pseudostates` are a special form of `States` that are used to represent transient states, such as the initial state.

A `Transition` in SysML represents the directed relation between a source and a target `State`. Activating a `Transition` is similar to the occurrence of an `Event` in Event-B (ll. 5 and 7), thus these elements are connected via a correspondence. An `Event` incorporates `Guards` to restrict its occurrences and `Actions` that take place during the `Event` (l. 9). `Actions` in Event-B (l. 12) thus correspond to `Effects` in SysML, with which the respective transitions are annotated. There are also generated actions (l. 11) that do not correspond to any effect in SysML. To keep the example small and understandable, only a subset of the language constructs from the original case study is covered here. The interested reader is referred to prior work (Salunkhe et al. 2021; Weidmann et al. 2021) for a more detailed description.

Besides the (triple) metamodel, a set of *rules* is used to define the consistency relation. A TGG-based consistency management operation, e. g., a backward transformation from Event-B to SysML, consists of a sequence of rule applications, which is made accessible to the user via the VICToRy debugger. Due to space limitations, only two rules are shown to demonstrate the rationale of rule-based consistency management.

Figure 4 shows the TGG rule *StatemachineToMachine*, which creates a state machine `sm` in the SysML model, and connects it to an Event-B machine `m` via a correspondence node. All elements are coloured green and marked up with a ++, which indicates that the respective elements are created when applying the rule. The expression at the bottom of Fig. 4 ensures that the values of the name attributes of `sm` and `m` are equal. Expressions of this kind are denoted as *attribute conditions*.

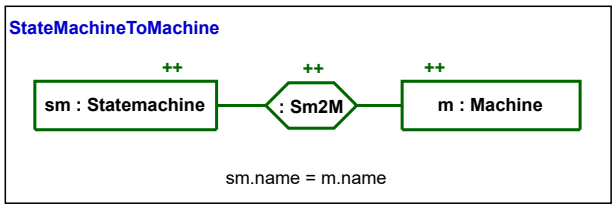

Figure 4 TGG rule: StatemachineToMachine

The second rule *PortToVariable* adds a port `p` to a state machine `sm`, and connects it to a variable `v` in the Event-B model (cf. Fig. 5). Again, the names of `p` and `v` must be equal. In contrast to *StatemachineToMachine*, the rule also involves black elements without mark-ups, which we denote as *context elements*. Context elements are required to exist before applying the rule, which is determined by *pattern matching* on the (triple) graph instance. In the concrete example, the rule *StatemachineToMachine* can create the required context elements for applying *PortToVariable*. Note that in both Fig. 4 and Fig. 5, the *declarative* forms of the respective TGG rules are shown, which

have to be operationalised for specific tasks, such as forward or backward transformations.

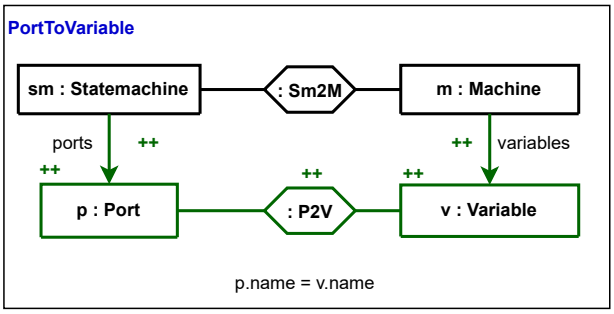

Figure 5 TGG rule: PortToVariable

In order to detect the root cause of the incorrect transformation result, these two rules are probably irrelevant, though, as the problematic part of the SysML model, i. e., the dangling edge going out of the `START` state, is not affected by the rules. It would therefore be helpful to define *breakpoints* that pause the transformation, e. g., when transitions are created, or when rules are applied that connect transitions to states. While this example is kept minimal for demonstration purposes, the specification of such breakpoints would substantially reduce time and efforts for debugging transformation sequences that consist of hundreds of rule applications.

Instead of analysing the problem's origin, it could also happen that SysML and Event-B experts update the respective models independently to remove the error. In case that the experts do not follow the same approach to improve the situation, it is especially challenging to restore consistency, as changes might be in conflict with each other. To overcome this issue, we propose a *concurrent synchronisation UI* as an extension of the VICToRy debugger, which enables the user to resolve such conflicts in a reasonable manner. Before turning towards the advanced debugging concepts, the debugger's software architecture will be sketched in the upcoming Sect. 4.

4. Architecture

VICToRy can be connected to different Java-based TGG tools by implementing an interface for transferring data between the debugger and the respective tool. An overview of this interface is depicted in Fig. 6. Compositions have multiplicities of `1` and `0..*`, if not stated otherwise. The central component of this interface is the `DataPackage` class, which bundles the data that is transferred between VICToRy and the TGG tool. A `DataPackage` contains all relevant `Rules`, `Matches`, and `Rule Applications`. Multiple `Matches` can be determined for the same `Rule`. Furthermore, a `RuleApplication` object is created when a `Rule` is applied for a concrete `Match`.

These three classes are represented as `Graphs` consisting 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 (cf. (Ehrig et al. 2006)). The `Nodes` have a set of `Attributes`, and a `Domain`, i. e., a marking that indicates whether they belong to the source or target model. For `Edges`, the domain can be determined from their source and target nodes: If both `Nodes` are part of the source (target) model, the `Edge` also belongs to the source (target) model. If `Edges` connect `Nodes` of different domains, VICToRy considers the `Edge` as a correspondence link (`EdgeType` "CORR"). Within a rule or a graph instance, `Edges` have the `EdgeType` "NORMAL". Mappings between rules and graph instances are represented as `Edges` of type "MATCH". Finally, each element has an `Action`, indicating whether the element is created, translated (marked), or required as context by the rule.

The component diagram in Fig. 7 describes how the debugger has been embedded into the eMoflon tool suite, and can potentially be connected to other Java-based TGG tools. The tool suite consists of the two main components IBeX (Weidmann et al. 2019) and Neo (Weidmann & Anjorin 2021), which both implement the interface to the debugger. The VICToRy adapter is tool-specific and needs to be implemented in order to connect the debugger to a TGG tool. It is responsible for providing both the debugger and the TGG tool with the required information, as shown in Fig. 6. The debugger itself consists of a controller that delegates user commands to the adapter, and, in turn, receives updated information about new matches and the current state of the models. All relevant information is made available to the user via the UI, whereas the breakpoint manager is responsible for checking breakpoint conditions. In the following, an overview of the breakpoint concept (Sect. 5) and the UI (Sect. 6) will be provided.

5. Breakpoint Concept

While performing a consistency management task with VICToRy, the tool switches between the two modes `RUN` and `BREAK`, as depicted in Fig. 8. In the `RUN` 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 according to a configurable component (e. g., at random in the simplest case) without user interaction, which is the usual work-flow 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 `BREAK` mode, where the VICToRy UI (Fig. 9) is visible and each rule application requires a user interaction: Either the user lets the tool choose the next rule application, or selects a rule application manually from the list of all options. To return to the `RUN` mode, the user resumes the automated choice of rule applications by a corresponding UI command. This behaviour is similar to debugging concepts in contemporary Integrated Development Environments (IDEs), but without the possibility of stepping *into* a rule application.

The implementation of the breakpoint concept follows a slightly more complex work-flow, as there are multiple *evaluation times* for breakpoints: A breakpoint can be hit either after the pattern matching step (as shown in Fig. 8), after the match selection, or after the actual rule application. As the handling of breakpoints is equal for all evaluation times, the diagram was simplified to preserve readability.

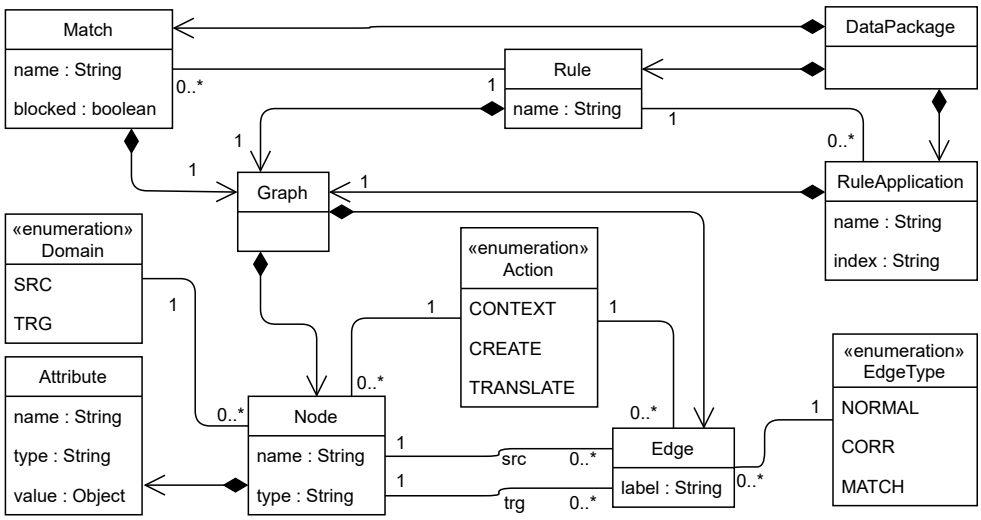

Figure 6 Data exchange with VICToRy

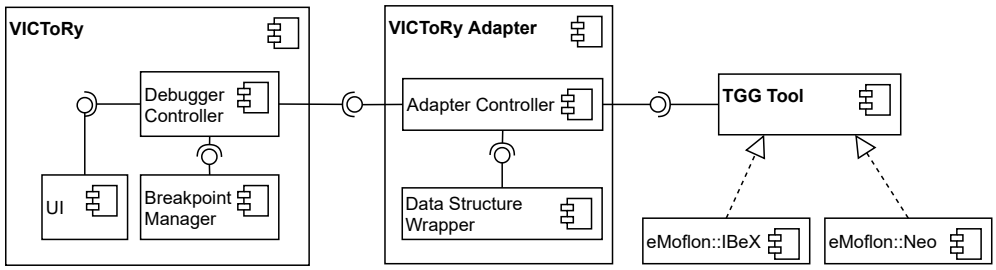

Figure 7 Integrating VICToRy into the eMoflon tool suite

While the previously described work-flow applies to all breakpoints of the VICToRy debugger, there are multiple types of breakpoints that serve different purposes. They can be subdivided into *model breakpoints*, *match breakpoints*, and *combined breakpoints*, which will be explained in the remainder of this section. Due to the estimated substantial efforts, pattern and node breakpoints have not been implemented yet, all other types are part of the VICToRy debugger.

5.1. Model Breakpoints

The break conditions of model breakpoints solely depend on the current state of the model instance at hand.

- **Model size breakpoint**: This breakpoint type counts the number of nodes in the triple. As soon as a predefined size is reached, the breakpoint is hit. The intention behind this type is to pause the transformation process at some point to inspect the intermediate result.
- **Pattern breakpoint**: Breakpoints of this type are hit as soon as a specified pattern can be matched on the host graph at least once. Especially for detecting domain constraint violations, such as multiplicity constraints defined in the metamodel, this breakpoint type is useful.

5.2. Match Breakpoints

Match breakpoints refer to the current match to be applied, i. e., the condition rather depends on the next transformation step than on the entire model instance.

- **Rule name breakpoint**: With this breakpoint type, the transformation process breaks each time a predefined rule (identified by its name) is applied. This type is handy in cases where the user assumes that the definition of a particular rule is faulty.
- **Number of matches breakpoint**: For a predefined number n, a breakpoint of this type is hit if at least n matches are collected. The counted matches can be restricted to one or multiple rules.
- **Element type breakpoint**: Every time an element of a specific type is created, the breakpoint condition is fulfilled. Element types of all three models can be used.
- **Attribute condition breakpoint**: Attribute conditions specify the consistency relation between attributes in source and target model (cf. Fig. 4 and 5). However, this breakpoint type is not restricted to conditions that are attached to the rules of the TGG at hand. Any boolean expression that can be defined using the meta-model attributes and Java standard language features can be expressed, making this breakpoint type especially powerful.

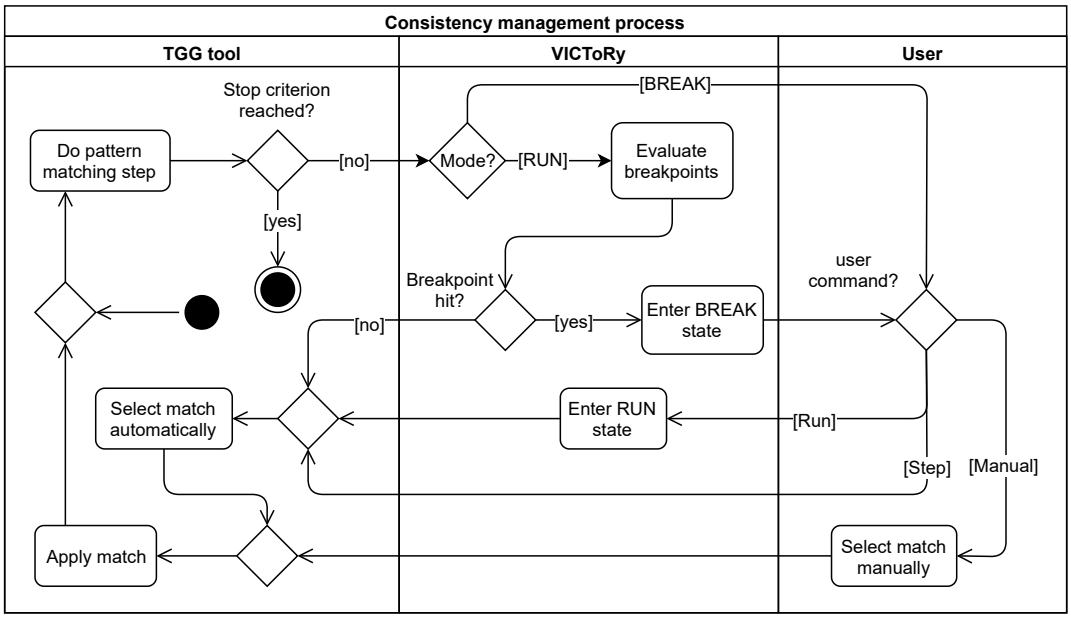

Figure 8 Breakpoint concept of the VICToRy debugger

– **Node breakpoint**: Finally, this breakpoint type pauses the transformation process as soon as a particular node, identified by its ID, is part of a match. It can be used in situations for which it is probable that a fault occurs when a specific node is created or translated.

5.3. Combined Breakpoints

Combined breakpoints can be formed out of all previously described breakpoint types (also denoted as "atomic" breakpoints). In a combined breakpoint, both atomic and combined breakpoints can be connected with AND, OR, and NOT, such that a propositional logic over breakpoints is defined. Considering a breakpoint as a boolean variable that is true if and only if the breakpoint condition is fulfilled, the combined breakpoint is hit as soon as the formed expression is true.

6. An Overview of the User Interface

The conceptual introduction to the architecture and the breakpoint concept of the VICToRy debugger is complemented with a brief presentation of the UI in this section. An overview of features of VICToRy from the UI perspective is provided, that can help novice users explore an unknown TGG and enable experts to detect faults in a complex specification.

6.1. Configurable Visualisation of Rules and Matches

First, the main windows of the debugger will be presented (cf. Fig. 9). 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. VICToRy supports both features via its visualisation section (1). It shows the visualisation of a model triple resulting from applications of the rules *StatemachineToMachine* (Fig. 4) and *PortToVariable* (Fig. 5). 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 the resulting triples is based on PlantUML[3] and is generated automatically on rule and match selection, which will be explained later in the course of this section. Editing rules is only possible in the underlying TGG tool, meaning that rules cannot be adapted at runtime.

To cope with a wide range of TGG rule sizes, model sizes, and the varying proficiency of users, it is crucial to be able to configure the visualisation. Via a pop-up menu (2), the user has a range of configuration options (available via clicking the "User Options" button):

– **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 on 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 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 with a distance of at most $k \in [0;3]$.

[3] https://plantuml.com/

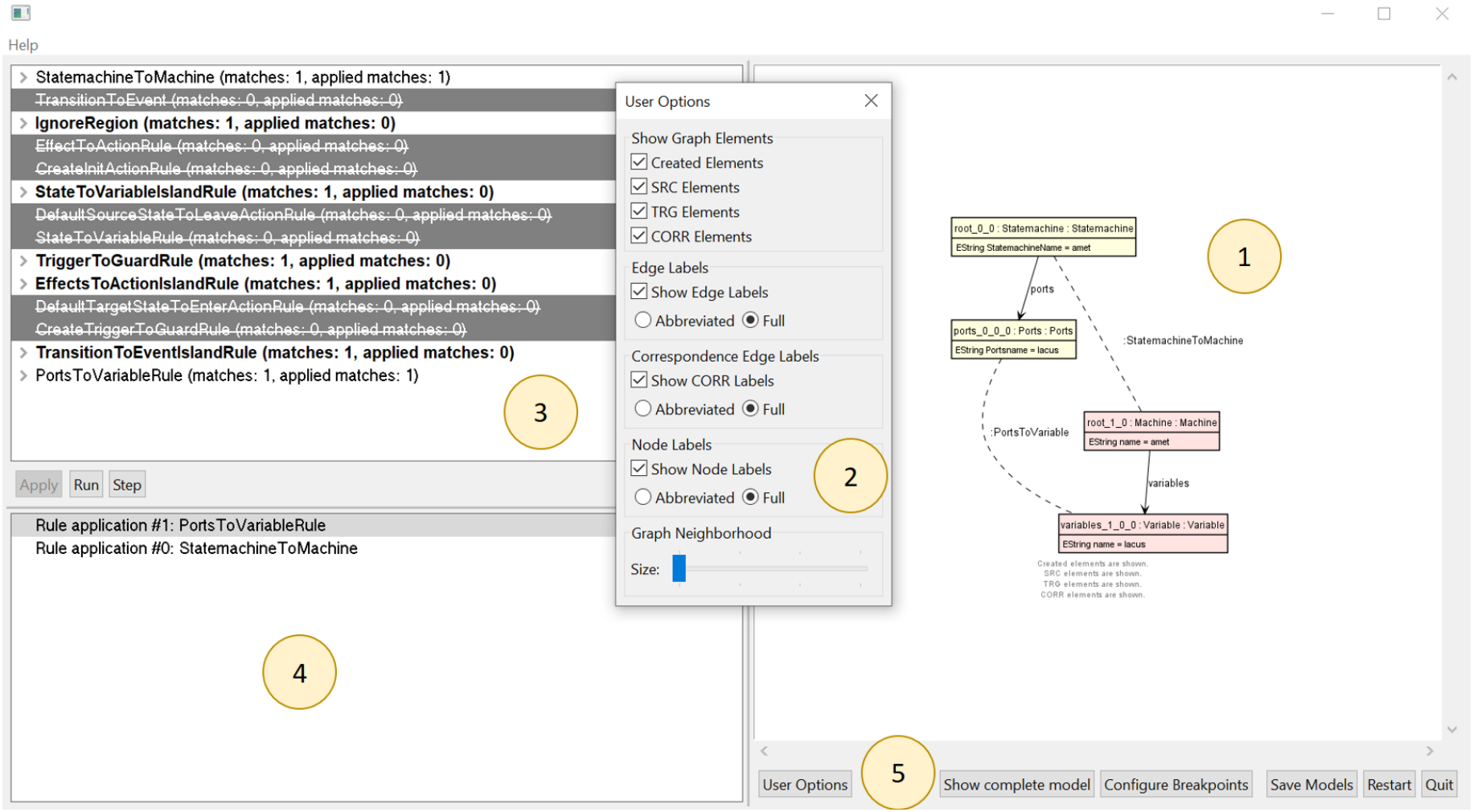

Figure 9 Main window of the VICToRy debugger

6.2. Interactive Overview of Applied Rules

Changes in the visualisation can be triggered by selecting rules or *potential* rule applications from the rule section (3) or *actual* rule applications from the protocol section (4).

The rules section provides an overview of all rules of the TGG. The entire rule set of the example TGG *SysMLToEventB* consists of 14 rules, which are depicted as a list, and involves node types that are not part of the metamodel of Fig. 3. 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 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. When selecting a rule from this list, it is visualised in the right part of the UI (1). 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 of this rule. The buttons "Run" and "Step" instruct the debugger to select the next match automatically (cf. Fig. 8). The user command is delegated to the connected TGG tool, which must handle the actual rule application. 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. 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 configurable neighbourhood. It is also possible to select multiple entries: the respective rule applications are then combined into a single step and visualised accordingly.

Several buttons provide the user with additional functionality (5). The "Show complete model" button resets the selection in the left part of the screen and visualises the entire triple instance. The three models in their current state can be stored as XML Metadata Interchange (XMI) files to continue the debugging process later ("Save models"). It is also possible to start the transformation process from scratch again ("Restart") or to stop the process completely ("Quit"). Finally, breakpoints can be defined on a separate screen, which is explained in the following.

6.3. Breakpoint Menu

The definition of breakpoints (cf. Sect. 5) for the debugging process is possible via a pop-up menu that can be opened by selecting the "Configure Breakpoints" button on the main window. A list of breakpoints of different types is depicted in Fig. 10 and 11. The colour scheme indicates that the model size breakpoint (in grey) is deactivated, whereas the number of matches breakpoint (in green) is evaluated after the pattern matching step. The combined breakpoint (in black) is evaluated also after the automatic selection step. Below the list of breakpoints, further options enable the user to configure the breakpoint. The options differ depending on the breakpoint type, such that the UI is dynamically adapted to the type of the selected breakpoint.

In Fig. 10, an element type breakpoint as part of the combined breakpoint is selected. The element type can be chosen via a drop-down menu, currently the type `Trigger` is selected. The evaluation time depends on the configuration of the combined breakpoint, but it is possible to disable the element type breakpoint at this level.

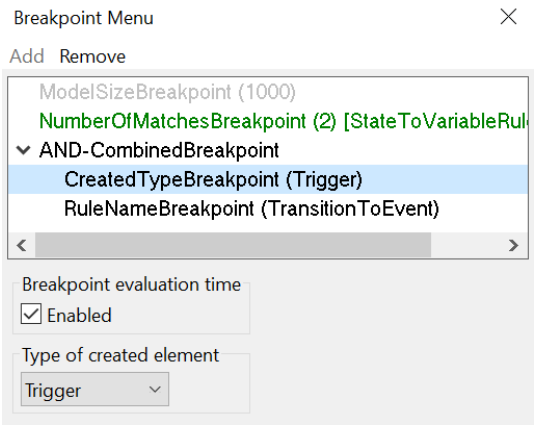

Figure 10 Element type breakpoint

The configuration of the combined breakpoint is depicted in Fig. 11. Besides the element type breakpoint of Fig. 10, it consists of a rule name breakpoint for the rule *TransitionToEvent*. By using the combination type `AND`, we specify that the combined breakpoint is hit only if the conditions for both sub-breakpoints are fulfilled. Via the radio buttons on the bottom, the user can choose whether the conditions must be fulfilled by the same match, or whether it is sufficient if the conditions are fulfilled by different matches. Combined breakpoints can also be used as part of other combined breakpoints, making it possible to create complex nested structures.

After providing an overview of the debugger's main component, the prototypical implementation of the concurrent synchronisation component is sketched in the following Sect. 7.

7. Concurrent Synchronisation Component

Existing TGG-based approaches to concurrent model synchronisation (Orejas et al. 2020; Fritsche et al. 2020; Weidmann, Fritsche, & Anjorin 2020) compute solutions in a fully automated manner. While this strategy is time-efficient and minimises manual efforts, the acceptance of a consistency management tool would benefit substantially from involving the user into the resolution of controversial decisions. To improve the situation, we designed a concurrent synchronisation component, enabling the user to influence the synchronisation process.

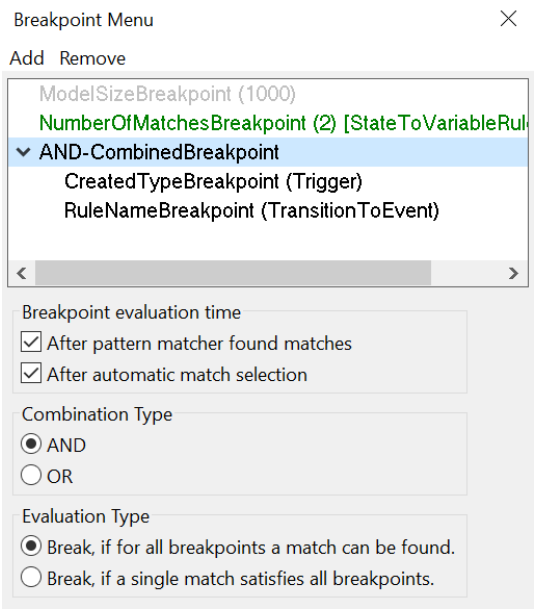

Figure 11 Combined breakpoint

As this component – to the best of our knowledge – is the first of its kind, the requirements for such an interactive synchroniser are largely vague. Clarifying the requirements and taking design decisions carefully seemed to be very important for the development process of the concurrent synchronisation component due to the novelty of an interactive model synchroniser. The ARCADIA method (Voirin et al. 2015), which originates from the systems engineering domain and recently gained popularity for software development processes as well, was used to design the prototype. ARCADIA, which was frequently used in industrial contexts already, promotes a view-point-driven approach and emphasizes a clear distinction between need and solution (Roques 2016). In an iterative manner, artefacts of previous phases are re-used in later phases of the development process. Due to time restrictions, we focussed on the *operational analysis* and *system analysis* phases to construct a first UI prototype, whereas ARCADIA involves three further phases building upon them. Up until now, a standalone UI prototype was developed; the actual integration into the VICToRy debugger is left to future work.

The front-end of the developed UI prototype is depicted in Fig. 12. In the middle section (1), the current state of the model instance is visualised. It shows a problematic situation resulting from independent changes that aim at removing the errors from the example instances of Fig. 1 (SysML) and 2 (Event-B): The SysML engineer has connected the states "START" and "STOP" with a transition, whereas the Event-B expert has deleted the "STOP" state and introduced a self-loop on the "START" state. This leads to two conflicts: First, the "STOP" state cannot be deleted and set as the target of the transition at the same time, denoted as *create-delete conflict*. Second, the transition cannot have two targets, which in this case is considered as a *create-create conflict*. For details on the TGG-based detection and classification of conflicts, the interested reader is referred to Fritsche et al. (Fritsche et al. 2020).

The colour scheme for the backgrounds of source and target model elements is the same as for the main component, but the semantics of the elements' frames is different: It indicates whether the respective element is created, deleted, or remains unchanged, as shown in the legend on the bottom right (2).

On the left-hand side (3), a list of conflicting changes that have not been resolved yet is presented to the user. For each conflict, all involved elements are listed. Each element of the list, or the entire conflict, can be highlighted in the visualisation via bold lines by selecting it in the conflict list. For each conflict, the user can choose whether it shall be resolved manually or automatically, as shown in the open context menu (4). For a manual resolution, the user selects one of the generated descriptions. In the concrete example, the component offers the user to resolve the create-create conflict by choosing either of the states as target for the transition. The automatic resolution chooses an option based on a predefined policy.

On the right-hand side (5), some statistical numbers are shown that help the user keeping track of the applied changes. In the tool bar (6), additional features of the concurrent synchronisation component are depicted: As for the main component, storing the current state of the models on disk ("Save" button) and configuring the visualisation ("Set Preferences") is possible. In case of undesired effects of the last action, the user can step back to the previous state. Furthermore, the user is in charge of accepting the final (conflict-free) solution as an outcome of the synchronisation process, and can trigger consistency checks at any point of time. Further opportunities and challenges will be discussed in Sect. 8.2 based on the feedback we received from experts during the development process.

8. Evaluation

In order to qualitatively assess the usefulness of VICToRy for involving the user into consistency management operations, we conducted an empirical evaluation with MDE experts from different subfields. This section can only provide a brief summary of the results, the interested reader is referred to a more extensive discussion in prior work (Jose 2021; Srivastava 2021). Our evaluation aims at answering two research questions related to motivational aspects for MDE debuggers:

RQ1 How valuable are breakpoints in different debugging scenarios and for different types of faults compared to debugging without breakpoints?

RQ2 How can a concurrent synchronisation component support the manual resolution of conflicts? Which features should such a component offer?

8.1. Breakpoint Concept

To assess the value that the introduced breakpoint concept adds to the debugger, semi-structured interviews with five TGG experts were conducted in May and June 2021. The goal of the interview process was to gather feedback and suggestions for future improvements, both for specific breakpoint types and for the general handling of the debugger.

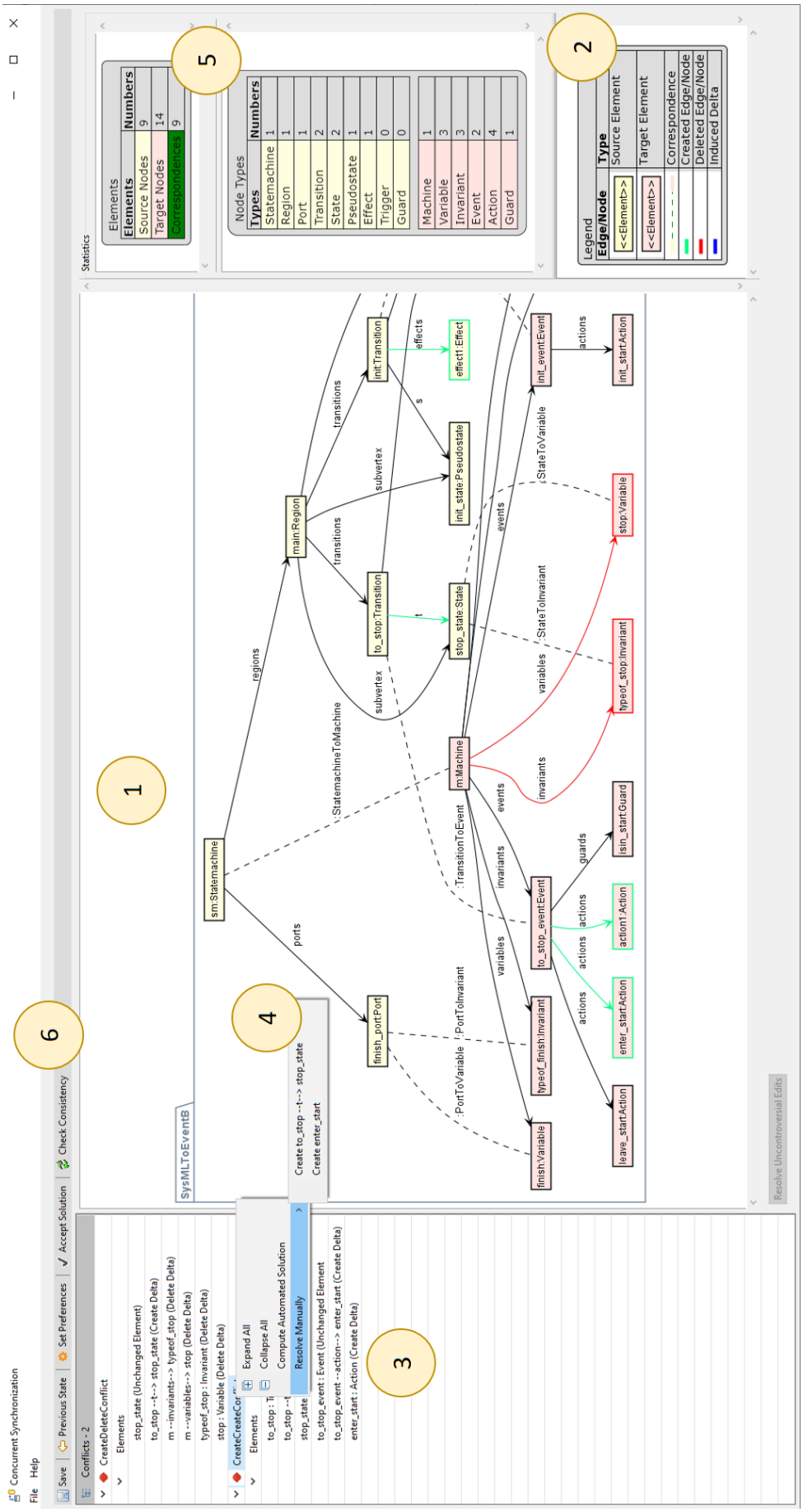

Figure 12 Front-end of the concurrent synchronisation component

Breakpoint	Assessment	Suggestions
Model size	Suitable for exploring new TGGs.	The PlantUML visualisation should be exchanged, it might crash even for medium-sized models. The scope could be restricted to a specific element type.
Number of matches	Also a suitable breakpoint for exploration purposes. Could be used to detect faults, if a rule is applied more often than expected.	
Rule name	Intuitive and easy to use. It can be used to detect the misbehaviour of a certain rule, or to skip rather uninteresting rule applications.	For larger TGGs, the selection of multiple rules would be a handy feature, e. g., by using regular expressions.
Element type	Similar to the rule name breakpoint, it is easy to use and understandable.	The user should define whether the breakpoint holds for created or translated elements (or both).
Attribute condition	Can be used when searching for specific matches or rule applications. Another typical use cases is to find faults in attribute conditions of TGG rules. While it is a powerful feature for experts, it can get confusing for novices.	A drop-down menu or an auto-completion feature could support inexperienced users in defining advanced conditions.
Combined	Enables the user to define statements of propositional logic for breakpoints. The visualisation as a tree structure is appropriate.	Plausibility checks should be added to avoid faults in the breakpoint definition itself.

Table 2 Feedback for specific breakpoints

Setup: Each interview started with a short presentation that introduced the problem along with a running example in the context of the *FamiliesToPersons* benchmark (Anjorin et al. 2017). Subsequently, each implemented breakpoint type was briefly described and demonstrated based on the tool. After each demonstration, the participants were given the opportunity to comment on typical use cases for such a breakpoint type and the usefulness in the respective context. In the third part of the interviews, a faulty version of the *FamiliesToPersons* TGG, i. e., a version with a few distorted rules, was shown to the experts. The VICToRy debugger was then used to detect those faults, such that the experts could comment on the usefulness of the debugger (including the breakpoint concept) in general.

Results: The experts' feedback concerning single breakpoint types and the debugger in general is summarised in Tab. 2 and 3. Regarding the breakpoint types, the combination of element type breakpoints and attribute condition breakpoints was perceived as very powerful. All implemented breakpoint types are useful for different purposes. The two types that were left to future work (pattern and node breakpoints) were regarded as useful additions, involving a presumably high implementation effort, though.

Also, the overall feedback for the debugger was largely positive. While the breakpoint concept is powerful and offers several configuration options, the handling of the debugger is perceived as adequate for the intended user group. In accordance with our observations in prior work (Weidmann, Anjorin, & Cheney 2020), the experts stated that the debugger is very helpful to understand *why* certain transformation steps take place, but unable to explain why, e. g., a particular rule is *not* applied.

8.2. Concurrent Synchronisation Component

To receive early feedback throughout all design phases of the component, we conducted semi-structured interviews with ten MDE experts between October 2020 and February 2021. Experts from different fields (TGGs, BX, and other sub-branches of MDE) were involved to gather requirements and assess the prototype from different perspectives.

Setup: Similar to the evaluation of the breakpoint concept, each interview started with a brief introduction to the problem based on a running example. As the interviews took place in parallel to the development process, we both incorporated feedback from earlier interviews and adapted the interviews' structure continuously. While the first five interviews were purely based on diagrams, the last five interviews included a live demo of the current state of the prototype (cf. Fig. 12).

Results: In the following, a short summary of the gathered feedback is provided, grouped by six aspects that played an important role in all interviews.

Goal: The concurrent synchronisation component should be able to list conflicting changes and provide the user with different options for resolving them. Besides resolving conflicts manually, it should be possible to start an automated resolution process, e. g., for incorporating uncontroversial changes.

Actors: All experts agreed on involving both technical and human actors into the synchronisation process. The "background operation", i. e., the synchronisation operation of the underlying tool, can be regarded as a technical actor. There were different opinions about the involved human actors: While some experts stated that one or two domain experts should use the tool (resolving conflicts collaboratively by negotiations), others regarded an integration expert as the system's key user.

Criterion	Assessment	Suggestions
User Interface	Clean and well-structured, the visualisation of rules and (partial) models, and the history of rule applications are useful.	The visualisation of translation markers would be a helpful improvement for model transformation and consistency checking operations.
Features	Sufficient for practical use. All breakpoints are helpful, especially the combination of the breakpoints for element types and attribute conditions.	Further configuration facilities could be implemented, a pattern breakpoint would be useful to examine graph constraints.
User Interaction	The interaction possibilities with the debugger are well-structured and understandable.	A handy feature would be to undo rule applications (as for the concurrent synchronisation component). This must be supported by the TGG tool as well.
Usability	Positive feedback in general. It is valuable that the main window, the configuration pop-up and the breakpoint menu can be used in parallel.	More support for the attribute condition breakpoint and another visualisation technique (cf. Tab. 2).
General	Eases the access of novices (students and practitioners) to TGG tooling. Useful for why-debugging, but not for why-not-debugging.	A quantitative user study and connections to other TGG tools could underpin the debugger's applicability.

Table 3 General feedback for the VICToRy debugger

Capabilities (main features): First and foremost, the component should provide visual support for change and conflict detection and resolution on models. A history or list of changes and resolved conflicts with time stamps should give an overview of the current state and recent actions. The tool should also be able to provide an "explanation" of the conflict in some way, and offer consistency checks to validate the results.

Resolution process: Both manual and automated conflict resolutions should be possible, resembling a "mixed-initiative approach" known from the human computer interaction domain. During the process, it should be possible to undo and redo operations, and to have a preview for the effects of possible next steps. The confirmation or acceptance of the final solution should always be done by the user. Interestingly, the expert disagreed on whether the conflict resolution should operate on the level of rule applications (undoing and redoing rule applications as in (Fritsche et al. 2020)) or user edits (performing arbitrary edit operations on graphs as in (Weidmann, Fritsche, & Anjorin 2020)). A question that remained open is how to automatically generate (understandable) descriptions of conflicts.

Additional features: Some of the suggested features were considered as handy, but not mandatory for the synchronisation process. The tool could offer a comparison of the current state and the previous state to visualise the effects of the last change. In accordance with the main debugger, an option to save intermediate model states was suggested. Furthermore, configuration options such as label abbreviation and hiding, and displaying the neighbourhood of a sub-graph (cf. Sect. 6) are perceived as helpful. Some features that could not be integrated into the prototype are search functionalities for, e. g., particular conflicts or elements, and tutorials or wizards to ease the access of novices to the tool.

Visualisation options: The visualisation of models and highlighting of conflicts is perceived as very helpful, but the experts criticised that the reordering of elements after each edit operation in PlantUML is confusing. The resolved conflicts should be marked as such to visualise the changes made during the synchronisation process. A legend that explains the visual syntax in use was considered as necessary. Some persons stated that a visual concrete syntax would be helpful for domain experts, but is hard to implement.

Overall, helpful feedback could be gathered for multiple aspects, including usability, conflict resolution strategies, and visual modelling aspects. In most cases, there was agreement between the experts, encouraging us to integrate the suggested features into the prototype.

8.3. Summary

Revisiting our research questions, the expert interviews indicate that the introduced breakpoint concept substantially eases detecting faults in either models or rules as precise conditions for pausing the transformation process can be defined (RQ1). An extension towards *why-not-debugging* facilities (Anjorin & Cheney 2019) still seems to be necessary to properly address bug finding tasks. For the interactive resolution of conflicting changes in concurrent synchronisation scenarios, requirements and feature requests were gathered in a series of expert interviews that ended up in a UI prototype (RQ2). The actual integration of the concurrent synchronisation component into VICToRy is left to future work, though.

8.4. Threats to Validity

The most striking issue of the evaluation procedure is its purely qualitative nature. For assessing the breakpoint concept and designing the concurrent synchronisation component, only 5 + 10 experts were involved in the interview process. Many of them possess expert knowledge about TGGs and have worked with the eMoflon tool suite before. The interviews followed a uniform structure, but due to the lack of a standardised questionnaire and open discussion phases, the results are not fully repeatable. As no quantitative experiments were conducted, we cannot provide hard empirical evidence for efficiency, ef-

fectiveness or user satisfaction. While we cannot generalise our results, our goal was not to measure the improvements of using VICToRy along with MDE tooling but rather to explore the design space in a realistic setting and receive early feedback from MDE experts for current features and possible extensions of VICToRy.

9. Conclusion and Future Work

We presented the add-on component VICToRy for interactively visualising single steps of different consistency management tasks. A concept for switching between different debugging modes via breakpoints is presented. Breakpoints of different types can be configured in many ways and combined to form complex breakpoint conditions. A component for interactively synchronising conflicting changes after concurrent updates was designed to address the special challenges of concurrent model synchronisation. Besides the inspection of possible rule applications in the current state, the user can retrace 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 Java-based MDE tools via a defined interface.

Carrying forward our research on user-centred consistency management, we plan to implement the two remaining breakpoint types, and to fully integrate the concurrent synchronisation component into VICToRy. While expert interviews were conducted to assess the applicability of this component, structured user acceptance tests with respect to the understandability and controllability of the consistency restoration process are left to future work. As an extension towards supporting why-not debugging, information about why rule applications are not applicable in certain situations should be presented to the user. In this way, logical faults in TGG rules or a mismatch with expectations in provided input models and tests can be detected.

Acknowledgments

This work was partially supported by the North Rhine Westphalian Ministry of Economic Affairs, Innovation, Digitalisation and Energy (MWIDE) through the Pro-LowCode project (005-2011-0022). We would also like to thank our anonymous reviewers for their helpful comments and suggestions.

References

Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., & Stevens, P. (2016). Introduction to Bidirectional Transformations. In *Bidirectional transformations* (pp. 1–28). Springer.

Abrial, J.-R., & Hallerstede, S. (2007). Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, *77*(1-2), 1–28.

Anjorin, A., Buchmann, T., & Westfechtel, B. (2017). The Families to Persons Case. In *Transformation Tool Contest (TTC) 2017, Proceedings* (pp. 27–34). CEUR-WS.org.

Anjorin, A., & Cheney, J. (2019). Provenance Meets Bidirectional Transformations. In *International Workshop on Theory and Practice of Provenance (TaPP) 2019, Proceedings*. USENIX Association.

Bousse, E., Corley, J., Combemale, B., Gray, J., & Baudry, B. (2015). Supporting efficient and advanced omniscient debugging for xDSMLs. In *Software Language Engineering (SLE) 2015, Proceedings* (pp. 137–148). ACM.

Cuadrado, J. S., Guerra, E., & de Lara, J. (2018). Quick fixing ATL transformations with speculative analysis. *Softw. Syst. Model.*, *17*(3), 779–813.

Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2006). *Fundamentals of Algebraic Graph Transformation*. Springer.

Fritsche, L., Kosiol, J., Möller, A., Schürr, A., & Taentzer, G. (2020). A precedence-driven approach for concurrent model synchronization scenarios using triple graph grammars. In *Software Language Engineering (SLE) 2020, Proceedings* (pp. 39–55). ACM.

Giese, H., Hildebrandt, S., & Lambers, L. (2014). Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars - Ensuring Conformance of Relational Model Transformation Specifications and Implementations. *Software and Systems Modeling*, *13*(1), 273–299.

Holt, J., & Perry, S. (2019). *SysML for Systems Engineering: A model-based approach*. Institution of Engineering and Technology.

Jose, J. (2021). *Large-Scale Model Transformation Debugging with Configurable Breakpoints. Master thesis, Paderborn University, Germany* [Master Thesis].

Jukss, M., Verbrugge, C., & Vangheluwe, H. (2017). Transformations Debugging Transformations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings* (pp. 449–454). CEUR-WS.org.

Klassen, L., & Wagner, R. (2012). EMorF - A Tool for Model Transformations. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, *54*.

Kolovos, D. S. (2009). Establishing correspondences between models with the epsilon comparison language. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA) 2009, Proceedings* (pp. 146–157). Springer.

Krasnogolowy, A., Hildebrandt, S., & Wätzoldt, S. (2012). Flexible Debugging of Behavior Models. In *International Conference on Industrial Technology (ICIT) 2012, Proceedings* (pp. 331–336). IEEE. doi: 10.1109/ICIT.2012.6209959

Laurent, Y., Bendraou, R., & Gervais, M.-P. (2013). Executing and debugging UML models: an fUML extension. In *Symposium on Applied Computing (SAC) 2013, Proceedings* (pp. 1095–1102). ACM.

Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., & Greenyer, J. (2014). A Comparison of Incremental Triple Graph Grammar Tools. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, *67*.

Mészáros, T., Fehér, P., & Lengyel, L. (2013). Visual debugging support for graph rewriting-based model transformations. In *International Conference on Computer as a Tool (Eurocon) 2013, Proceedings* (pp. 482–488). IEEE.

Mierlo, S. V., Tendeloo, Y. V., & Vangheluwe, H. (2017). Debugging Parallel DEVS. *Simulation*, *93*(4), 285–306.

Oakes, B. J., Lucio, L., Verbrugge, C., & Vangheluwe, H.

(2018). Debugging of Model Transformations and Contracts in SyVOLT. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2018, Workshop Proceedings* (pp. 532–537). CEUR-WS.org.

Orejas, F., Pino, E., & Navarro, M. (2020). Incremental Concurrent Model Synchronization using Triple Graph Grammars. In *Fundamental Approaches to Software Engineering (FASE) 2020, Proceedings* (pp. 273–293). Springer.

Rieke, J. (2015). *Model Consistency Management for Systems Engineering* (Unpublished doctoral dissertation). Paderborn University, Germany.

Roques, P. (2016). MBSE with the ARCADIA Method and the Capella Tool. In *European Congress on Embedded Real Time Software and Systems (ERTS) 2016, Proceedings.*

Runge, O., Ermel, C., & Taentzer, G. (2011). AGG 2.0 - New Features for Specifying and Analyzing Algebraic Graph Transformations. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE) 2011, Proceedings* (pp. 81–88). Springer.

Salunkhe, S., Berglehner, R., & Rasheeq, A. (2021). Automatic transformation of sysml model to event-b model for railway CCS application. In *International Conference on Rigorous State-Based Methods (ABZ) 2021, Proceedings* (pp. 143–149). Springer.

Schönböck, J., Kappel, G., Wimmer, M., Kusel, A., Retschitzegger, W., & Schwinger, W. (2013). TETRABox - A Generic White-Box Testing Framework for Model Transformations. In *Asia-Pacific Software Engineering Conference (APSEC) 2013, Proceedings* (pp. 75–82). CEUR-WS.org.

Schürr, A. (1994). Specification of Graph Translators with Triple Graph Grammars. In *International workshop on graph-theoretic concepts in computer science (wg) 1994, proceedings* (pp. 151–163). Springer.

Srivastava, A. (2021). *Visualization of Concurrent Synchronization Processes based on Triple Graph Grammars. Master thesis, Paderborn University, Germany* [Master Thesis].

Tichy, M., Beaucamp, L., & Kögel, S. (2017). Towards Debugging the Matching of Henshin Model Transformations Rules. In *International Conference on Model Driven Engineering Languages and Systems (MODELS) 2017, Workshop Proceedings* (pp. 455–456). CEUR-WS.org.

Troya, J., Segura, S., Burgueño, L., & Wimmer, M. (2022). Model transformation testing and debugging: A survey. *ACM Comput. Surv.*.

Ujhelyi, Z., Horváth, Á., & Varró, D. (2012). Dynamic backward slicing of model transformations. In *International Conference on Software Testing, Verification and Validation (ICST) 2012, Proceedings* (pp. 1–10). IEEE.

Voirin, J.-L., Bonnet, S., Normand, V., & Exertier, D. (2015). Model-Driven IVV Management with Arcadia and Capella. In *International Conference on Complex Systems Design & Management (CSD&M) 2015, Proceedings* (pp. 83–94). Springer.

Weidmann, N., & Anjorin, A. (2021). eMoflon::Neo - Consistency and Model Management with Graph Databases. In *International Workshop on Bidirectional Transformations (Bx) 2021, Proceedings* (pp. 54–64). CEUR-WS.org.

Weidmann, N., Anjorin, A., & Cheney, J. (2020). VICToRy: Visual Interactive Consistency Management in Tolerant Rule-based Systems. In *International Workshop on Graph Computation Models (GCM) 2020, Proceedings* (pp. 1–12). EPTCS.

Weidmann, N., Anjorin, A., Fritsche, L., Varró, G., Schürr, A., & Leblebici, E. (2019). Incremental Bidirectional Model Transformation with eMoflon: : IBeX. In *International Workshop on Bidirectional Transformations (Bx) 2019, Proceedings* (pp. 45–55). CEUR-WS.org.

Weidmann, N., Fritsche, L., & Anjorin, A. (2020). A search-based and fault-tolerant approach to concurrent model synchronisation. In *Software Language Engineering (SLE) 2020, Proceedings* (pp. 56–71). ACM.

Weidmann, N., Salunkhe, S., Anjorin, A., Yigitbas, E., & Engels, G. (2021). Automating Model Transformations for Railway Systems Engineering. *Journal of Object Technology*, *20*(3), 10:1-14.

Wimmer, M., Kappel, G., Schönböck, J., Kusel, A., Retschitzegger, W., & Schwinger, W. (2009). A petri net based debugging environment for QVT relations. In *International Conference on Automated Software Engineering (ASE) 2009, Proceedings* (pp. 3–14). IEEE.

About the authors

Nils Weidmann works as post-doc researcher at Paderborn University. He received his Ph.D. in computer science at Paderborn University in 2022. His research interests include triple graph grammars, bidirectional transformations, model-driven engineering, as well as low-code development.

Enes Yigitbas is currently an interim professor for human-computer interaction at Paderborn University. He received his Ph.D. in computer science from Paderborn University in 2019. His research interests cover model-driven engineering, human-computer interaction with a special focus on augmented-, mixed- and virtual-reality, as well as self-adaptive software systems.

Anthony Anjorin is currently a systems engineer at IAV automotive engineering. He obtained his Ph.D. degree in computer science at the Technische Universität Darmstadt in 2014. His research interests include triple graph grammars, graph transformations, bidirectional transformations, model-driven engineering, and model-based systems engineering.

Ankita Srivastava is a software developer at SDA SE Open Industry Solutions. She received a Master's degree in Computer Science degree from Paderborn University in 2021. Her research interests cover model-driven engineering (MDE), bidirectional transformations, concurrent model synchronisation, and visual debugging in MDE contexts.

Jane Jose currently works as a software developer at Diebold Nixdorf, and received a Master's degree in computer science with the focus area software engineering from Paderborn University in 2021. Her research interests include model-driven engineering, bidirectional transformations with triple graph grammars, and visual debugging.