Transformations Debugging Transformations

Māris Jukšs
School of Computer Science
McGill University
Montréal, Québec, Canada
Email: mjukss@cs.mcgill.ca

Clark Verbrugge
School of Computer Science
McGill University
Montréal, Québec, Canada
Email: clump@cs.mcgill.ca

Hans Vangheluwe
University of Antwerp
Flanders Make vzw
McGill University
Email: hans.vangheluwe@uantwerp.be

***Abstract*—Practical model transformation (MT) frameworks are usually composed of multiple execution contexts, building an overall workflow by abstracting different aspects of execution. This heterogeneity brings additional challenges to debugging, which must address a combination of quite different graphical and textual formalisms. In this work we describe a layered approach to debugging, mapping familiar debugging operations to different formalisms, as well as the transitions between them. Our design allows for seamless debugging through different abstractions, and supports both traditional imperative debugging as well as declarative, query-based approaches. We demonstrate our approach by prototyping a MT debugger in the *AToMPM* research tool. Our approach shows that it can be applied to other MT tools as well.**

I. Introduction

Debugging Model Transformations (MT) brings additional complexity due to the presence of a hierarchical execution stack. Multiple execution layers are used, consisting of different languages/formalisms at different levels of abstraction. A schedule language, for example, contains the rules, which themselves hold patterns based on action code. Further down the stack, we find the pattern matching and application routines. As problems can manifest at each layer, or between layer interactions, thorough debugging requires a tool permitting inspection and modification throughout the execution stack.

In this work we describe the design of a MT debugger that addresses debugging of the whole MT stack, from the schedule level down to the pattern matching and application routines details. Our design avoids resorting to code-level debugging, aiming instead to remain at the level of abstraction similar to the Domain-Specific Language (DSL) of the models being transformed.

The declarative nature of models and MTs has a further benefit in that it is a natural setting for many debugging tasks, particularly event or watch-based goals, such as pausing execution when a pattern matcher accumulates a certain portion of the match, or when an undesirable pattern appears in the output model. In this context, *query-based* debugging techniques [1] yield further inspiration for our debugger design, allowing us to inspect all the relevant MT stack levels with declarative queries during the debugging session.

Our goal is to provide a practical and flexible solution for MT debugging for modern tools. We here show an initial validation of our design by describing our experience with realizing such a debugger in *AToMPM*, a research-based MT tool. Specific contributions of our work include the following.

- We describe a structured view of a debugging process that lends a unified way of navigating the debug target.
- We describe a simple language for automated debugging of MTs using declarative queries and breakpoints spanning the levels of the MT stack.
- Our declarative approach subsumes a separate control scheme for direct user interaction. In this way our design provides a unified debugging model, suitable for automation.

II. Structured View of Debugging

In this section we present a general view of debugging in a structured way. We will use this approach in order to have a uniform operational semantics for debugging in situations where a general programming language paradigm may be less applicable, such as in debugging DSLs and in particular MTs. In particular, this lets us clarify the notion of basic debugger operations, such as *step*, which may be ambiguous in declarative debugging contexts.

Navigating the Debugging Target. To take a generic view of debugging control flow (process) we can imagine it to be somewhat similar to navigating the multi-story building representing the debugging target. On the vertical dimension (here we use the term without adhering to its strict definition) we have a multitude of floors representing the many levels of nested structures. These hierarchical levels can be found in nested function calls, hierarchical models, etc. We call these nested levels **vertical levels** (*VL*) and the movement between them as **vertical movement**.

Each vertical level also has a horizontal dimension, this can be imagined as the apartments on a floor. If we use the programming language analogy, these levels represent a single kind of scope. We call this horizontal dimension a **horizontal level** (*HL*) and the movement within that level as **horizontal movement**. Each horizontal level contains items of interest for the debugging. These can be statements, expressions, nodes and edges in the model. The unifying characteristic of these items is the fact that when the debugging target has processed, visited or executed one item, it will move to the next item—items on a horizontal level are dynamically enumerated as a result of executing general step-over operations on the debug target. In Figure 1, we show the horizontal and vertical

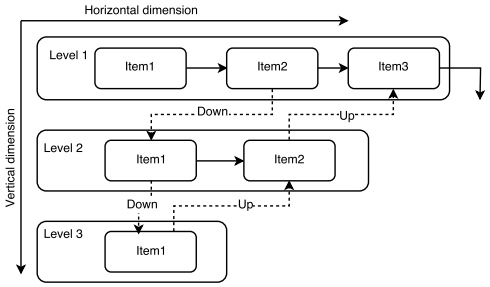

Fig. 1. Horizontal and vertical dimensions or levels. Arrows between items on a horizontal level represent horizontal movement operation. Vertical operations are dashed arrows and labeled.

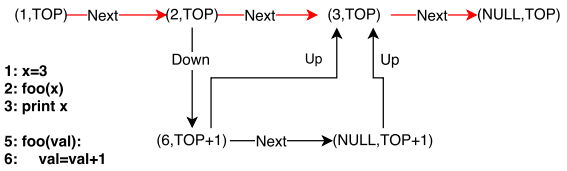

Fig. 2. Navigation pointer evolution. Red arrows represent a step-over debugging scenario.

dimensions[1] of a debugging control flow. Some of the items on the horizontal level can contain other items (nested) and therefore lead to lower vertical levels. For example, these can be hierarchical model elements or function calls to descend into. Therefore, the vertical dimension in a debug process is explored in moving up and down between the hierarchical items (as illustrated by the dashed arrows in Figure 1).

Figure 1 gives us a conceptual graph structure reminiscent of the general forms of stack-based program execution. Each vertical level embodies a particular behavior, execution scope, or level of abstraction. The horizontal elements then correspond to the smallest units of processing, computation, execution, or specification within that level of abstraction. Program execution, whether procedural or MT-based, can then be understood in terms of navigating this 2D hierarchy.

Navigation Pointers. For our purposes we need to be aware of the control flow position during debugging with respect to the horizontal and vertical dimensions. Horizontally we are concerned with the *item pointer* (*IP*) and vertically with the *level pointer* (*LP*). We can then use the navigation pointer tuple (*IP,LP*) to describe the control flow position in the debugging target, similar to the position of a point on a plane, described by its two coordinates. We also define certain constant values for the navigation pointers. These allow us to refer to the typical positions and the debugging situations within the target.

The following are the constant values the *IP* can take.

- *NULL* - this value indicates that the pointer is not initialized and is not pointing to any item on the *HL*. This value is useful in describing the situation when the *IP* moves past the last item on the *HL*.
- *FIRST* - the first item of interest on the *HL*.

The following is the constant value the *LP* can take.

- *TOP* - the very top *VL* in a debugging target.

Taking the above values into account, the end of the debugging (termination) for example, can be specified with a tuple (*NULL,TOP*).

Operations. We now propose the operations that allow us to navigate the vertical and horizontal dimensions in a debugging target. Operational semantics essentially follows a controlled stack-based traversal, giving us operations to move between (ordered) siblings, into child nodes, or back to parent nodes. The effect of these operations includes the modification of *LP* and *IP* pointers, and presumes an execution stack of horizontal item pointers, σ. We begin execution at (*FIRST,TOP*), with σ empty.

- ***Next***. This operation processes the current item on a *HL* and move on to the next item at the same level. Given non-*NULL* values for (*IP*,*LP*) and stack σ,

$$Next((IP, LP), \sigma) = \begin{cases} ((IP+1, LP), \sigma) & \text{if } \exists\ IP+1 \\ ((NULL, LP), \sigma) & \text{if } \nexists\ IP+1 \end{cases}$$

  Given (*NULL*,*LP*),σ, a *Next* operation delegates to an *Up*.
- ***Down***. This operation moves one vertical level down if possible, pushing the current state and setting the *IP* to the first item on the next level. If no deeper level exists from this item, this delegates to a *Next* operation.

$$Down((IP, LP), \sigma) = \begin{cases} ((FIRST, LP+1), IP{:}\sigma) & \text{if } \exists\ LP+1 \\ Next((IP, LP), \sigma) & \text{if } \nexists\ LP+1 \end{cases}$$

- ***Up***. This operation completes processing of all remaining items on the current horizontal level and move one level up vertically. This is idempotent, and implies terminating the program if it attempts to ascend past *TOP*.

$$Up((IP, LP), \sigma) = \begin{cases} ((IP', LP-1), \sigma') & \text{if } LP \neq TOP \wedge\ \sigma = IP'{:}\sigma' \\ ((NULL, TOP), \emptyset) & \text{if } LP = TOP \end{cases}$$

We can now describe the program execution in terms of the navigation pointer evolution. Using a simple textual example, Figure 2 demonstrates a graph of possible pointer values in the nodes and the operations that result in the changes as edges. In this example the items were the statements of the program and were identified by the line number. At each point a debugger may move to the next statement at a given level, either as a typical *step-into* (black *Next* arrow) or *step-over* (red arrow) any lower levels (the latter being *Down* operations that delegate to *Next*). An actual *Down* operation can be performed on the method call to enter the method body, at which point an *Up* operation can be requested to complete execution, skip debugging the method body and return to the caller, or *Next* can be used to flow through the execution of the increment statement, and *Up* executed when no more horizontal execution is possible.

Debugger behavior is of course not entirely addressed by this control flow model. We also need to consider how and when a debugger accesses data. Global, static data is universally available, but access to other, local data can depend on the language semantics given by the position in the control flow (such as with local, stack variables in a procedural

[1]In this paper we use the terms level and dimension interchangeably.

language). As this depends on the language being debugged, we will require the target-language context provide a means to expose (and represent) data, given a (current) navigation state, allowing the debugger read and write access in accordance with the expected semantics.

III. Structured View of MT Stack

In this section we apply our structured view of debugging to a MT stack typically found in rule-based MT systems.

Usually, in MT debugging we are mainly concerned with the MT specification, its use of the source model and the final effect on the target model. Inside the transformation specification we can discover the hierarchical structure of the schedule encompassing MT rules. Further down, we find individual patterns contained within the pre/post-condition parts of the rule. These individual patterns are used for matching in the source model and modifying the target model.

In Figure 3 we outline a conceptual, level-based view of the MT stack. Rectangles represent the components such as static models and dynamic routines. Nesting relationship represents hierarchy or containment. There is a clear separation of data where the model artifacts are concerned. The data found inside the operational semantics of related components, however, such as the matcher used for patterns, is not clearly distinguishable on the diagram. We will clarify this concern below. In the following paragraphs we investigate how the MT stack fits within our debugger.

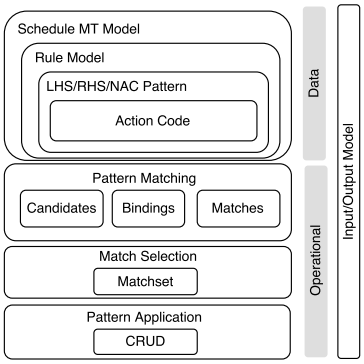

Fig. 3. A MT stack view. Nesting of boxes represents hierarchy.

Input/Output Model Level. Shown on the right of Figure 3, this is the main data part of a model transformation. In our prototype example we perform in-place transformations, and therefore we expose that single model as the global data accessible to the debugger. Of course the model may contain sensitive data, to which the debugging target may want to limit access, depending on requirements.

Schedule/Rule Model Level. The MT specification describes the control flow of the MT execution. This level is the heart of the debugging target. The nested structure of MT specification is giving us hints to the vertical and horizontal dimensions for the debugging. Note that schedule also represents data for our debugger to query. The MT rule contained in the schedule typically consists of *LHS* and *RHS* parts (with an optional *NAC*), each in turn containing patterns. We need to decide on the *VL* and *HL* items. We do this by analyzing the possible movement in horizontal and vertical dimensions. In this case, in terms of debugging we would navigate horizontally from a rule to a following rule in the schedule, as the rules are processed at the same level of abstraction. The actual rule transformation is at a different level of abstraction from rule scheduling, and represents a descent into a deeper, vertical level, wherein there is horizontal movement between the process of applying first the *LHS*, and then the *RHS* parts of the rule.

The execution of the *LHS* and *RHS* parts contains further, nested execution complexity. That is the presence of some action code (AC) used to specify imperative constructs otherwise too complex to express declaratively. The model element attribute evaluation is one example of AC use. Treatment of AC requires a context switch in interpreting the MT specification. Execution semantics depend on the action language, and so requires a formal view of the language in terms of the debugger navigation pointer values, or would need to relay to the underlying general purpose language debugging facilities. We utilize those facilities in our prototype evaluation to deal with AC.

In summary, for this part of MT stack, the *VL* can be obtained by exploring the containment relationship in the MT schedule presented. If the schedule is presented in textual format the hierarchical relationship could also be explored by descending into the function/procedure calls. In turn, a horizontal dimension of each *VL* is exposed by enumerating the items without exploring the hierarchy, as in the case of the *LHS* and the *RHS* parts of one rule.

Pattern Matching Level. Going a level down into the pattern matching process we necessarily encounter a tool specific implementation. We unify pattern matching through a generic, algorithm-agnostic view of the major steps involved. Generally, a pattern matcher must first find a set of *candidates* for constructing bindings for the pattern elements. Candidates that are successfully bound form a set of *bindings*. A complete set of bindings that matches the input pattern is called a *match*, and a set of valid matches constitutes a *matchset*. Movement in the horizontal dimension within these stages happens by iterating over the corresponding set items. Finally, the sets can also be considered as data that we can query during debugging. Note that non-determinism may be present in terms of which match is actually selected for pattern application from the matchset. This can be approached simply as exposing the selected match.

Pattern Application Level. For the pattern application process in the *RHS* part of the MT rule we also need unified execution concepts, as again this design can be quite implementation specific. As a general solution we rely on basic Create, Read, Update, Delete (CRUD) operations affecting the input model, and define queryable sets related to each one (except for the Read, in this context, as it belongs to the matching domain). Our interpretation of the CRUD operations is based on the unique labels on pattern elements used to assist

in identifying elements meant to be created or deleted. The navigation in the horizontal dimension here is approached just as in the case of pattern matching.

IV. Debugging Language

Our debugger design builds on a custom debugging language expressed through *debugging rules,* which follow the familiar MT rule structure, including *LHS* and *RHS* parts. This approach allows us to incorporate domain-specific syntax for different layers. Debugging rules can also be chained to form *debugging scenarios*, which can be executed separately from a target MT, facilitating automated debugging.

Querying. We embed debugging operations into MT rules based on the fact that the *LHS* of a MT rule is a pre-condition for the application of the *RHS*. The *LHS* patterns are matched in the input model, essentially acting as queries over the debugging target state. These queries can include inspecting the input/output model, as well as the MT specification. The MT problem domain formalisms are then reused in the debugging rule without the need for any extra effort on the part of an engineer.

The data involved in the pattern matching and application process can also be the target of queries. For example, a match containing the bindings between the pattern and the input model or any other such data can be exposed to the debugger. In this context we also reserve certain keywords for the match sets described in Section III for more precise querying.

In order to reason about the location within the debugging target we may need to form a query based on the navigation pointer pair, or some part of it. This is useful in order to perform an action when the MT control flow enters a desired location in the MT specification, such as an execution of a particular rule. Due to variability in specification of navigation pointers and pattern matching/application related data we may need to resort to the use of AC instead of declarative queries.

Action. After successful query discovery (or simply a pre-condition satisfaction), we want to perform an action. For this, we use the *RHS* of the rule, specifying traditional debugger actions, as well as modification of the various parts of the query domains. We focus mainly on the former in this work, but many other effects of the *RHS* action are possible, including modifications to the input/output and the MT specification models. The latter, for example, allows us to perform the *adaptation* of MT specification for the exploration of new execution scenarios. We may also want to influence the pattern matching and application process. More detailed investigation of execution adaptation and pattern matching/application influence represents an advanced debugging session, which we leave for future work.

Navigation Commands. One of the goals of our debugger is to control the execution of the program by means of issuing navigation commands. To issue basic debugging target navigation commands we embed them within the *RHS*. Application of the rule, and successful matching of the *LHS* pattern then results in the command being performed. Such debugger commands can be simply issued through the use of AC. A visual representation, however, better fits the MT paradigm and we show a possible iconic representation of a *Next* command in Figure 4. In terms of common debugging parlance, the *Next* operation represents *step over*, while *step into* maps to *Down*, and the *Up* operation maps to the *step out* operation.

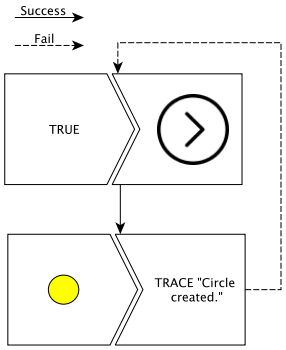

Fig. 4. A debugging scenario that steps through the MT execution until the circle pattern in found resulting in a trace message.

An example of usage is shown in Figure 4. In this automated debugging scenario, we want to step through execution until a given query (detecting a circle node) is satisfied. The scenario works through two rules, stepping through the execution by issuing *Next* commands (with an unconditional query) until the creation of a circle in the input model can be detected by a second rule. Rule scheduling applies the first rule (which always succeeds), then the second, returning to the first rule when the second fails to apply. Upon success of the second rule a debugging action is performed, in this case a trace is done using action code. The pace of debugging is here given by the use of a *Next* operation, which depends on the navigation pointer position at the time the command is issued; it may also be desirable to step execution at finer granularity, such as by using a *Down* operation in the first rule to model a fine-grain (step-into) execution. Note, that given the many sources of data described in this paper, the second rule in Figure 4 is ambiguous in terms of which input model is applicable to the query, this discussion we omit for brevity.

The second rule in Figure 4 represents a slight semantic departure from typical MT rule design. In traditional MT rule design, the absence of the *LHS* pattern in the *RHS* of Figure 4 indicates that the occurrence of the *LHS* pattern should be deleted (as well as the trace action performed). As this is not typically the combined intent of a debugger action, we assume that when a navigation command is present in the *RHS* the rule becomes read-only, and the occurrence of the query present in the *LHS* will not be modified. When modifications to the occurrence of a query in combination with navigation is desired, the user will need to perform this action with two rules in sequence, one to perform the modification, and another to issue the navigation command.

Other debugging actions are of course possible. A pause action, for example, could be performed (when the debugging

target is running continuously) instead to realize a breakpointing functionality. In this case, the debugger still needs to ensure that the target state does not change during query evaluation. Depending on a query, we may need to implicitly pause the target's execution on every, fine-grained navigation pointer change before each query evaluation. Additional, higher level syntactic constructs can be used to encapsulate verbose debugging scenarios, such as one in Figure 4, for convenience.

V. Evaluation in AToMPM

Evidence of the utility and practicality of our design is given by a sample implementation in AToMPM [2], a browser-based Tool for Multi-Paradigm Modeling.

The heart of our debugger is implemented using the well-known *Statecharts* formalism [3]. *Statecharts* were chosen because of their convenience in describing autonomous, concurrent, and reactive systems. The inspiration for our solution comes from work by others on reimplementing existing model execution engines in *Statecharts* with the addition of a debugger related functionality [4]. We chose a different path and implement the main logic in a central *Statecharts* model (Python-based). This model deals with the navigation commands, receives navigation pointer values from the debugging target, and permits the target to advance the execution. In Figure 5 we demonstrate the modified version of the actual *Statecharts* generated from importing the SCXML file [2] into QTCreator's [3] *Statecharts* editing facility. The navigation pointer events

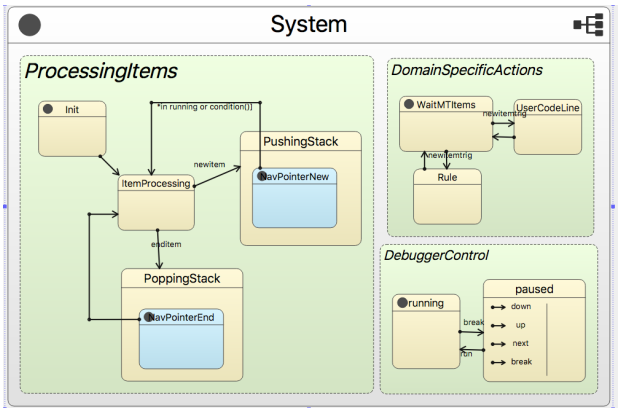

Fig. 5. A *Statecharts* model of the debugger. Orthogonal components are responsible for processing navigation commands, navigation pointers, and implementation specific items corresponding to navigation pointers.

from the debugging target are processed in the *ProcessingItems* state. The debugging target is then allowed to proceed with the execution in the transition from the *NavigationPointerNew* state according to the guard condition in the particular runtime context. This context is maintained in the *DebuggerControl* state, where we process the navigation commands. Finally, the navigation pointer changes can also carry information such as rule names, action code line numbers, etc. These are processed in the separate parallel state called *DomainSpecificActions*. Here we can perform such actions as visualization, highlighting or ensure that the data is shared properly between components. This state is intended to be more implementation specific as opposed to other states aiming to be as generic as possible. In fact, this *Statecharts* model was reused almost without modification in another MT tool, AToM^3 [5] to specify the debugger. AToMPM and AToM^3 implementations share the *Statecharts* model's compilation target language Python.

We now discuss the tool specific changes necessary to communicate with the *Statecharts* model described above. We identify all locations in the Python back-end code that are relevant to the changes in navigation pointers. These places are mainly processing the schedule, individual rules, the *LHS* followed by the *RHS*, down to relevant action code evaluation and pattern matching/application routines. From each new navigation pointer location we send events to the *Statecharts* model and the MT tool immediately waits on a response from the *Statecharts* to proceed. This trivially enables pausing functionality. If the *DebuggerControl* is in the running state the execution may or may not be interrupted, according to the navigation commands.

The action code treatment is approached utilizing the AC's language facilities. In this case the AC is Python, which provides an interface to develop custom Python debuggers called *BdB*. By extending the class we can process Python code specific events and initiate communication with the *Statecharts* model to announce the navigation pointers change. The navigation pointers change on events when the control flow descends into the function or the next line/statement is processed. The navigation pointers can carry line numbers that we can highlight in the action code as demonstrated in Figure 6. In addition, we can inspect variables. In the similar

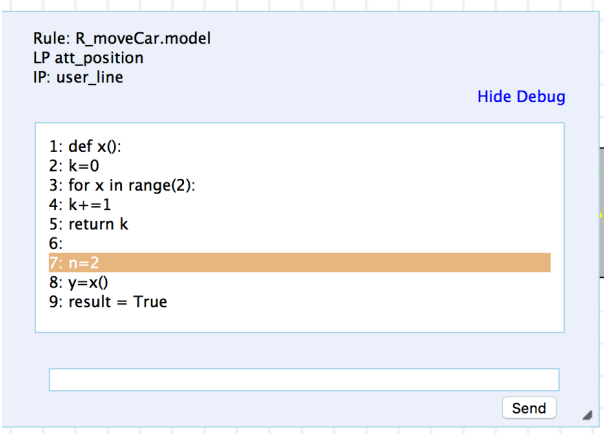

Fig. 6. A screenshot of our tool and processing of action code

fashion of dealing with AC, we envision dealing with the MTs specified entirely in AC (a loop calling a function representing the MT rule for example).

Finally, our design supports automated debugging, but also

[2]https://www.w3.org/TR/scxml/
[3]https://www.qt.io/ide/

manual, user-triggered debugging actions. We can have a debugging rule for each navigation/debugging operation, and we simply model the button press as the execution of a debugging rule (and essentially a MT) with a required action.

Efficiency Considerations. It is evident that the context switching from the MT to the debugger MT will impact performance. Generally, query evaluation in debugger rules is based on the same computationally expensive pattern-matching problem as in general MTs. A comprehensive performance evaluation is well beyond the scope of the basic design we introduce here (or our initial prototype).

VI. Related Work

A variety of approaches exists for framing debugging designs. A prominent approach is to follow an *event-based* view of a debugging process. One example is found in the moldable debugger (MD) [6]. This flexible approach is based on several primitive debugging events resulting from a debugging target operations combined to produce domain specific events/operations. This approach can be used to implement the event-based part of our debugger. We, however, take a more structure-focused perspective, considering debugging as a process of navigating execution within and across hierarchical levels.

Query-based debugging from the general purpose language domain was used as an inspiration in this paper [1], [7], [8]. *Whyline* debugger [9] generates useful queries that can be applied to the recorded program execution traces. Generic query designs can of course bring performance concerns. EMF-IncQuery tool [10] performs efficient, incremental declarative query evaluations for MT verification. We can potentially utilize similar incremental pattern matchings to improve the performance of our query evaluation.

The GDL debugging language [11] defines debugging operations that are embedded into a general programming language, and complex debugging scenarios can then be specified programmatically. Similarly, we embed the MT debugging language elements into the MT language. This allows us to use branch and loop constructs and create user specified transformations for the purpose of debugging.

MT and model debugging have of course been explored in the past. AToMPM, for instance, already supports MT debugging at the level of the MT schedule and down to individual rules, as previously described [12]. Other MT debugging solutions have also been described [13]–[16].

VII. Conclusions and Future Work

In this chapter we explored the design of a MT transformation debugger based on model transformations themselves. The debugger allows for specification of debugging scenarios aiming at discovering complex MT execution artifacts, using the syntax and semantics of MTs. This reduces the learning curve as the user is operating within the familiar domain of MTs and reuses existing DSLs. The advantage of a debugging scenario, just like the MT itself, is that it can be left to run unattended and perform the desired tasks. A modeled solution to debugging has other benefits as well. Debugging scenarios can be exchanged between engineers, reused and analyzed.

Our approach was further based on a structured view over the general debugging process. This allows us to bring clarity into the notion of a step in the declarative context of model transformations. Our non-trivial prototype implementation, based on this view, allows us to evaluate the feasibility of our debugger and demonstrate the treatment of AC as well.

For future work we look to address the performance evaluation of debugging scenarios and their in-depth usage. We expect however, that the ability to debug model transformations in a way presented in this paper may outweigh the runtime effects on the whole system.

The authors would like to thank Simon Van Mierlo for his insight into *Statecharts*-based implementation of debuggers.

References

[1] R. Lencevicius, U. Hölzle, and A. K. Singh, "Query-based debugging of object-oriented programs," in *Proceedings of the SIGPLAN OOPSLA 1997*. ACM, pp. 304–317.

[2] R. Mannadiar, "A multi-paradigm modelling approach to the foundations of domain-specific modelling," Ph.D. dissertation, McGill University, 2012.

[3] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[4] S. Van Mierlo, "Explicitly modelling model debugging environments," in *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*, 2015, pp. 24–29.

[5] J. de Lara and H. L. Vangheluwe, "Using AToM^3 as a meta-CASE environment," in *4th International Conference On Enterprise Information Systems*, 2002, pp. 642–649.

[6] A. Chiş, T. Gîrba, and O. Nierstrasz, "The moldable debugger: A framework for developing domain-specific debuggers," in *SLE 2014, Västerås, Sweden, Proceedings*. Springer International Publishing, pp. 102–121.

[7] A. Potanin, J. Noble, and R. Biddle, "Snapshot query-based debugging," in *2004 Australian Software Engineering Conference. Proceedings.*, 2004, pp. 251–259.

[8] R. Lencevicius, "On-the-fly query-based debugging with examples," in *Proceedings Fourth International Workshop on Automated Debugging*, 2000.

[9] A. J. Ko and B. A. Myers, "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proceedings of the ICSE 2008 Leipzig, Germany*. ACM, pp. 301–310.

[10] M. Búr, Z. Ujhelyi, Á. Horváth, and D. Varró, "Local search-based pattern matching features in EMF-IncQuery," in *ICGT 2015, L'Aquila Italy, Proceedings*, pp. 275–282.

[11] R. H. Crawford, R. A. Olsson, W. W. Ho, and C. E. Wee, "Semantic issues in the design of languages for debugging," *Comput. Lang.*, pp. 17–37.

[12] R. Mannadiar and H. Vangheluwe, "Debugging in domain-specific modelling," in *Proceedings, SLE 2010 Eindhoven, The Netherlands*. Springer-Verlag, pp. 276–285.

[13] R. T. Lindeman, L. C. Kats, and E. Visser, "Declaratively defining domain-specific language debuggers," in *Proceedings GPCE 2011 Portland, Oregon USA*. ACM, pp. 127–136.

[14] L. Geiger, "Model level debugging with Fujaba," in *Proceedings of 6th International Fujaba Days 2008*.

[15] T. Mészáros and T. Levendovszky, "Visual specification of a DSL processor debugger," in *Proceedings OOPSLA Workshop on Domain-Specific Modeling 2008*, Nashville, USA, pp. 67–72.

[16] M. Lawley and J. Steel, "Practical declarative model transformation with tefkat," in *Satellite Events at the MoDELS 2005 Conference: MoDELS 2005 International Workshops Doctoral Symposium, Educators Symposium Montego Bay, Jamaica, October 2-7, 2005 Revised Selected Papers*, J.-M. Bruel, Ed. Springer Berlin Heidelberg, pp. 139–150.