Debugging in Domain-Specific Modelling

[†] Raphael Mannadiar and [†,‡]Hans Vangheluwe

Modelling, Simulation and Design Lab

† McGill University
3480 University Street
Montréal, Québec, Canada

‡ University of Antwerp
Middelheimlaan 1
2020 Antwerpen, Belgium

Abstract. An important obstacle to the wide-spread adoption of model-driven development approaches in industry is the lack of proper debugging facilities. Software debugging support is provided by a combination of language and Integrated Development Environment (IDE) features which enable the monitoring and altering of a running program's state. In Domain-Specific Modelling (DSM), debugging activities have a wider scope: *designers* debug model transformations (MTs) and synthesized artifacts, while domain-specific *modellers* debug their models, unaware of generated artifacts. This work surveys the state-of-the-art of debugging in the context of DSM and proposes a mapping between debugging concepts (e.g., *breakpoints*, *assertions*) in the software and DSM realms.

1 Introduction

DSM allows domain experts to play active roles in (software) development efforts. It provides them with means to manipulate constructs they are familiar with and to automate the error-prone and time-consuming conceptual mapping between the (often very distant) problem and solution domains. Empirical evidence suggests increases in productivity of up to one order of magnitude when using DSM and automatic artifact synthesis as opposed to traditional code-driven development approaches [9, 5].

MTs[1] are used to specify the semantics of Domain-Specific Languages (DSLs) by defining interpreters or by mapping onto formalisms whose semantics is well understood such as Petri Nets, Statecharts or code. In this work, we focus on rule-based approaches, where MTs are composed of rules, each parameterized by a left-hand side (LHS) and right-hand side (RHS) pattern, an optional negative application condition (NAC) pattern, condition code, and action code. The LHS and NAC patterns respectively describe what sub-graphs should and should not be present in the source model for the rule to be applicable while the RHS pattern describes how the LHS pattern should be transformed by its application. Further applicability conditions may be specified within the condition code while post-application actions may be specified within the action code.

[1] See [2] for a detailed feature-based classification of existing MT techniques.

R. Mannadiar and H. Vangheluwe. Software Language Engineering., pp. 1–10, 2010.
© Springer-Verlag Berlin Heidelberg 2010

The typical work flow of a DSM project consists of the specification of one or more DSLs and of the MTs that define their semantics. Subsequently, Domain-Specific models (DSms[2]) are created. In practice, DSms, MTs, and synthesized artifacts may all require debugging. In Section 2, we survey the state-of-the-art of debugging in the context of DSM. In Section 3, we review common debugging concepts such as *breakpoints* and *assertions* from the programming world, which we map onto the DSM world in Section 4. The contributions of this work are our mapping of debugging concepts from the programming to the DSM world, and the demystification of the amount of effort required to produce DSM debuggers.

2 Related Work

Little research has focused on debugging in DSM. In the numerous industrial applications of DSM presented in [9, 5], the debugging of DSms, of their associated ad hoc generators, and of the synthesized artifacts is always accomplished without tool support and is performed at the code rather than the DSm level, dealing only with artifacts and modelling tool APIs. Conceptually, this equates to debugging compilers and bytecode to find and resolve issues in a coded program.

Wu et al. presented the most advanced DSm debugger to-date in [13]. By transparently building a detailed mapping between model entities and synthesized code, they allow the creator of textual DSms to re-use Eclipse's debugging facilities (e.g., setting breakpoints and stepping in DSms) without being exposed to the synthesized code or its generator. The main advantage of this technique is perhaps that it avoids the implementation of a brand new debugger. However, it is limited to textual DSLs, restricts the modeller to the Eclipse tool, assumes that generated artifacts are code, and does not consider MT debugging.

In [7], we laid the basis for extending Wu et al.'s work to visual DSLs. MT rules are instrumented such that *backward links* (or "traceability links") are maintained between constructs at different levels of abstraction during artifact synthesis. These links enable DSms to be animated and updated in real-time as their corresponding synthesized artifacts are executed. Means to further exploit these links to ease DSm and artifact debugging will be explored in Section 4.3.

Certain tools (e.g., AToM^3 [3]) enable basic MT debugging by allowing rule-by-rule execution, manual intervention when multiple rules are simultaneously applicable (across multiple matches), and model modification between rule applications. Thus, the rule designer can observe the effects of each rule in isolation and stear the MT. However, advanced functionality such as pausing the execution when arbitrary rules or patterns are encountered are not natively supported.

The inclusion of exceptions within Model Transformation Languages (MTLs) is proposed in [11]. A control flow environment for rule execution is extended

[2] We refer to domain-specific modelling and model as DSM and DSm, respectively.

with provisions for specifying exceptions (and their handlers). The technique for the modelling of interruptions by exceptions can readily be extended to support interruptions by debugging commands, as we will discuss in Section 4.1.

3 Debugging Code

Several authors have looked into common sources of bugs, what makes some of them more insidious than others, and popular debugging activities [4, 14]. It seems observing system state and hand-simulation are often used for locating and resolving bugs. Means to carry out these activities are thus commonly found in modern programming languages and IDEs. Below, an overview is given of common and useful debugging facilities featured in languages and IDEs.

Print Statements Print statements are commonly used to output variable contents and verify that arbitrary code segments are executed.

Assertions Assertions enable the verification of arbitrary conditions during program execution. They cause the execution to be aborted when their condition fails, and compilers provide means to enable and disable them – as opposed to manual removal – to avoid undesired output or computation.

Exceptions Exceptions are *thrown* at runtime to indicate and report on problematic system state. They may halt the execution of a program or be *caught* by *handlers* which take appropriate action. Provisions for defining new types of exceptions enable support for application-specific exceptional situations.

The language primitives above are traditionally used to create a "poor man's debugger". The facilities below are commonly provided by modern IDEs.

Execution Control Modern IDE debuggers support continuous and line-by-line execution, as well as terminating and non-terminating interruption via "play", "step", "stop" and "pause" commands respectively. There are usually three step commands: *step over*, *step into* and *step out* (or *step return*). The first atomically executes the current statement. The second executes one sub-statement contained within the current statement, if any, thus effecting a change in scope. Advanced debuggers support stepping into seemingly atomic constructs into their corresponding lower-level representations, if any (e.g., Eclipse supports stepping through Java bytecode). Stepping out causes continuous execution until the current scope is exited. Finally, most IDEs allow running code in release (as opposed to debug) mode leaving only the play and stop commands enabled.

Runtime Variable I/O IDE debuggers usually provide means to read (and change) global and local variables when the program's execution is paused.

Breakpoints Breakpoints are commonly set on statements indicating that program execution should be interrupted before running the said statements.

Stack Traces Visible when the execution is paused, stack traces display which function calls led the program into its current state and enable navigation between the scopes of any level in the call stack for debugging purposes.

4 Debugging in DSM

The development process in DSM has two important facets: developing models and developing MTs[3]. These facets introduce two very important differences between the programming and DSM worlds. First, in the latter realm, artifacts to debug are no longer restricted to code and include model transformations, synthesized and hand-crafted models, and other arbitrary non-code artifacts. Second, the counterparts of the common DSM activities of designing and debugging MTs (i.e., designing and debugging code compilers/interpreters) are both specialized activities in the coding realm. This section explores how the previous concepts translate into the debugging stages of both facets of DSM development.

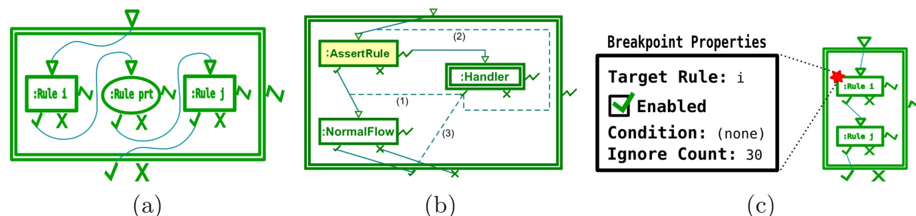
4.1 Debugging Transformations

To carry meaning and be more than metamodels for blueprints, DSLs need associated semantics which the Model-Driven paradigm dictates should be specified as MTs [1]. Operational semantics can be specified as a collection of rules each describing the transformation between valid system states. Executing these rules effectively executes the model. Denotational semantics define the meaning of a DSL by mapping its concepts onto other formalisms for which semantics are well defined. This mapping is often encoded within code generators that transform DSms to code[4] [9, 5]. We demonstrated in [7] that modelled mapping onto intermediate modelling formalisms (as opposed to directly onto code) is modular, adheres closely to the Multi-Paradigm Modelling philosophy [3], and considerably facilitates debugging by easing the specification, display and maintenance of backward links between models and synthesized artifacts. For both operational and denotational semantics, the resulting transformation model describes a flow of rule applications which may require debugging. Below, we re-visit the debugging concepts described earlier and translate them to MT debugging.

Print Statements Print statements for MTs could be emulated by creating rules with output function calls in their action code. This entails accidental complexity: LHS and RHS patterns need to be identical to avoid modifying source models which in turn implies the necessity of loop prevention action code and/or NAC patterns. A more domain-specific solution is to enhance MTLs

[3] Describing a formalism's operational or denotational semantics.

[4] Non-code artifacts such as XML, documentation and models may also be synthesized.



(a) (b) (c)

Fig. 1: (a)Print rule `prt` inserted between traditional rules i and j; (b)Resuming, restarting or terminating after handling an assertion; (c)A breakpoint on a rule.

with printing functionality by introducing *print rules* that could be sequenced with other rules, as shown in Fig. 1a. Print rules would be parameterized by a pattern, condition code and *printing code*. Their natural semantics would be to execute the printing code if the condition code is satisfied when the pattern is found in the host model. Automatic synthesis of the contrived traditional rules described above could trivially be achieved from print rules using higher-order transformation rules[5] thus leaving the transformation execution engine unchanged. Although it may seem odd to support a language construct whose usefulness is mostly restricted to debugging purposes, we should remember that print statements (whose usefulness is mostly restricted to debugging purposes) are supported in every modern GPL.

Assertions Assertions in transformation models can be emulated similarly to print statements: their conditions encoded in rule condition code or patterns, and exception throwing calls in rule action code[6]. This entails the same accidental complexities as those mentioned for print statements. A similar solution applies: enhancing the MTL with *assertion rules* – parameterized by a pattern, condition code and *assert code* – that could also be sequenced arbitrarily with other rules. Analogous reasoning applies regarding their semantics, their automatic translation to traditional rules and the validity of their inclusion in MTLs.

Exceptions Exceptions and their handlers in the context of model transformation debugging were extensively studied in [11]. Syriani et al. provide a classification of several common exceptions and support user-defined exception types. They propose enhancing MTLs with *exception handler rules* to which traditional rules can be sequenced to in case of exceptions. Figure 1b shows an assertion rule sequenced to a normal rule and an exception handler.

Execution Control Certain MT engines already support continuous and step-by-step execution. In the former, the MT is executed until none of its rules apply with user input optionally solicited when more than one rule applies simultaneously. In the latter, the user is prompted after every rule application.

[5] Rules that take other rules as input and/or output.
[6] This exact approach is presented in [11].

Stepping over in the context of rule-based MT corresponds to the execution of one (possibly composite) rule. Stepping into a composite rule should allow the modeller to execute its sub-rules one at a time. In T-Core-based languages [12], where rules are no longer atomic blocks but rather sequences of primitive operations, stepping into non-composite rules may also be sensible. Conversely, stepping out should cause the continuous execution of any remaining rules or primitives in the current scope. As for pausing an ongoing MT[7], the naive approach is immediate interruption although transactional systems might choose to commit or roll-back the current rule before pausing while T-Core-based systems might offer pausing between primitives. Either choice has its merits and, like the step commands, is heavily dependent on MTL and engine features: it seems reasonable that pausing only occur when the system state is consistent and observable. Finally, release (as opposed to debug) mode in this context should also disable pausing and stepping functionality.

Runtime Variable I/O MT debuggers should allow modellers to observe and change a rule's (or T-Core primitive's) inputs and outputs. Ideally, these should be presented using domain-specific constructs (and editors) rather than in their internal format. If future MT engines support on-the-fly changes (i.e., do not require the transformation model to be recompiled and/or relaunched for changes to take effect)[8], rule parameters and sequencing should themselves become viewable and editable at runtime.

Breakpoints MT breakpoints could be set on composite and non-composite rules, and T-Core primitives, as depicted in Figure 1c.

Stack Traces Stack traces could enable the navigation between the contexts of sub-rules and their parent rules, and between the contexts of T-Core primitives and their enclosing rule. Like stepping and pausing, they are thus clearly and closely related to the supported level of granularity of MTLs.

4.2 Creating Debuggable Artifacts

Before proceeding, we briefly review our technique for artifact synthesis from DSms. In [7], we proposed a means of generating backward links between DSms of mobile device applications, generated Google Android code, and intermediate representations. Our technique is based on triple graph rules (TGRs)[10], which provide generic means of relating constructs in LHS and RHS rule patterns. Figure 2 shows four perspectives of the same system connected via a complex "web" of links that reflects the application of the numerous TGRs that describe the DSL's denotational semantics. Though this web is not intended for direct human consumption, it can considerably facilitate the implementation of numerous components of a DSM model and artifact debugger by, amongst other things,

[7] To our knowledge, this is only supported by VMTS [6].
[8] To our knowledge, no such MT tools exist yet.

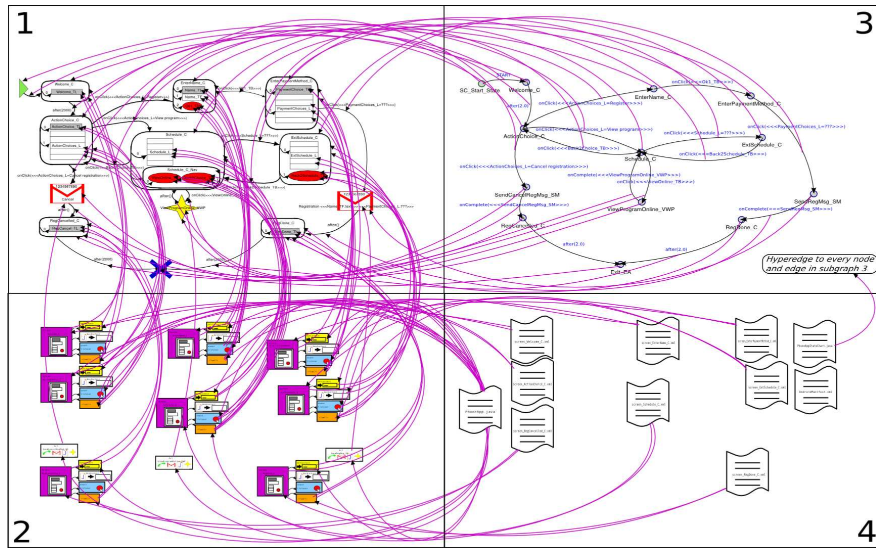


Fig. 2: (1)A DSm of a mobile conference registration application, generated models which isolate its (2)layout and (3)behavioural aspects, and (4) a trivial generated model of the final files on disk, all connected via traceability links.

enabling elegant two-way communication between DSms and generated artifacts, and providing implicit and navigable relationships between related concepts.

4.3 Debugging Models and Artifacts

MT debugging facilities may not be sufficient for debugging models whose semantics are specified denotationally. Running such models implies executing synthesized artifacts[9], not MTs. It is sensible to assume that in industry, DSLs (and their semantics) will often be defined by different actors than the end-users of the DSLs. Thus, we distinguish between two types of users. *Designers* are fully aware of the MTs that describe their models' semantics and generate artifacts. *Modellers* have an implicit understanding of their models' semantics but have little or no knowledge about how they are specified. As an example, consider Figure 2: a designer would be aware of each perspective, the links between them, and of the synthesized code and executable; a modeller would only be concerned with the DSm and synthesized executable. Consequently, the debugging scenarios for designers and modellers differ. Designer debugging focuses on ensuring the correctness of the MTs whereas modellers may assume that the MTs applied to their models are flawless and must instead establish the correctness of their models. In both scenarios, the basic work flow entails observing DSm versus artifact evolution[10] (with the designer possibly studying intermediate forms, if any).

[9] We restrict our attention here to programs and models.
[10] DSm evolution here equates to program variables taking on values at runtime.

A formal discussion on implementing this exceeds the scope of this paper but the approach reviewed in Section 4.2 enables the sort of communication between DSm and artifact that would be required. Below, we re-visit the debugging concepts described earlier and translate them to DSm and artifact debugging.

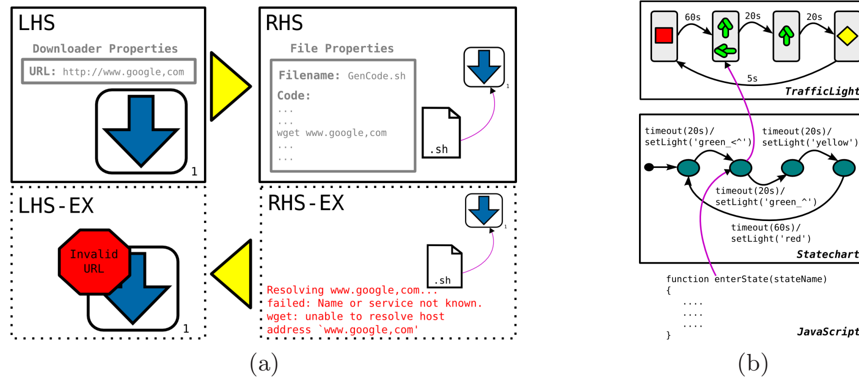
Print Statements Modern DSm editors offer means to display pertinent information as concrete syntax (e.g., depictions of Petri Net *place*s often include their number of *token*s) and provide easy access to construct parameters. Still, explicit output may be required. An elegant solution is the (semi-)automatic integration of output constructs in DSLs at design time. This is closely related to what we propose for MTLs and what is done in GPLs.

Assertions Assertions can also be (semi-)automatically specified DSL constructs. Provisions to enable and disable them, and to halt model animation and artifact execution when they fail are needed. These might be rules or function calls and should be generated (and weaved into artifacts) automatically.

Exceptions Although DSms may be animated, what is truly being executed are synthesized artifacts. Consequently, exceptions originate from artifacts and are described in terms of their metamodel. Some exceptions may describe irrelevant transient issues, others may describe issues relevant only to designers. Their handling is thus a design choice of the DSL architect. "Silent" exception handlers for irrelevant exceptions should be generated along with synthesized artifacts while relevant exceptions should be translated into domain-specific terms and propagated[11] to the modeller or designer (as depicted in Fig. 3a where a low-level error message results in updated concrete syntax at the DSm level).

Execution Control The play, pause, stop and step commands also require two-way communication between DSm and artifact. Playing and stopping simply require means to remotely run or kill the generated model or program. The meaning of a step in an arbitrary DSm, however, is not obvious. A general definition is that *a step constitutes any modification to any parameter of any entity in a DSm.* Still, stepping can be considered from two orthogonal perspectives: the modeller's and the designer's. On one hand, the three step commands, in conjunction with our general definition, intuitively translate to DSLs which support hierarchy and composition. On the other, generating artifacts from DSms creates an implicit hierarchy between them. Thus, a designer, may prefer for the step into operation to take a step at the level of corresponding lower level entities. This scenario is depicted in Figure 3b where two successive step into operations lead a designer from a domain-specific traffic light model entity to a corresponding Statechart *state* and finally to a function in the generated code. Conversely, stepping out would bring the designer back to higher level entities and stepping over would perform a step given the current formalism (e.g., one code statement, one operational semantics rule). As for the pause command, a

[11] The backward links described in Section 4.2 may be instrumental in this propagation.



(a) (b)

Fig. 3: (a)A high-level construct, its mapping to code, and a possible rendering of exception translation specification facilities; (b)Stepping into in “designer debug mode” navigates across levels of abstraction.

sensible approach is to pause the execution before running what would have been the next step at the DSm level for the modeller, and at the artifact level for the designer. These distinct stepping and pausing modes further motivate our previous proposal of separate debugging modes for designers and modellers. Finally, all of the above implies that (ideally automatic) instrumentation of artifacts to enable running only parts of them at a time are required[12].

Runtime Variable I/O Model editing tools can be used to view and modify DSm and generated model variable values whereas code IDE debugger facilities can be used for synthesized coded artifacts. The challenge is to propagate changes such that consistency is preserved across all levels of abstraction[12].

Breakpoints Breakpoints in state-based languages (e.g., Statecharts) could be set on states. For languages where state is implicit (e.g., Petri Nets whose state is their *marking*), however, they could be specified as patterns (pausing execution upon detection). Finally, designers should be able to specify breakpoints at any level of abstraction while modellers should be restricted to the DSm level.

Stack Traces Stack traces remain tightly bound to the step into and out commands. Thus, they might display related actions at different levels of abstraction for designers while reflecting construct composition, if any, for modellers.

5 Conclusion and Future Work

We proposed a mapping between concepts in the software and DSM debugging realms. We distinguished between MT and DSm debugging, and between debug-

[12] Once again, traceability links between DSms and artifacts may be key.

ging scenarios for *designers*, who are fully aware of the MTs that describe their models' semantics and generate artifacts, and *modellers*, who have an implicit understanding of their models' semantics but little or no knowledge about how they are specified. Our work is meant as a guide for developing DSM debuggers: numerous MT debugger features can be built-in to MTLs and engines, while modular TGR-based artifact synthesis can considerably facilitate the implementation of most DSm and artifact debugger facilities. We plan to fully implement the concepts proposed in this work in AToM^3's successor, AToMPM.

References

1. Alan W. Brown. Model driven architecture: Principles and practice. *Software and Systems Modeling (SoSym)*, 3:314–327, 2004.
2. Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal (IBMS)*, 45:621–645, 2006.
3. Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and Systems Modeling (SoSym)*, 3:194–209, 2004.
4. Marc Eisenstadt. "My Hairiest Bug" war stories. *Communications of the ACM (CACM)*, 40:30–37, 1997.
5. Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation*. Wiley-Interscience, 2008.
6. László Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Model transformation with a visual control flow language. *International Journal of Computer Science (IJCS)*, 1:45–53, 2006.
7. Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. In *The 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, 2010.
8. Raphael Mannadiar and Hans Vangheluwe. Modular synthesis of mobile device applications from domain-specific models. Technical Report SOCS-TR-2010.5, McGill University, 2010.
9. Laurent Safa. The making of user-interface designer a proprietary DSM tool. In *7th OOPSLA Workshop on Domain-Specific Modeling (DSM)*, page 14, http://www.dsmforum.org/events/DSM07/papers.html, 2007.
10. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*, 1995.
11. Eugene Syriani, Jörg Kienzle, and Hans Vangheluwe. Exceptional transformations. In *International Conference on Model Transformation (ICMT)*, volume LNCS 6142, pages 199–214, 2010.
12. Eugene Syriani and Hans Vangheluwe. De-/re-constructing model transformation languages. In *9th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT)*, 2010.
13. Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Software : Practice and Experience*, 38:1073–1103, 2008.
14. Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, second edition, 2009.