Automated Testing of DSL Implementations

Experiences from Building mbeddr

**Daniel Ratiu · Markus Voelter ·
Domenik Pavletic**

Abstract Domain specific languages promise to improve productivity and
quality of software development by providing problem-adequate abstractions
to developers. Projectional language workbenches, in turn, allow the defini-
tion of modular and extensible domain specific languages, generators and de-
velopment environments. While recent advances in language engineering have
enabled the definition of DSLs and tooling in a modular and cost-effective
way, the quality assurance of their implementation is still challenging. In this
paper we discuss our work on testing different aspects of the implementa-
tion of domain specific languages and associated tools, and present several
approaches to increase the automation of language testing. We illustrate these
approaches with the Jetbrains MPS language workbench and our experience
with testing mbeddr, a set of domain specific languages and tools on top of C
tailored to embedded software development. Based on the experience gained
from the mbeddr project, we extract generic lessons for practitioners as well
as challenges which need more research.

1 Introduction

Domain specific languages (DSLs) promise an increase in productivity and
quality of software development by providing abstractions that are adequate
for a particular application domain. Using adequate abstractions is a key en-
abler for constructive quality assurance (a wide variety of errors are impossible
to make) and for deep analyses (which are easier to perform at the abstraction
level of the domain). Recent advances in language workbenches, the tools used
to implement languages and their IDEs, enable a very productive approach to

Siemens AG. E-mail: daniel.ratiu@siemens.com
independent/itemis AG. E-mail: voelter@acm.org
itemis AG. E-mail: pavletic@itemis.de

building domain specific tools around DSLs [25]. In addition to the language itself, authoring support (e.g., editors, refactorings) and transformations (e.g., code generators), they enable a plethora of tools (e.g., analyzers) which are aligned with, and thus can take advantage of, the abstractions of the DSL.

Language Workbenches (LWBs)   are tools for efficiently implementing languages and their IDEs. They support defining the language structure and syntax, context sensitive constraints, type systems and transformations, as well as IDE features such as refactorings, find usages, debuggers or domain specific analyses. Typically, a language workbench ships with a set of DSLs for describing each language aspect, avoiding much of the accidental complexity traditionally involved in language implementation. In a sense, they move language development from the domain of computer science into software engineering.

Jetbrains MPS   is an open source language workbench developed by Jetbrains since the early 2000s[1]. It has comprehensive support for developing DSLs by specifying structure, syntax, type systems, transformations and generators, debuggers and IDE support. MPS is one of the most fully-featured language workbenches [7]. One distinguishing feature of MPS is that it uses a projectional editor: it does not use parsing to construct the abstract syntax tree (AST) from a program source, instead, editing gestures directly change the AST and the concrete syntax is projected from the changing AST. This approach has multiple advantages such as the possibility to use concrete syntaxes other than text, such as diagrams, mathematical notations or tables [30]. Furthermore, projectional editors never encounter parsing ambiguities, which is why MPS supports a wide variety of language composition techniques [26]. A short introduction in language development with MPS is given in Section 2.

mbeddr   is a set of 81 languages and C extensions for embedded software development built using MPS.[2] We started building mbeddr in 2012 and we have invested so far more than 10 person years of effort split across 5 main developers. Details about mbeddr can be found in [31,33,28], and our lessons learned regarding the development of mbeddr with MPS are discussed in [29]. Based on mbeddr, Siemens builds a commercial product called the Embedded Software Designer (ESD)[3]. In this paper, we focus on our experience with testing the implementation of mbeddr. The mbeddr project has over 2,300 tests in total, with over 6,000 assertions. This corresponds to approximately 50,000 lines of code, and is about 50% of the size of the mbeddr implementation itself. mbeddr is still under development and while we are developing new language features, we are continuously enlarging our knowledge about testing DSLs. This paper reflects the current state of testing mbeddr.

---

[1] `https://www.jetbrains.com/mps/`

[2] `http://mbeddr.com`

[3] `https://www.plm.automation.siemens.com/en_us/products/lms/imagine-lab/`
`embedded-software-designer.shtml`

1.1 Language Quality

An important aspect of software engineering is quality: the developed artifacts must conform to a defined minimum quality level for them to be a useful asset in practice. Several techniques are established in software engineering to reach this goal. These include process aspects such as systematic requirements management, issue tracking and code reviews, but also encompass specific practices and techniques such as (automated) testing, coding standards and refactoring.

Because of the increasing reliance on DSLs in diverse software development activities, there is a strong need to ensure a high quality of the language implementation and its associated tooling.

Due to the use of language workbenches for developing languages, the quality of the language implementation is ensured constructively to some degree: the DSLs provided by LWBs for language development prevent engineers from making certain categories of errors. However, many kinds of errors are still possible, and as languages become more complex, more widespread and developed in shorter cycles, systematic quality assurance is essential. In this paper we focus on the following categories of defects of the implementation of DSLs and associated tooling as detailed in the following.

Front-end Defects (D1)   The front-end of a DSL relates to the definition of the language structure (abstract syntax), definition of the editors (concrete syntax), context sensitive constraints and type system (static semantics).

- *Abstract syntax:* The abstract syntax (AST) of the language defines the available language constructs how they can be composed. Defects of the AST might either lead to the impossibility of defining some models that would be valid (missing constructs in the DSL, or lack of support to combine these constructs).
- *Concrete syntax:* As a projectional editor, MPS interprets the editing gestures and immediately creates models without any parsing involved. The creation of proper models might involve local re-orderings of the abstract syntax tree such as in the case of dealing with priorities of arithmetic operators. Defects in the editors might lead to invalid structures of the AST or discrepancies between what is shown to the DSL user and the content and structure of the model.
- *Static semantics:* If the constraints or typing rules are faulty, they allow language users to define models which are not semantically meaningful; if they are too strong then users cannot define meaningful models, if constraints or typing rules are too weak, runtime-errors (e.g. exceptions) occur in the IDE, or the generators may fail or produce non-compiling code.

Back-end Defects (D2)   Once valid models are authored, generators are used to produce lower-level artifacts such as code or configuration files. Due to the abstraction gaps between the input languages and the generation targets,

or because of optimizations implemented in the generators, they can become very complex and are thus error prone. We distinguish the following classes of generators problems:

— *Robustness defects:* For example, generators might simply crash on certain input models, or they might produce output which does not conform to the target language.
— *Semantics defects:* This refers to situations where the generators produce output that conforms to the definition of the target language but does not realize the intended semantics of the input model.
— *Stability of generated artifacts defects:* This refers to situations when using the same model as input, the generated artifacts change between generator runs. The instability of generated artifacts might lead to difficulties when the code reviews are performed or when versioning tools are used on the generated artifacts.

Defects in the Tooling (D3)   Domain specific languages open new possibilities for domain specific tools, such as analyzers or debuggers. They are often complex and depend on models at various abstraction levels. Furthermore, many of these tools rely on the integration of external tools (e.g. C-debuggers, C-level model checkers). We distinguish the following categories of defects:

— *Semantics misalignment defects:* Analyzers are prone to errors such as semantic misalignment with the generators, or misinterpretation of models. Semantic misalignments are especially important because they lead to inconsistencies between different semantic interpretations of the model (generators, or different analyzers). Independent of the existence of an explicit and formal semantics definition, the interpretations of the high-level models by different tools must be consistent with each other.
— *Lifting defects:* An important class of defects relate to lifting the analyses results from the output of the analysis tools back to the domain level. Bridging backwards the abstraction gap between the DSL and lower level artifacts make lifting challenging, error-prone or even impossible (e.g. due to differences between big-step vs. small-step semantics).
— *Interaction defects:* In certain cases, such as the integration of external debuggers, the pattern of interaction with the external tool is complex. For example, performing a step in a debug session at DSL level might require performing several steps at C-code level. Another example is an optimized interaction with a verification tool: in order to make use of multicore processors for analysis, we first run the verification tool to find out which properties are in scope, and then we start several instances of the verification tool (in parallel, on different cores) to check each of these properties separately.

1.2 Taming the Testing Challenges

Finding the defects described above in a systematic and (semi-)automated way requires a broad set of techniques, which we summarize below.

- *Unit tests:* MPS provides several DSLs for writing unit tests for different language definition aspects (Section 3).
- *Automating unit testing:* In order to test mbeddr-specific functionality (such as formal verification or domain specific debugger) we have created new testing DSLs in addition to those provided natively by MPS (Section 4).
- *Random generation of models:* We generate random models and use them to test the robustness of language definitions in a fully automated manner (Section 5).
- *Comparison with baseline:* Defects about stability of the generated artifacts can be automatically tested by using a baseline of generated artifacts and comparing the newly generated artifacts with that baseline (at text level). Comparison with a baseline can also be used to support manual code review about how changes in generators affect the generated artifacts.
- *Importing existing models:* An importer can be used for testing the structure of the DSLs for which models already exist. Language constructs or terms not covered by the DSL can be automatically identified.

In Table 1 we provide an overview of the defects defined in the previous section (D_1-D_3), the technical means which can be used to detect them and the degree to which their testing can be automated. We distinguish two categories of automation: fully automatic defects identification and semi-automatic identification using manually written unit tests and subsequently automatic execution of these tests.

Formal Techniques for Assuring the DSLs Implementation One focus of mbeddr is to make formal verification technology easy to use [23, 22, 15, 24, 20], and mbeddr incorporates several formal verification techniques which help practicing embedded developers increase the reliability of their programs.

The reader might ask why we have not relied more on formal techniques for assuring the correctness of the implementation of mbeddr itself. For example, one could formally specify the semantics of DSLs, automatically generate test cases using symbolic techniques, or formally verify the correctness of the generators. We did not do this because we are currently not aware of techniques that are scalable and user-friendly enough to make the formal specification of DSLs and verification of their implementation work in practice – the definition of a formal semantics for C alone is work for a whole PhD thesis [6] and very much beyond what we could have done. The contribution of this paper is to illustrate how *testing* can achieve a reasonably good quality of a language implementation. Formal techniques would be a worthwhile addition to the testing approaches presented in this paper. However, this requires the evolution of verification technology itself and better integration into language workbenches in order for the approach to become economically feasible.

Defect Category	Defect Class	Concrete Defect	Automation Degree
Front end	concrete syntax	priorities of operators	semi (U)
		projectional editor actions	semi (U)
		editors robustness	full (R)
	abstract syntax	language definition too relaxed	full (R)
		language definition too constrained	full (I)
		missing constructs	full (I)
	static semantics	type system errors - semantic	semi (U) / full (R)
		type system errors - robustness	full (R)
		constraints too relaxed	full (R)
		constraints robustness	full (R)
		dataflow analyses	semi (U)
Back end	generators	robustness defects	full (R)
		stability defects	full (B)
		semantic defects	semi (U)
Tooling	formal verification	semantic misalignments	semi (U)
		lifting of results	semi (U)
	debugger	interaction	semi (U)
		lifting of results	semi (U)

Table 1 Overview over language implementation defects (described in Section 1.1), the technical means to detect them (Section 1.2) and their testing automation degree. We refer to the technical means used for defects identification by using the following convention: U - manually written unit tests, R - random generation, I - using C code importer, B - using comparison with a baseline. Testing automation degree: semi-automatic (manually written test case and automatic running of tests) vs. fully automatic tests.

1.3 Structure and Contribution

In this paper we advocate that modern DSLs and their tools required a *holistic* approach for quality assurance which goes far beyond the techniques used in traditional compiler construction. Testing DSL-based tools requires a broader approach due to different language implementation aspects involved (in addition the language definition itself, we have a plethora of tooling and IDE support). Furthermore, the process and budget constraints for domain specific tools are different compared to traditional general purpose languages – requirements change more regularly, development is agile and the budget is more limited. The paper makes the following contributions to the state of the art:

- We present a categorization of various aspects of the quality of DSLs implementation and associated tooling that must be tested (Section 1.1).
- We extend the MPS native testing support with an approach to automate testing of domain specific tools such as model checkers and debuggers (Section 4).
- We present a fully automated approach for random testing, using synthesized models which will then be used to test the robustness and correct implementation of different language aspects (Section 5).

- We present our approach to measure the coverage of tests with respect to the different DSL and tooling implementation aspects (Section 6).
- Based on the experience so far, we present a set of best practices (lessons learned) and of fundamental challenges which we hope will trigger further research about testing the implementation of DSLs (Section 7).
- We illustrate the concepts in practice and present our experience with testing mbeddr, to the best of our knowledge, one of the largest DSLs-based projects, accumulating more than 10 person years of development.

This work is an extended version of an already published paper [21]. The basic approach for testing debuggers has been published in [17]. This paper presents a holistic, detailed and self contained view about testing of DSLs implementation.

Structure of the paper   In Section 2 we present a brief introduction about developing domain specific languages with MPS. In Section 3 we present the native support in MPS for testing different aspects of the DSLs implementation. In Section 4 we present our approach to increase the automation of testing domain specific tools such as the integration of the CBMC model checker in mbeddr and the debugger. In Section 5 we then present our strategy to automate testing by synthesizing random models (that respect the structures defined by the abstract syntax) and using increasingly complex oracles to detect bugs in the implementation of different language aspects. In Section 6 we illustrate our approach to measure the coverage of tests with respect to the different language implementation aspects. In Section 7 we discuss variation points and present the main lessons learned and more fundamental challenges with testing mbeddr. Section 8 contains related work, and we conclude the paper in Section 9.

2 Language Development in MPS

In MPS, a language implementation consists of several *language aspects*. At the core, these include structure, concrete syntax, constraints, type system, transformations, interpreters, data flow analyses or debuggers. We describe the most important of those in this section. MPS also supports additional IDE extensions such as buttons, menu items, or additional views; we have used those extensively for integrating domain specific tools such as analyzers. However, since these are implemented via regular Java/Swing programs and a couple of MPS-specific extension points, we do not discuss them in this section. Analyzers typically also read the model; to this end, MPS supports a rich API to navigate, query and modify models. We do not discuss this API in any detail in this section.

MPS ships with a dedicated DSL for implementing each language aspect; the idea is to apply DSLs not just to a developer's target domain, but to treat language development as just another domain for which DSLs can improve productivity. This approach is common in language workbenches [7].

For reasons of brevity, we will not describe any of these aspect-specific languages in detail; we refer the reader to [1,3,27]. However, in order to make this paper self contained, we provide some intuition in the rest of this section.

Structure   The structure aspect uses a declarative DSL to describe the AST of a language. Each AST element is called a *concept*, and concepts can have children (that constitute the tree), references (cross-references through the tree), and properties (primitive-typed values). In addition, a concept can extend one other concept and implement several concept interfaces. The structure aspect of MPS languages is closely aligned with EMF or EMOF.

Editor   Projectional editors do not use parsers; instead, they render, or project, a program's AST in a notation defined by the language developer. MPS' editor aspect defines this notation. An editor is made up of cells arranged in a hierarchy. For example, Fig. 1 shows the editor definition for an `if` statement. At the top level, it consists of a list of cells (`[- .. -]`). Inside the list, the first cell is a constant that projects the text "if". Between the constants for opening and closing parens, we embed the `condition` child using the `%child%` notation. The remaining parts of the `if` statement are optional; the `optional` cell handles this aspects. The `elseIfs` and `elsePart` children, just like the `condition` expression, come with their own editor definition which is embedded into the containing editor when programs are projected.

Fig. 1 The editor definition for C's `if` statement in MPS. In MPS, editors are composed of cells. Cells can be constant (e.g., the `if` keyword or the parentheses), they can embed other nodes (e.g., `%condition%`), or they can provide special editor interaction behavior (e.g., the `optional` cell). More details are explained in the text.

MPS supports the definition of several editors for a single concept that can be switched at runtime. In addition, editor definitions can contain logic to project the same concepts in different ways, depending on context or user preference.

Type System   Type systems are specified using declarative typing equations; MPS processes them using a solver. Various different kinds of equations are supported, the most important one is the type inference rule. It acts as type inference, but also acts as a type checker if several equations expect different types for the same AST nodes. Fig. 2 shows a rule that verifies that the condition of an `if` statement is Boolean type.

MPS also supports checking rules; these are essentially Boolean expressions that check some property of the AST and report errors if invalid code is detected. For example, they may report an error if they detect a duplicate name where unique names are required. The analyses performed by checking rules can be arbitrarily complex. For example, errors found by a data flow analysis

```
typeof(ifStatement.condition) :==: <BooleanType()>;
```

Fig. 2 A typing rule that verifies that the condition of an `if` statement is Boolean. Typing rules are expressed as equations, where free variables are implicitly declared using `typeof` and nodes, such as `<BooleanType()>`, represent fixed values. MPS' solver uses these equations for checking and inferring types.

are also reported via a checking rule: in this case, the implementation of the checking uses the data flow analyzer to perform the underlying analysis.

Scopes and Constraints   Typing and checking rules inspect an AST and report errors if invalid structures or types are found. In contrast, scopes and constraints *prevent* the user of a language from constructing invalid programs. Remember that in a projectional editor, users can only enter code that is proposed through the code completion menu. The structure of a language determines the concept that can be used in a child or reference link (e.g., the `condition` child of an `if` statement requires an `Expression`, or the `targetState` reference of a transition in a state machine requires a `State`). Scopes and constraints further constrain the valid nodes, beyond the concept.

Scopes determine which nodes are valid targets for a reference; they are written as arbitrary (procedural) Java code that returns a set of nodes, typically by traversing the AST and collecting/filtering valid targets. For example, a scope would determine that only those states that are in the same state machine as the current transition are allowed as a transition target.

Constraints affect the tree structure. For example, they might express that, even though an `AssertStatement` is a `Statement` and structurally (because of polymorphism) can be used in all places where a `Statement` is expected, they can actually only be used under a `TestCase`.

Transformations   MPS supports two kinds of transformations. Like many other tools it supports text generation from an AST. The DSL to achieve this essentially lets developers add text to a buffer and helps with indentation. However, most of MPS' transformation work is done by mapping one AST to another one; only at the very end are text generators used to output text files for downstream compilation (using gcc in the case of mbeddr, for example).

Fig. 3 shows an example of an AST-to-AST transformation from a language extension that supports concurrent queues and C. It uses the concrete syntax of the target language (C in this case) to define the target AST. Transformation macros (`$COPY_SRC$` or `$[]`) replace parts of that AST with parts of the input, recursively.

Generators assume a correct input model. In particular, this means that the assumptions made about correctness in the generator must be aligned with the typing rules, checking rules and constraints expressed in the language. Non-alignment is a source of errors that we discuss in this paper.

Generator Stack and Generation Plan   Generation in MPS is achieved through a sequence of generation steps which form the generation stack. Each generator step transforms a higher-level model into lower-level one until the

```
<TF {                                                          TF>
      $COPY_SRC$ [int8 ]___val = 0;
      boolean ___taken = false;
      atomic <->$[q]/readWrite> {
        if (!$COPY_SRC$ [q].isEmpty) {
          ___val = $COPY_SRC$ [q].take;
          ___taken = true;
        } if
      }
      if (___taken) {
        $COPY_SRCL$ [int8 code; ]
      } if
}
```

Fig. 3 An example of a generator template that defines the translation to C of a language extension that picks a value from a concurrent queue. The input is not shown, but the screenshot illustrates how source nodes are copied into the output (COPY_SRC) for further transformation, and how references are resolved using the -> macro. More details on the transformation process are explained in the text.

target language is reached; then, a text generator creates the final output for subsequent compilation.

Every language used in a particular model can contribute its specific generators to the generation stack. The (partial) ordering of generation steps in the stack is statically defined before the generation starts based on pair-wise priorities between different generators. The set of generation steps along with partial ordering among them form the generation plan. Thus, using an additional language for a given model, the ordering of steps in the generation plan changes and thereby bugs might be introduced.

3 MPS Native Support for Testing

In this section we discuss the facilities for language testing provided by MPS out of the box. We cover structure and syntax, editors, the type system, as well as the semantics as expressed through transformations.

3.1 Structure and Syntax

In a parser-based system, testing the structure and syntax essentially involves checking if a string of text is parsable, and whether the correct AST is constructed. In addition, one may want to check that, if invalid text is parsed, meaningful error messages are reported and/or the parser can recover and continue parsing the text following the error. These kinds of tests are necessary because, in a text editor, it is possible to enter invalid text in the first place; parser-based LWBs such as Spoofax directly support such tests [13]. In a projectional editor like MPS, entering code that is structurally invalid at a given location is impossible in the sense that the editor cannot bind the text and construct the AST; the entered text is rendered with a red background

as shown in Fig. 4. No AST is constructed. MPS does not support any way to test this: since the code cannot be entered, one cannot even sensibly write test cases.[4]

```
for ( int8 i = 0; i < 10
int8 add(int8 x, int8 y) {
  struct
  return x + y;
} add (function)
```

Fig. 4 Trying to enter code in a context where it is not allowed leads to unbound (red) code. Here we show the attempt at entering a `for` statement outside a function and a `struct` inside a function, where both of them are illegal, respectively.

Note that, because one cannot write code before the language structure and syntax are implemented, MPS does not support test-*first* development for language syntax. However, it supports test-*driven* development, where, for every piece of language syntax one defines, one can immediately write a test case. For other kinds of tests (type checks, semantics), a test-*first* approach is feasible and has been used occasionally by the mbeddr team.

3.2 Testing Editors

MPS also supports writing editor tests in order to test whether editor actions (right transformations, deletions or intentions) work correctly. They rely on entering a initial program, a script to describe a user's changes to the initial program using simulated interactions with the IDE (`press UpArrow`, `type "int8"`, `press CtrlSpace`), and an expected resulting program. When running such tests, MPS starts with the initial structure, executes the scripted user actions on that initial structure, and then validates whether the resulting structure corresponds to the resulting program specified in the test case. In Figure 5 we illustrate an example of testing the transformation of the `else` part of an `if` statement into an `else if`.

3.3 Validating the Structure

In addition to testing the technical aspects of the language structure and editors, one also has to verify whether the language is able to express all relevant programs from the domain. In general, there is no automated way of

---

[4] In principle, users could open the files in which MPS stores its models and modify the XML directly, potentially leading to invalid models. However, such low-level modifications are possible in many tools (UML tools, Simulink, MS Word) and we classify those more as sabotage and not error prevention. This is why we do not discuss this possibility any further and assume that users will modify their models exclusively with the IDE.

```
Editor test case If_statement_convert_to_else_if
description: no description
before: if (true) { } <cell else { }>
result: if (true) { } else if (<condition>) { }
code:
    type " if"
```

Fig. 5 Example of editor tests – we test that the `else` part of an `if` statement can be transformed into an `else if`. The `cell` annotations defines where the cursor is located, i.e., where the text typed by the user (" if") will go.

doing this – the language developer has to collaborate with domain experts to try and write a set of example programs that are agreed to cover the domain.

However, if a language is implemented for which a body of programs exist, these programs can be exploited. For the core C part of mbeddr, this is the case: all C programs ever written (with very few exceptions, detailed in [32]) should be valid mbeddr C programs.

We have automated this validation process using the mbeddr importer (see the Is in Table 1), a tool that parses existing, textual C code and converts it to mbeddr C code. Several open source code bases have been imported. By making sure those could be imported fully, we tested both the completeness of the language structure (all imported code must be representable in terms of the AST) and the importer (a failing import hints at a bug in the importer).

3.4 Testing the Type System

MPS supports the calculation of types for program nodes through a DSL for type system implementation. It supports type checking, type relationships such as supertypes as well as type inference. Depending on the language, type-system rules can be very complex. Native MPS tests can be used to check the proper calculation of types on certain nodes. In Figure 6 we illustrate an example of testing type system rules for an integer constant and of a basic logical operation.

```
Test case PrimitiveTypesTest
nodes
  (                                                                        )
    [ void primitiveTypesCheck() {
        <check 128has type (uint8 const volatile || int16 const volatile)>;
        <check 1 < 2has type boolean const>;
    } primitiveTypesCheck (function)                                       ]
```

Fig. 6 An example of testing the calculation of types: the tester writes example code (here: a function with numbers in it) and then adds assertions that check the types (the `check ... has type ...` annotations.).

```
Test case StatemachineConstraints
nodes
  ( ⎡ statemachine SM1 initial = s1 {        ⎤ )
    ⎢   in event e1()                         ⎥
    ⎢   <check in event e1()  has error ·     ⎥
    ⎢   in event e2()                         ⎥
    ⎢   state s1 {  }                         ⎥
    ⎢   <check state s1 {  } ·has error        ⎥
    ⎢   state s2 {  }                         ⎥
    ⎣ }                                        ⎦
```

Fig. 7 Example for testing context sensitive constraints: the names of events and of states of a statemachine must be unique. Similar to the type tests in Fig. 6, the tester writes example code and then embeds annotations that check for the presence of errors on the annotated nodes.

Checking for Error Annotations   This kind of test verifies whether MPS detects non-structural programming errors correctly; in other words, they verify that the expected red squigglies appear on the nodes that have errors. An example is testing the checking rule for uniqueness of event and state names of a state machine is shown in Fig. 7 . When such a test case is executed in MPS, nodes that have an error without having the green `has error` annotation will be flagged as a test failure. Conversely, nodes that have a `has error` annotation but do not actually have an error attached will also count as a failure. Note that this approach can also be used to check typing rules, because a failing typing rule also leads to an error annotation in the code.

Program Inspection   The test cases used for testing for error annotations can also be used to inspect other properties of a program: any program node can be labelled, and test cases can query properties of labelled nodes and compare them to some expected value. This has two major use cases, which we discuss below.

The first use case tests the construction of the AST based on what the user types, especially when a tree structure is entered linearly. The primary example for this are expressions. If the user enters `4 + 3 * 2`, precedence must be respected when constructing the AST: it must correspond to `4 + (3 * 2)` and not to `(4 + 3) * 2`. Using a node test case, the tester writes the program and then adds a test method that procedurally inspects the tree and asserts over its structure. Fig. 8 shows an example.

The second use case is testing behavior methods associated with program nodes. In a test case similar to the one shown in Fig. 8, a user can call a method on a labelled node and check if the returned values correspond to some expected value. It is also possible to query for the type of a node and inspect it; however, it is easier to use the declarative `has type` annotation shown above.

3.5 Testing Execution Semantics

We want to test that the results of the execution of an mbeddr program are as expected by the developer. In mbeddr, since extensions define their semantics by transformation to C, we test this by generating the mbeddr program to C and then executing the C code. If semantics are implemented by an interpreter, that interpreter can be called from a node test case, and asserting over the resulting behavior as part of a test case similar to the one shown in Fig. 8

Semantics testing requires the ability to express the expected behavior (oracle) in the DSL programs. Thus, one of the first C extensions we developed for mbeddr was the support for unit tests. As the right half of Fig. 9 shows, mbeddr test cases are similar to `void` functions, but they support `assert` statements. The number of failed assertions is counted, and the test case as a whole fails if that count is non-zero. Test cases can be executed from the console (or on a continuous integration server), or they can be run from within the MPS IDE. We have developed specific language extensions to test specific C extensions. The example in Fig. 9 shows the `test statemachine` statement that can be used to concisely test the transition behavior of state machines: each line is an assertion that checks that, after receiving the event specified on the left side of the arrow, the state machine transitions to the state given on the right.

Such extensions are not just useful for the language developers to test the semantics; they are also useful for the end user to write tests for a particular state machine. Language developers expect the state machine and the test case to both be correct, thus testing whether the generators work correctly. mbeddr end-users expect that the code generator is correct, testing whether the state machine corresponds to the assertions expressed in the test case.

3.6 Testing Stability of Generators and Language Migrations

The manually written test suites of mbeddr contain several hundreds of models. The generated code, once manually validated, can be used as an oracle for

```
Test case testSideTransformations
( [void f() {
     <expr  4 + 3 * 2>;
  ]} f (function)
test methods
  test testPrecedence {
    assert expr.isInstanceOf(PlusExpression);
    assert expr.Left.isInstanceOf(NumberLiteral);
    assert expr.right.isInstanceOf(MultiExpression);
  }
```

Fig. 8 This test case asserts that the structure of linearly entered expression respects the precedence specified for the + and * operators. It does so by peeking into the tree structure explicitly.

```
statemachine SM initial = S1 {        exported testcase testState {
  in event e()                          SM sm;
  in event f()                          assert(0) sm.isInState(S1);
  state S1 { on e [] -> S2 }            test statemachine sm {
  state S2 { on e [] -> S3 }              e → S2
  state S3 { on e [] -> S1 }              f → S2
}                                         e → S3
                                          e → S1
                                      } }
```

Fig. 9 An example state machine plus a test cases that uses a state machine-specific C extension to express the test for the state machine. Essentially, the tests contains a list of `event -> state` pairs, asserting that after triggering the `event`, the machine transitions into `state`.

further testing. We can generate C code out of these models, create a baseline and compare newly generated code with this baseline. We use the baseline both for automatic and semi-automatic testing:

1. Detecting the stability of generators can be done automatically by subsequent generation of code from models without performing any change.
2. Detecting changes in the generated code introduced by language refactorings or migrations can be done automatically. Subsequent manual inspection is needed because some of these changes are not genuine errors (e.g. change in naming convention of the generated entities).

Fig. 10 shows an example of instability of the mbeddr code generator which was automatically detected by using baselines of the generated artifacts. The ordering of the treatment of events changed between different generations.

The advantage of this testing strategy is that it is generic both with respect to the DSLs and to the target language of the generator; it could thus easily be supported directly by a language workbench. Furthermore, the structural comparison of the generated code with a baseline can be used to support manual code review of how changes in the generator affect the generated artifacts. This is especially important in the case when the target language is not executable (purely structural, such as XML configuration files) and thus no executable tests can be written. The downside is that considerable manual review effort is needed since many of the identified changes are as intended and thereby benign; they do not change the semantics.

4 Specialized Testing DSLs to Increase Automation

MPS is bootstrapped; it uses DSLs also for the development of DSLs themselves, and also supports extension of those languages. We exploited this for extending the MPS BaseLanguage (MPS' version of Java) in order to facilitate definition of test oracles at domain level. Using DSLs for expressing the oracles, as opposed to plain JUnit code, dramatically eases the definition, maintenance and review of tests because the expected test results are expressed at domain
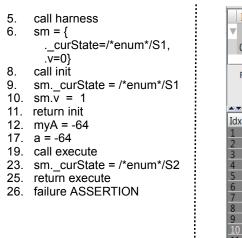
level. In this section we describe testing for two such domains: 1) consistent representation of counterexamples that result from the model checking of C code (Section 4.1) and 2) testing of a DSL-specific debugger (Section 4.2).

Fig. 10 Example of generator instability of statemachines identified automatically by comparing the generated artefacts with a baseline.

Fig. 11 High-level model representing a statemachine (left); the corresponding generated C code consisting of several functions and complex data structures (right)

```
 5.    call harness
 6.    sm = {
         ._curState=/*enum*/S1,
         .v=0}
 8.    call init
 9.    sm._curState = /*enum*/S1
10.    sm.v  =  1
11.    return init
12.    myA = -64
17.    a = -64
19.    call execute
23.    sm._curState = /*enum*/S2
25.    return execute
26.    failure ASSERTION
```

Idx	Property	Status	Trace...	Time
▼	Assertions (1)	**FAIL (1)**		1,29s
001	Assert: sm.isInState(S1)	**FAIL**	20	1,29s

Filter: [   ]

☐ Mark Results in Editor   ☐ Project cex.

Idx	Raw	...	Kind	Value
1	17	...	**call**	harness
2	18	...	**state**	S1
3	18	...	sm.v	0
4	20	...	**initialize statemachine**	sm
5	22	...	**state**	S1
6	23	...	sm.v	1
7	24	...	**leave statemachine init**	sm
8	25	...	myA	-64
9	27	...	a	-64
10	31	...	**trigger event**	sm->e
11	38	...	**state**	S2
12	40	...	**leave trigger event**	e
13	41	...	**FAIL**	Assertion Viol...

Fig. 12 Model-checking C code. The example shows an assertion failure and a counterexample (left); plus the lifted analysis result and counterexample (right).

4.1 Testing the Lifting of Counterexamples

mbeddr enables end-users to use advanced formal verification on their models. To enable this, mbeddr integrates the CBMC [5] model checker [23]. Our approach to increasing the usability of formal verification uses domain specific languages for property specification and environment definition. The verification itself is performed on the generated C code. This leads to the problem that the C-level results must be lifted back to the abstraction level of the original DSLs. A CBMC-based verification in mbeddr consists of the following steps:

1. mbeddr models are generated to C code and CBMC-specific macros,
2. CBMC is run on the generated code and produces a counterexample at C level if the verification fails,
3. The results and counterexamples are lifted back to mbeddr and each lifted step is associated with a node from the original input program.

In Fig. 11-left we show a simple state machine and a verification environment; on the right hand side we show the generated C code. In Fig.12-left we illustrate the counterexample at C level as produced by CBMC, and on the right hand side we show the IDE's representation of the analyses results: the lower part illustrates the lifted counterexample as displayed to the mbeddr user. A more complete example in the context of safety-critical systems is described in [15].

Lifting the results back to the domain level is challenging since it requires bridging the gap between the small-step semantics (at C-level) and the big-step semantics at domain level. Furthermore, it depends on technical details of MPS such as mapping the lines of the generated C code to the high-level AST

nodes from which that code originates. This mapping is not always accurate since a C-line can be mapped to several AST nodes. Furthermore, the mapping itself is subject to change as the code generators evolve. Automated testing is essential to ensure correct lifting.

During the early stages of mbeddr development we used plain Java and JUnit for testing the lifting of counterexamples. After several months, we realized that the Java-level test cases were hard to write and maintain. We then decided to describe the desired shape of counterexamples, as seen by users, by using a DSL as illustrated in Figure 13. The counterexample expressed in the DSL is subsequently translated into Java for integration with JUnit and the MPS testing framework.

```
test testSM0Counterexample {
  model m = model/test.analyses...counterexample.statemachines/;
  CBMCLiftedResult res = checkAsserts(m, "SM0", "test").get(0);
  counterexample test for res
    01 | call                     -> Function test
    02 | state                    -> State S1              ;
    03 | cnt.localVar             -> LocalVariableDeclaration 0
    04 | initialize statemachine  -> GenericDotExpression cnt
    05 | state                    -> State S1
    06 | cnt.localVar             ->  1
    07 | leave statemachine init  -> GenericDotExpression cnt
    08 | arg                      -> ExpressionStatement 1
    09 | trigger event            -> GenericDotExpression cnt->e
    10 | state                    -> State S2
    11 | leave trigger event      -> GenericDotExpression e
    12 | arg                      -> ExpressionStatement 2
    13 | trigger event            -> GenericDotExpression cnt->e
    14 | state                    -> State S1
    15 | leave trigger event      -> GenericDotExpression e
    16 | FAIL                     -> Assert Assert Failed
}

CBMCLiftedCounterexampleState currentState = res.getCex().getState(idx);
String kind = currentState.nodeKindAsString();
String value = currentState.nodeValueAsString();
SNode node = currentState.getNode();
Assert.assertEquals("Mismatch in node kind", "call", kind);
Assert.assertEquals("Mismatch in node value", "testCounterexample", value);
Assert.assertEquals("Mismatch in analyzed node", "Function",
    SNodeOperations.getConceptDeclaration(node).getProperty("name"));
```

Fig. 13 Example of counterexample DSL which consists of sixteen steps (top) and the translation of the first step of the counterexample into Java (bottom).

The testing of counterexample lifting in mbeddr contains 78 high-level tests with a total of 714 steps. Building this DSL early in the development of analyses proved to be a very good investment over time. We had to fix several times the counterexamples lifters and the tests in order to keep them aligned with the evolution of the generators. Having the tests written with a DSL drastically eased the maintenance process of the CBMC integration.

4.2 Testing Debuggers

mbeddr comes with an extensible source-level debugger allowing language users to debug their code on the abstraction level of the mbeddr languages, hiding the details of the generated C code [19,18].

Each mbeddr language contributes a debugger extension, built with a framework/DSL that was developed during the development of mbeddr [19]. These extensions provide application-specific debug actions (buttons or menu items in the IDE) and views on the program state (additional windows in the IDE). Debugging support is implemented specifically for each DSL by lifting the call stack/program state from the base-level (C program) to the source-level (mbeddr program and its extensions); stepping and breakpoints are mapped in the other direction. Similar to counterexample lifting described above, we facilitate this using DSLs.

The mbeddr debugger uses gdb[5] to perform an actual debugging session on the generated C code. It obtains the debug state from gdb and and uses it to control the execution of the compiled C program. Language engineers extend the mbeddr debugger to build debugging support for a C extension by describing the mapping for language constructs from the C base language to their language extensions (e.g., state machines). This mapping comprises different aspects: 1) lifting the call stack by hiding stack frames that have no representation on the source level and rendering the remaining ones with their proper identifiers, 2) lifting watch variables by showing only those that have a representation on the source level and lifting their values accordingly (e.g., lifting a C `int` value to an mbeddr `boolean` value), 3) translating breakpoints from the source level to the C level and 4) customizing the stepping behavior based on the semantics of the languages used in a program.

Due to the interplay between the different semantic levels (i.e. abstract models on the one hand and C code on the other hand), these mappings can become very complex. Ensuring the correctness of the debugging aspect requires comprehensive and rigorous testing. In this section, we present our approach to automate the testing of debuggers.

4.2.1 Example: Debugger Extension for the State Machines Language

In the following, we sketch the debugger extension for the state machines language (Fig. 9) in order to illustrate the mbeddr debugger framework.

Breakpoints To stop the execution of certain high level DSL constructs (e.g. when transitions are fired, when `TestStatemachineStatement` are entered), we must support setting breakpoints at the DSL level.

Watches Language constructs with *watch* semantics are translated to C variables. When suspending program execution, C variables with a source-level representation are lifted to the source level and thus end up in the debugger

---

[5] https://www.gnu.org/software/gdb/

UI as watch variables (e.g. the current state of a state machine, the value of internal variables of state machines).

Stepping   Language constructs where program execution can suspend after performing a *step over* (e.g., `Transition`), *step into* (e.g., `TestState-machineStatement`) require special treatment from the debugger framework. For these language constructs, language engineers use a DSL to describe where program execution will suspend to imitate the expected behavior of *step into*, *step over* and *step out*.

Call Stack   For language constructs that are callable (i.e., look like functions, e.g., `state machine`), the mbeddr debugger shows stack frames in the source-level call stack. Similarly to implementing stepping behavior, the debugger framework provides a DSL that is used to describe how source-level stack frames are constructed from the low-level C call stack.

4.2.2 Automating the Testing of Debuggers

In early versions of the mbeddr debugger framework, we had no possibility to test the debugging behavior automatically. Language engineers had to manually test the debugging support by running the debugger on mbeddr programs, manually verifying the assumptions about the expected debugging behavior. Not having an automated testing approach was a serious problem, because mbeddr languages evolve quickly and thus regularly break the mappings (see Section 4.2.1) that have been described with DSLs of the mbeddr debugger framework. These mappings depended on the language structure and the implementation of generators. Modifying either of the two usually caused the mapping to become invalid, breaking the debugger. With more users adopting mbeddr, automated testing of debuggers became necessary. Implementing such tests in plain Java would be very complex and hard to maintain and validate. We have thus developed DeTeL (*De*bugger *Te*sting *La*nguage), a generic and extensible DSL for testing interactive debuggers [17].

DeTeL should allow language engineers to test all debugger aspects introduced earlier: call stack, program state, breakpoints and stepping. Automated testing of the debugger should cover validation of the debugger state and of debug control.

Changes in generators can modify the structure of the generated code and the naming of identifiers and this way, e.g., invalidate the lifting of the program state in the debugger. To be able to identify these problems, we need a mechanism to validate call stacks, and for each of their stack frames the identifier, program state and location where execution is suspended. In terms of program state, we must be able to verify the identifiers of watch variables and their respective values.

Changes in the code generator can also affect the *stepping behavior* – e.g. changing the state machine generator to translate entry and exit actions to separate functions instead of generating one function for a `state machine` would require modifications in the implementation of *step out* as well. To

be able to identify these problems, we need the ability to execute stepping commands (in, over and out) and specify locations where to break.

4.2.3 Testing the State Machine Debugger Extension

Before we use DeTeL to write test cases for validating the debugging behavior for the state machine language, we first take the example program from Fig. 9, introduce a test-suite (`testcollection`) that gets reduced to a `main` function and annotate this program in Fig. 14 below with program markers. These markers are subsequently used by debugger `test case`s for specification and verification of code locations where program execution should suspend.

```
entrypoint testcollection main {          statemachine SM initial = S1 {
tests:                                      in event e() <no binding>
  [testState; (unittest)]                   in event f() <no binding>
}                                           state S1 {
exported testcase testState {                 [on e [ ] -> S2 ] inS1
  SM sm;                                    } state S1
  assert(0) sm.isInState(S1);               state S2 {
  [test statemachine sm { ] stepIntoSM        [on e [ ] -> S3 ] inS2
    e → S2                                  } state S2
    f → S2                                  state S3 {
    e → S3                                    [on e [ ] -> S1 ] inS3
    e → S1                                  } state S3
  ]                                       }
} testState(test case)
```

Fig. 14 Test vectors for testing the debugger are mbeddr models annotated with information about the expected behavior at runtime when the debugger is used on these models.

Stepping Into a State Machine   For testing *step into* for state machines, we first create a debugger test (Fig. 15) `SMTesting` that will contain all debugger `test case`s described in this section. `SMTesting` refers to the binary `UnitTestingBinary` to be executed (compiled from the generated C code) and instructs the debugger runtime to execute tests with the gdb.

Inside this debugger test, we create a debugger `test case` named `stepIntoStatemachine` shown at the bottom of Fig. 15. This `test case` first suspends program execution at the program marker `stepIntoSM` (the `test statemachine` statement), and then performs a single *step into* command before it compares the actual call stack against the call stack specified at the bottom. With this specified call stack, we expect to suspend program execution inside the `state machine` at the program marker `inS1` with no watch variables being shown by the debugger.

```
Debugger Test SMTesting      tests binary:  UnitTestingBinary
                             uses debugger: gdb

test case stepIntoStatemachine {
  suspend at:
    stepIntoSM
  then perform:
    step into 1 times
  finally validate:
    call stack {
      2 SM: inS1, watches {}
      1 testState
      0 main
    }
}
```

Fig. 15 DeTeL `test case` for testing *step into* for the `state machine` from Fig. 14. This `test case` first suspends program execution at the `test statemachine` statement (`stepIntoSM`), next performs a single *step into* command and validates afterwards that the source-level call stack contains three stack frames.

Stepping within a State Machine   The second testing scenario verifies that performing a *step into*, *over* or *out* command on a fired transition (`inS3`) of a `state machine` suspends program execution back on the trigger, which is in our example a `test statemachine` statement (`stepIntoSM`). To test this particular scenario, we create in Fig. 16 three different `test case`s that suspend execution location at the program marker `inS3`, perform a single stepping command (*step into*, *step over* and *step out*) and to then verify that program execution has suspended inside `testState` (a `testcase`) at the program marker `stepIntoSM` and a single watch variable `sm` is present representing the local variable declaration `sm` of type `SM`. Remember, since the transition at `inS3` has no guard condition and no body, performing a *step into* or a *step over* on this transition has the same effect as performing a *step out*.

The outcome of executed DeTeL `test case`s are visualized in a dedicated UI showing the test result (success or fail) and the process output.

4.2.4 Testing the mbeddr Debugger

For the purpose of testing the mbeddr source-level debugger, we have used the testing DSL presented in this section to create a test suite targeting Windows and Linux environments, and comprising 496 program markers, 60 debugger tests and 308 debugger test cases. This test suite validates debugging behavior for the mbeddr C implementation and for 11 language extensions including components, state machines anad physical units.

5 Test Synthesis

To increase testing productivity, and hence to increase the quality of language implementations, we must increase automation. In particular, we rely on automatically synthesizing mbeddr models and then using various oracles

```
test case stepOverFiredTransition {          test case stepOutOfFiredTransition {
  suspend at:                                  suspend at:
    inS1                                         inS1
  then perform:                                then perform:
    step over 1 times                            step out 1 times
  finally validate:                            finally validate:
    call stack {                                 call stack {
      1 testState: stepIntoSM, watches {sm}        1 testState: stepIntoSM, watches {sm}
      0 main                                       0 main
    }                                            }
}                                            }
test case stepIntoFiredTransition {
  suspend at:
    inS1
  then perform:
    step into 1 times
  finally validate:
    call stack {
      1 testState: stepIntoSM, watches {sm}
      0 main
    }
}
```

Fig. 16 Three different DeTeL `test case`s that validate the debugging behavior for performing *step into*, *over* and *out* commands on the transition in state `S1` (`inS1`). The `test cases` perform the respective stepping command from the same program location (`inS1`) and validate afterwards that program execution has suspended inside the caller `testcase`, which contains the next to be executed statement, by verifying the structure of the actual call stack.

for checking the robustness and correctness of the languages implementation. Our approach relies on the following steps; we explain them in detail in the following subsections.

1. Automatically synthesize input models that are correct in terms of context insensitive syntax (i.e., comply with the abstract syntax of the language).
2. Run the context-sensitive constraints and type system checks against these models. Models where no errors are reported are considered valid mbeddr models and may be processed further; those with errors are deemed correctly recognized as invalid and discarded. Runtime errors in the checkers reveal robustness bugs in their respective implementations.
3. Generate C code from the non-discarded models. If the generator throws exceptions while processing a valid input model, we have found a bug in the generator.
4. Compile the generated C code. If the compiler produces an error, we have also found a bug in the generator: it has produced non-compilable C code for a valid model.
5. If code compiles successfully, we then evaluate a suite of assertions that check the functional requirements on the generator. These assertions describe the relation between the structure of the input models and the generators output of the generator and are defined by the language engineers in addition to the generators.

5.1 Automatic Model Synthesis

We generate test vectors (i.e. mbeddr models) by using concepts from the languages under test. If the language under test can be composed with other languages (a crucial feature of MPS and language engineering in general), we test them together by generating models which cover the space of possible compositions as defined in a model synthesis configuration. For big languages, generating non-trivial models like those typically written by a programmer requires dealing with an explosion in complexity because the combinatorial space of composing the language constructs is large. To address this problem we have developed a mutation algorithm which uses as input existing, manually written models and replaces model elements with new ones according to a user-defined strategy.

Generation Configuration   To specify the parameters for a generation session, we must define a generation configuration. Fig. 17 shows an example of a configuration for generating models. Given a set of models as `starting points`, the algorithm attempts for a certain `number of tries` to mutate these models. To perform a model mutation, it randomly chooses a node from this model of a certain type (`seed chooser`). It replaces this node with another subtree that is synthesized up to a certain `depth` by using meta-classes from the set of `languages of interest`. The language concepts are chosen as specified by the `concept chooser` configuration.

```
starting point: random from module-reference/test.ex.ext.statemachine/
maximal number of tries: 10000
seed chooser: concepts: IStatemachineContents, IStateContents, Expression
interesting languages:
  com.mbeddr.core.statemachines, com.mbeddr.core.expressions,
  com.mbeddr.core.modules, com.mbeddr.core.statements
concept chooser: random concept chooser
depth: 2 .. 5
```

Fig. 17 Example of a testing configuration which takes existing models from the manually written tests for the statemachines, and tries 10000 times to mutate nodes of type `IStatemachineContents`, `IStateContents` or `Expression` using randomly chosen concepts from the given set of interesting languages. Only mutations with depth between 2 and 5 are considered as valid and saved.

Choosing the seed   There are two extreme possibilities to choose the starting point in the generation. One extreme would be to start from an empty model and let the model synthesis engine grow models with increasing depth and complexity. Another extreme would be to use a large set of existing, manually written test cases (which presumably cover the languages properly) and perform "small" mutations on these tests – the second strategy resembles fuzzy testing. In the cases when the generator generates deep models (as opposed to "shallow" fuzzying of existing models), a significant fraction of the synthesized models are "uncommon" due to the randomness of the deep synthesis and thus

implausible to be built by end users. Light mutation of existing models results in models closer to what a language user would build and thereby eventually find more common bugs. Furthermore, the manually written test models have a good coverage of major DSLs constructs and their combinations. Slight modifications of these models by fuzzying is similar to increasing the coverage in a incremental manner.

After experimenting with different strategies for choosing seeds, we came to the conclusion that fuzzing is the most efficient. Growing from scratch models of relevant size proved to be very difficult – the synthesis algorithm often got lost in building and discarding huge amount of models which are invalid. Starting from a certain model to be fuzzed requires that we specify the fuzzing strategy and where the mutation starts from. A naive way would be to choose randomly a node and start mutating it. This strategy does not work because the number of models candidate to be fuzzed can itself be very big. Each of these models has several hundreds of nodes from which the mutation could plausibly start. To further guide the testing process (e.g. for a test campaign for state machines) we must restrict the strategy for choosing seeds (e.g., by requiring that the seed is a node which belongs to the state machines language).

Choosing the replacement concept   Once the node to replace is chosen, we need to "grow" other nodes which can replace it. We can define the strategy to grow these nodes with the help of a `concept chooser`. The naive way to deal with this is to chose randomly concepts from the set of enabled languages and which can be used to replace the node. We can however guide the concepts selection process for example by specifying different probabilities for choosing the concepts.

Performance concerns   We measure performance of the models synthesis along two directions: how good do the synthesized models cover the space of the input languages, and how fast can we generate models. The speed with which new valid models are obtained depends on the chosen minimal depth, on the starting point definition and on the chosen concepts. For measuring the coverage of the generated tests we use the techniques described in Section 6.

Experience   mbeddr's implementation of C has 317 language concepts, 116 concept interfaces, as well as 170 relations between them. mbeddr has many extensions which can themselves be quite big – e.g. the state machine language has 41 concepts, 7 interfaces, and 38 relations. All these extensions can be composed with C and with each other – e.g. state machines can be declared in C modules, C expressions can be used in transition guards, and C types can be used as types for event arguments; furthermore, state machines can be directly used as implementation of mbeddr-components (another extension). We use the models that have been manually written for unit tests of the languages as starting points for fuzzing. This way, no additional effort has to be spent on writing the starting point models.

5.2 Robustness of the Front-End

After a model is synthesized, we exercise the functionality of the language
front-end (e.g., editor, constraints, type system). We expect that each model
can be opened in the editor without any runtime error due to faults in the
implementation of the editors. We also expect that the constraints and type
system checks are performed successfully, in the sense that no runtime er-
rors occur as a consequence of a faulty implementation of the checks them-
selves. Otherwise we have identified a potential robustness defect in the editor,
type system rules or constraints. The code causing these runtime errors must
be manually inspected because sometimes a particular model cannot be con-
structed by the user as a consequence of restrictions in the editor, so the error
can never occur with end users. The mutation algorithm does not consider the
editor restrictions when models are created.

Experience   Before starting with automatic testing, we had a high confi-
dence in the front-end because front-end errors are very visible while using
mbeddr; we assumed that most had been found and reported. To our surprise,
the automated tests quickly identified dozens of errors for the mbeddr core lan-
guages. This is because the random model synthesizer produces models that
a typical tester may not think of but are still plausible to be encountered in
practice.

   In Fig. 18 we illustrate an example of a major robustness defect of editors.
In the upper part of the figure is the starting model which is displayed prop-
erly. However, if component `aName_832184` extends the component `A` (leading
to cyclic extension, which is not allowed), then the editor crashes with the
result in the bottom part of the figure. The cause of this editor crash is a
stack-overflow exception. In this case we have identified two bugs: 1) a miss-
ing check that no circular extensions among components are allowed, and 2)

Fig. 18 Example of the editor working properly (top), and how the same editor is displayed
when a runtime exception is thrown (bottom). The model causing the runtime exception is
obtained through a small mutation of the original model (top) by creating a cyclic extension.

the editor bug itself (i.e. editors crash in presence of cyclic extensions among components). Other similar errors with cyclic references have been found this way, for example, among typedefs.

In Fig. 19-left we illustrate an example of a front-end error in which the synthesizer replaced the reference to variable `sm` of type `Statemachine` with a reference to a variable of type integer, causing a `ClassCastException` in the code that scopes the right side of the dot expression. Fig. 19-right shows the original front-end code as well as the fix.

5.3 Robustness of Generators

When implementing generators, developers make assumptions about the space of possible input models. These assumptions should be codified into constraints or type checks, to prevent invalid models from entering the generator. If this is not done, the generator may throw exceptions during its execution. As explained earlier, we consider a generator *robust* if for each input model which is free of (reported) errors, the generator does not throw exceptions during its execution. If exceptions are thrown, either the generator has a bug (because it fails to process a valid model correctly) or the constraints in the language are insufficient (failing to reject invalid models).

```
int16 anIntVar;
void test1() {
  SM cnt;
  cnt.init;
  cnt.trigger(evt);
} test1 (function)
```

```
int16 anIntVar;
void test1() {
  SM cnt;
  cnt.init;
  anIntVar.trigger(evt);
} test1 (function)
```

```
link {event}
  referent set handler:<none>
  scope:
    (exists, referenceNode, contextNode, enclosingNode
      node<GenericDotExpression> gde;
      gde = enclosingNode : GenericDotExpression;
      node<> tpe = gde.expression.type;
      return tpe : StatemachineType.machine.inEvents();
    }
```

```
link {event}
  referent set handler:<none>
  scope:
    (exists, referenceNode, contextNode, enclosingNode
      node<GenericDotExpression> gde;
      gde = enclosingNode : GenericDotExpression;
      node<> tpe = gde.expression.type;
      ifInstanceOf (tpe is StatemachineType smtpe) {
        return smtpe.machine.inEvents();
      }
      return new nlist<InEvent>;
    }
```

Fig. 19 Example of defects in the implementation of the front-end. On the left-top side we present the mbeddr model which does not show any error due to a robustness issue of the checker. On the bottom-left we show the mbeddr model with functioning constraints. On the top-right we present the initial implementation (which caused the exception). On the bottom-right we present the fixed implementation (changes are highlighted in red).

Experience   In our experience with mbeddr, surprisingly many problems result from assumptions of the generators not properly reflected in constraints or typing rules. For example, in the case of the statemachines language (containing ca. 40 constructs) after synthesizing only a few hundred models, we identified about a dozen robustness issues with their generator. Most of the issues originate from the insufficient constraints in the front-end of mbeddr which do not reflect all the assumptions made by the generators.

5.4 Structural Correctness of Generated Code

It is possible that the output code (e.g. C) is not meaningful because it does not comply with the target language definition. In this case, the generated code does not compile. Again, this can be caused either by too weak constraints on the input model, or by generic errors in the generator. The oracle in this case is a checker of the well-formedness of the target models. In practice, we try to compile the code with gcc.

Experience   mbeddr assumes that for any valid (error-free) mbeddr model, the generated C code compiles. Due to the complexity of the C language, we have found dozens of valid mbeddr models that result in C code that does not compile. By manually inspecting these errors, we found that over 90% of these errors originate from a lack of checking in the font-end. Inspection revealed that often, checks existed in the code but they were not strong enough to cover all possibilities.

In Fig. 20 we present several examples of models which mbeddr considered valid, but which lead to C code which does not compile. One of them is a function with a single variadic argument – this is invalid in C but due to the lack of constraints, mbeddr mistakenly considered this to be valid.

```
exported void singleVariadicArgument(...) {

exported void variadicArgumentOnFirstPosition(..., int8 a) {

exported typedef point as my_point;
struct point {
  int8 x;
  int8 y;
};
```

Fig. 20 Examples when insufficient constraints in mbeddr lead to C code which does not compile. On the top of this figure are functions with one argument of variadic type, in the middle variadic argument on the first position, and on the bottom an exported typedef which is generated in a .h file referencing a structure declared only in .c file.

5.5 Semantic Correctness of the Generator

To check the semantic correctness of the generator (beyond the structural correctness discussed above) we must check the relation between the input models and the target models. Target models are the lowest level models in the generation stack, they closely resemble the subsequently generated textual C code.

Assertions   We allow language developers to define assertions about the relation between the input model at DSL level and target model that capture the intent of a generator. These assertions represent the understanding of how the DSL should be translated into the target language and are thus a partial specification of the semantics. In Figure 21 we present a set of assertions which check that for each state machine the corresponding C program parts are generated. First, we check that for each `input event` we generate an `enumeration literal`; second, we check that a `struct` declaration which contains internal data is generated, and that for each state machine variable declaration a member is generated into this `struct`; third, we check that for each `state` we generate an `enumeration literal`; and last but not least we check that the `execute` and `initialize` functions are generated.

Experience   Our experience so far with the effectiveness of these semantic checks shows that only very few additional bugs can be identified. We have three hypotheses about why this is the case. First, we already have a significant number of executable unit tests (like the one presented in Fig. 9) which make sure that the generators work correctly. Second, our assertions are mostly coarse grained; detailed errors for corner-cases would not be caught with these coarse-grained constraints. Third, essentially the same `foreach` statements are used in the generator templates themselves which are also written in a declarative manner, so it is unlikely that we would find additional errors there.

The bugs we did find with this mechanism are due to the more complex, procedural aspects of the generator (e.g. flattening of composite states). There, the declarative properties are in fact much simpler (and hence, less likely to contain errors) than the algorithmic implementation in the generators.

6 Measuring the Quality of Tests

A common criterion for measuring test quality is the structural code coverage of the to-be-tested system. In the case of DSLs development, the system under test is composed of code written in the MPS DSLs for the various language aspects (abstract syntax, editors, type-system rules, constraints, generators) as well as the domain specific that go along with the DSLs. To measure the coverage with respect to all these aspects, wee follow a staged approach:

1. concept coverage: ensure that all abstract syntax elements of the language(s) of interest are instantiated in tests;

```
foreach Statemachine : sm from: original.descendants<concept = Statemachine> {
  // enum declaration for input events
  exists EnumDeclaration : ed from: outputModel.nodes(EnumDeclaration)
    ed.name.equals(sm.name + "__inevents") : "no enum for input events found" {
    foreach InEvent : ie from: sm.inEvents() {
      exists EnumLiteral : el from: ed.literals
        el.name.equals(sm.name + "_" + ie.name + "__event") :
          "input event " + ie.name + " does not have a corresponding enum literal"
    }
  }
  // structure declaration for statemachine state
  exists StructDeclaration : sd from: outputModel.nodes(StructDeclaration)
    sd.name.equals(sm.name + "__data") : "no structure for internal state" {
    exists Member : m from: sd.members
      m.name.equals("__currentState") : "no '__currentState' member found"
    foreach StatemachineVariableDeclaration : svd from: sm.localVariables() {
      exists Member : m from: sd.members
        m.name.equals(svd.name) : "variable " + svd.name + " does not have a corresponding member"
    }
  }
  // enum declarations for states
  exists EnumDeclaration : ed from: outputModel.nodes(EnumDeclaration)
    ed.name.equals(sm.name + "__states") : "no enum declaration found" {
    foreach AbstractState : s from: nonCompositeStates(sm) {
      exists EnumLiteral : el from: ed.literals
        el.name.equals(literalName(sm, s)) : "no enum literal found for state " + s.name
    }
  }
  // execute function
  exists Function : execFun from: outputModel.nodes(Function)
    execFun.name.equals(sm.name + "__execute") : "execute function does not exist" {
    foreach AbstractState : s from: nonCompositeStates(sm) {
      exists SwitchCase : swCase from: execFun.descendants<concept = SwitchCase>
        swCase.expression : EnumLiteralRef.literal.name.equals(s.name) : "no switch-case found"
    }
  }
  exists Function : initFun from: outputModel.nodes(Function)
    initFun.name.equals(sm.name + "__init") : "init function not found"
}
```

Fig. 21 Example of a DSL which describes assertions about relations between input and output models.

2. editors coverage: ensure that the entire logic for rendering the concrete syntax (projection) is covered by the tests;
3. constraints: make sure that all constraints, checking and type-system rules from the language definition are applied in tests;
4. generators: make sure that all generators and transformations rules of these generators are exercised by the tests;
5. tooling: check that tooling such as debuggers or various analyzers work properly with the DSLs for which they are enabled

Measuring the concept coverage is easy to achieve in MPS by automatically inspecting the models and checking which concepts are instantiated. For all other aspects we measure the coverage of the Java code[6] which is ultimately generated from all the code expressed in MPS' language implementation DSLs. In Fig. 22 we show an assessment that reports all concepts from the statemachines DSL that are not instantiated in the model that contains the tests for

---

[6] We use EMMA for measuring coverage http://emma.sourceforge.net/

the lifting of counterexamples; the lifting of counterexamples is not tested at all for the listed concepts.

```
assessment: coverage_of_statemachine_concepts_in_counterexample_tests
query:       concepts not instantiated in scope
             languages of interest: com.mbeddr.ext.statemachines
             search scope: model/test.analyses.counterexample.statemachines/
```
```
▌ SmSetStateTarget
▌ SmHasTxFiredTarget
▌ CommentedStateContent
▌ EmptyStateContents
▌ CommentedStatemachineContent
▌ StatemachineTestStep
▌ StatemachineTestStatement
```

Fig. 22 Example of the assessment of the coverage of statemachine concepts in the tests for counterexamples lifting. Below the line are the results of running the assessment, namely, the concepts from the statemachines language which are not instantiated at all in the set of tests for counterexamples lifting for statemachines.

Java-level code coverage measurement results are lifted to MPS and presented to language engineers in a convenient manner. In Fig. 23 we present an example of coverage measurement results; the top part of the figure is the coverage information at Java level (as offered natively by EMMA), on the bottom part the figure illustrates the same coverage result lifted to the level of MPS – in this case, transformation rules.

Fig. 23 EMMA based coverage for the statemachines generator: the upper part of the figure illustrates the coverage information at Java level in HTML; the bottom part of the figure illustrates the coverage information lifted at the level of MPS.

Experience We have started measuring the test coverage relatively late in the development of mbeddr because MPS does not offer support for measur-

ing coverage of the DSL implementation out-of-the-box – we had to retrofit this functionality ourselves by using EMMA and lifting the raw Java coverage results in MPS. We incrementally enriched the test suites such that better coverage is achieved. For example, the line coverage for the statemachine generator is currently 95%, in the case of other DSLs the generators coverage is over 75%[7]. In the case of the checking and typesystem rules the current lines coverage is around 50% for the statemachines language and between 30% and 80% for the other DSLs.

We have also noticed that the concepts coverage is very coarse-grained – i.e., it is "easy" to construct mbeddr models which instantiate at least once all concepts of a DSL, while not resulting in meaningful coverage for the other DSL aspects.

7 Discussion

In this section we summarize the lessons learned so far, discuss why and how things could have been done differently, and point out challenges and open points that need to be addressed in the future in order to better support efficient testing of DSLs implementations.

7.1 Lessons learned

DSLs for Testing The native support of MPS for testing different language definition aspects (e.g. type system, checking rules) helps increase testing efficiency. We have built on this positive experience by developing additional DSLs for the other aspects of MPS-based DSL implementation, such as counterexamples and debuggers. These investments paid off during the evolution of mbeddr since they drastically simplified the job of writing unit tests and keeping them up-to-date with the evolving languages.

Cost-Benefit of Different Generator Testing Strategies Executable tests proved to be very effective to test the correct implementation of generators. As programs written in the DSL, they are relatively easy to write and are similar to the code normally written by language users. Once the infrastructure is implemented, writing executable tests is easy to do and represents the best ratio of error detection vs. effort. On the other hand, assertions on the relation between input and output models are relatively difficult to define for meaningfully non-trivial relations, repeat the logic of the generator implementation and are thus less likely to find errors. In addition to those two approaches, a third strategy compares the generated code after each change

---

[7] Note: due to the way in which MPS generates Java code from the DSLs used to implement different language definition aspects (e.g. default constructors, catch-blocks) and how EMMA measures the coverage, it is often technically impossible to achieve 100% line coverage.

of a DSL with a previously recorded baseline, with subsequent manual review of differences to assess their semantic impact. The advantage of this method is that it is completely generic and thus independent of both the DSL and the generation target language, and it works even if the DSLs are not executable. The disadvantage is the high manual review effort involved when generator changes are benign.

Fuzzy Testing   Fuzzy testing proved to be a very effective means to synthesize valid models (see the discussion on "Efficiency of Models Synthesis" below) and to discover robustness defects. Approximately 10% of the randomly generated models reveal a defect of some kind, and about half of these defects are genuine bugs, relevant to the language users which should be addressed. Reviewing defects and classifying them according to their severity proved to be tedious, especially when a single defect is reflected slightly differently in several test instances.

Measuring the Coverage of Tests   We introduced systematic measurement of the test coverage relatively late in the development of mbeddr. We were surprised that despite the fact that we already had many tests, we could still find parts of the code uncovered. Furthermore, by systematically investigating the uncovered parts we realized that some of the uncovered code is actually dead code (cannot be reached at all). Lifting the EMMA based coverage information from the Java level to the level of MPS dramatically helped with reviewing the coverage information (just like any other lifting).

7.2 Variations

DSL Development vs. Compiler Construction   The ability to test languages efficiently, and the resulting need for automation, is especially prominent for DSLs. In general purpose languages, relatively few but complex languages are developed, with relatively few changes over time; they are used by a large user community. A high test effort can be justified. DSLs are usually developed for a relatively smaller user group, and languages typically evolve more rapidly. The resulting economic constraints do not allow spending a lot of effort on testing, which is why efficient, automated testing is critical.

Efficiency of Model Synthesis   A major problem with automatic testing is the difficulty of generating "sensible" random models which are correct (pass through all type-checking rules) and which reveal bugs. We have noticed that the effectiveness in obtaining valid mutants and in discovering bugs depends strongly on the chosen seed model and on the allowed mutation depth. The highest effectiveness was reached when we looked for specific types of bugs (e.g., bugs with the chosen type of the argument of events of state machines) and when we allowed only "shallow" mutations (i.e. fuzzing). This assumes that the set of manually written seeds have a good coverage of the language. In the cases when we allowed "deep" mutations, the algorithm is much slower

(many mutants are discarded as being incorrect w.r.t. the checks of mbeddr) and the rate of identified bugs is smaller. Nonetheless, over all we consider model fuzzing useful and efficient for finding defects.

7.3 Challenges

Specifying the Semantics of DSLs   Testing the implementation of a DSL assumes that the semantics of the DSL are well-defined. One of the biggest challenges in this context we faced is the lack of pragmatic means and methodologies that support the formal specification of DSL semantics. The assertions in the test cases represent only small parts of a full semantic specification. As a consequence, the single point of truth about the semantics is implied by the translators of the DSL to the target language. Having the semantics specified only implicitly by generators makes it very difficult (or even impossible) to ensure its completeness or its consistency between different tools which interpret the model.

Testing Modular and Composable Languages   Given a set of existing languages $L = \{l_1, \ldots, l_n\}$, our goal is to test how a new language l' that extends one or more of languages from L integrates with the existing languages from L. To this end, we must test both the constructs of l' in isolation, as well as the integration with the languages from L. Modular extension and integration of languages is easily possible in MPS, but tools and practices for testing in such scenarios are still in an early stage. Testing combinations of uses of different languages in the same model is very expensive.

Testing in an Open World   In MPS, languages can be extended without the need to modify the original languages. When a DSL is designed, it is impossible to anticipate all of its future extensions. MPS thus assumes an "open world" view of languages – unless an extension or composition is explicitly forbidden, it is by default allowed. In this context, it is a challenge to plan in advance a set of tests when a language is extended by other languages. A step forward would be to use a design-by-contract methodology in DSLs development and thereby make it possible for each DSL to specify assumptions from the surrounding DSLs and the offered guarantees.

Testing Domain Specific Tooling   DSLs are enablers for a wide range of domain specific tools such as analyzers, debuggers or interpreters. Each of these tools have an (implicit) interpretation of the semantics of the developed language. This requires ensuring that the different tools are semantically aligned with each other, and with the code generators. This heterogeneity makes testing the alignment of the interpretation of the languages' semantics challenging. The absence of pragmatic means to explicitly define the DSLs semantics makes the testing of alignment between different tools ad-hoc and in general unsatisfactory.

Increasing the Degree of Testing Automation This paper presents our struggle with automating the testing of a range of language implementation aspects of mbeddr. There are cases in which the automation can be further increased, such as testing the generators, testing the tool UI and construction of models with a UI testing tool. The biggest challenge with testing the generators is to find better oracles which capture the semantics of the generator. A possibility would be to have an executable semantics of the language defined in an independent manner from the generator (e.g. an interpreter) and then compare the semantics of the generated code with the semantics of the high-level models. We have used this approach with other (simpler) languages in the medical domain; however, for C, developing an interpreter that is faithful to the semantics in the details was decided to be too much work.

Economics of Testing in DSLs Development Language workbenches like MPS enable language engineers to develop languages in a very efficient manner. Testing these languages is however not supported to the same extent. While in classical software development projects the testing accounts for ca. 40% of the total efforts [12], due to the efficiency of language development with MPS and lack of equal support for testing, that ratio is much higher. Further research is needed to improve the effectiveness of testing and to assess empirically the effectiveness of the methods developed in this paper in the context of other large-scale DSL projects.

Formal Verification Testing has well known limitations with respect to the covered input space. A further step for increasing the confidence in the DSLs implementation would be the use of formal verification techniques. In the case of mbeddr, a stack of extensions over the C language, this would imply a huge effort for specifying the semantics of the languages (including C) in order to verify the generators. Currently, despite valuable published works which present promising results that could be relevant for our case ([6, 11]), we are not aware of user friendly tool support which would allow us to use these techniques on the mbeddr project. As we have already said in the introduction, we did not have enough time, skills and tooling for this to be feasible. It would also be a different approach, and a different paper.

7.4 Threats to Validity

Internal Validity The development of many of the techniques presented in this paper was motivated by the need to assure the quality of the implementation of mbeddr. All of the presented techniques are used to some extent in different parts of mbeddr, but the extent to which they are used varies significantly. For example, executable tests (described in Section 3.5) cover all languages and generators, whereas testing the generators by comparing to a baseline (described in Section 3.6) has been used so far only for some of the languages due to the manual efforts required to re-validate generated code when small changes in the generator occur.

External Validity   Our approach and experience is mostly based on the JetBrains MPS language workbench; it is different from most other language workbenches mostly because of its reliance on projectional editing. Consequently, some of the proposed methods and lessons learned are specific to MPS' approach of developing languages and tooling and might have to be adapted for other language workbenches. In particular, for parser-based tools such as Xtext, one has to test the grammar definition, whereas in MPS one does not (as we described in Section 3). Furthermore, the presented paper presents only the lessons learned from testing mbeddr, which is an instantiation of MPS.

The techniques were developed in a generic way to enable their use with other languages and tools as well. Using existing testing strategies drastically lowers the effort for implementing future languages with MPS. Our experience showed that the DSLs testing approaches presented in this paper can be used for testing the implementation of other DSLs as well – i.e. the techniques proved to be very valuable for testing other DSLs which have been developed outside of the mbeddr project (e.g. the DSLs stack built as part of the Embedded Software Designer tool developed by Siemens).

We believe that these techniques are relevant for other language workbenches – not necessarily projectional – but this has to be proven through further case-studies.

8 Related Work

To the best of our knowledge there is no other work on testing the implementation of languages and tooling in a holistic manner as discussed in this paper. Furthermore, there is no work on industrial experience with testing DSLs and their tools. Last but not least, literature about language testing does not deal with projectional editors. However, considering testing of specific aspects of DSLs, many papers overlap with ours. We discuss some of them below.

Language Testing and Language Workbenches In their seminal work [34], Wu et al. describe a unit testing framework for testing the behavior of DSLs. They develop a framework to lift test results from the level of the target programing language to which the DSL is translated, back to the level of the DSL. This way the testing process is eased for the language developers. Our work extends this approach by providing testing DSLs for all aspects of a language definition, e.g., editor (concrete syntax), type system, scoping, transformation rules, and finally the debugger.

Kats et al. describe language testing in Spoofax [13]. Spoofax supports testing parsers, abstract syntax tree construction, type systems and transformations; essentially the same features that we describe for MPS in Section 3. As a consequence of Spoofax' language composition support, some of these tests can be expressed using the concrete syntax of the subject language. However, no special DSL extensions for testing domain specific tools (as in Section 4) and no automation of language tests (as in Section 5) in described.

Xtext, another popular tool for DSL development, does not specifically support the testing of language implementations. However, the Xpect [8] add-on can be used to test non-syntactic aspects such as scopes and type systems (essentially everything that annotates error markers onto the subject program). Xpect annotations are expressed in comments as to not interfere with the syntax of the subject language. Also, since many aspects of DSL are implemented using Java APIs and frameworks, the normal means for testing Java code can be used (albeit possibly not very efficiently because of the low level of abstraction of the tests and the implementation).

Compared to the approaches used by Xtext and Spoofax, we focus on testing the implementation of DSLs and tooling developed with projectional editors. Furthermore, we present a set of extensions of MPS which were needed in order to further automate testing in the context of the mbeddr project.

Compiler Testing   In [36] Yang et al. present a seminal work on testing C compilers by randomly synthesizing C programs. Errors which occur during compilation reveal robustness defects in the compiler. This is similar to our approach to robustness testing. Furthermore, the programs are compiled by different compilers and then tested using randomized differential testing. If different executables produce different outputs then one of the used compilers has a bug in the translation. This approach tests the semantics (as opposed to the structure of the generated code). It works so well here because the same program is executed on *several* compilers that are all supposed to generate programs with identical semantics. For DSLs, this approach is usually not available, because usually, there is only *one* generator for the language. If however, multiple generators or an interpreter plus a generator are available, this approach leads to very good results; as mentioned, we have used it with other, simpler DSLs.

Grammar Testing and Program Synthesis   There is considerable work on generating models in order to test grammars or model transformations. [14] represents a seminal work in testing grammars. [35] is a literature review of meta model-based instance generation. The problem of guiding a random models generation algorithm to come up with a higher percentage of useful models is also addressed in the literature [16,9]. Our work is based on MPS, which supports abstract-syntax-oriented definition and composition of languages. Besides being in a different technological space, our work takes a more comprehensive view of language testing: we test multiple language aspects such as abstract syntax, context-sensitive constraints, generators and tooling. Testing the abstract syntax, or grammar, is only part of the overall testing effort. Furthermore, due to the modularity and composability of languages in MPS, we face a bigger challenges for testing the interactions of implementations of different languages. We plan to extend in the future our random models synthesis with techniques for guiding models synthesis towards more meaningful models.

Testing Model Transformations Amari and Combemale present a comprehensive overview of properties on model transformations that can be verified [2]. Currently, there is no formal verification approach for MPS generators, and due to the fact that MPS allows the use of Java within its high-level transformation language, the formal verification is hard to implement. Instead, our approach relies on testing: we synthesize models, run the generators and check various properties.

Testing Debuggers There is a plethora of work around testing debuggers for general purpose languages. For example, the tests for the GNU Debugger (GDB) debugger cover different aspects of the debugger functionality and are written in a scripting language [10]. The GDB project tests debugging behavior for all of its supported languages, such as C, C++, Java, or Ada. In contrast to GDB testing, our work focuses on testing debugging functionality of extensible DSLs. Furthermore, we have created the DeTeL language which allows engineers to directly annotate models written in the high-level DSL with the information about expected behavior of the debugger.

The moldable debugger framework [4] is used for adapting debuggers to a particular domain allowing developers to analyze their software systems on the domain level with custom debug operations and views. Debuggers built with this framework are tested by using manually written test cases. In our work, we provide a DSL to write functional debugger tests.

9 Conclusion and Future Work

Modern language workbenches offer means to develop complex domain specific languages and tooling very efficiently. Adequate testing of these DSLs is of paramount importance for their large scale adoption in industrial practice. Furthermore, DSLs evolve quickly, and to avoid regressions, a high test coverage and a high degree of automation is essential. The situation becomes more challenging when modular languages are composed without a priori knowledge.

In this paper we described our work with testing the different aspects of DSLs. The work is directly motivated by the need to test the implementation of the mbeddr language stack built with MPS. All techniques and methods presented in the paper were applied to some degree for testing mbeddr. We presented a classification of defects of the implementation of DSLs and ways in which the testing for these defects can be automated. We describe the out-of-the-box features provided by MPS for testing language implementations. We then introduce additional testing support for testing the domain specific tools such as debuggers or analyzers. We further present a fully automatic approach to testing the completeness of constraints and type system checks and the robustness of generators that relies on automatic synthesis of models and the use of a variety of test oracles in the transformation-generation-compilation pipeline. For each approach, we presented our experience, lessons learned, variation points and challenges.

While we are convinced that our work on increasing the degree of automation for testing DSL implementations is the right direction, we also believe strongly that there are many additional steps that can be taken to increase the efficiency and effectiveness of testing DSL implementations. In particular, language workbenches should integrate techniques such as program synthesis, fuzzing or coverage measurement directly such that they can be used out-of-the-box for testing DSLs. This way, language testing can be made just as efficient as language implementation. Furthermore, a big part of the automation described in this paper is about automating the execution of tests, whereas automatied testing is much broader and involves the automatied synthesis of models, automated oracles or defects classification. On the mbeddr side we are continuously working on extending the set of available tests. We also plan to integrate the fuzzy-testing approach in the nightly build and use the entire set of hand-writing tests as seeds for fuzzy-ing. We are continuously exploring additional techniques that can be applied for testing in order to increase confidence in its reliability. This happens within the context of mbeddr, but also for other languages developed as part of our teams.

References

1. JetBrains MPS Documentation. https://www.jetbrains.com/mps/documentation/.
2. M. Amrani, B. Combemale, L. Lucio, G. M. K. Selim, J. Dingel, Y. L. Traon, H. Vangheluwe, and J. R. Cordy. Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, 14(3):1:1–43, 2015.
3. F. Campagne. *The MPS Language Workbench*. CreateSpace Publishing, 2014.
4. A. Chis, M. Denker, T. Gîrba, and O. Nierstrasz. Practical domain-specific debuggers using the moldable debugger framework. *Computer Languages, Systems & Structures*, 44:89–113, 2015.
5. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, - 10th International Conference*, 2004.
6. C. M. Ellison III. *A formal semantics of C with applications*. University of Illinois at Urbana-Champaign, 2012.
7. S. Erdweg, T. Storm, M. Völter, et al. The State of the Art in Language Workbenches. In *Software Language Engineering*, LNCS. Springer, 2013.
8. M. Eysholdt. Executable specifications for xtext. Website, 2014. `http://www.xpect-tests.org/`.
9. B. Fetscher, K. Claessen, M. Pałka, J. Hughes, and R. B. Findler. *Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System*, pages 383–405. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
10. Free Software Foundation. The GNU Project Debugger, 2015.
11. A. Gargantini, E. Riccobene, and P. Scandurra. A semantic framework for metamodel-based languages. *Automated Software Engg.*, 16(3-4):415–454, Dec. 2009.
12. V. Garousi and J. Zhi. A survey of software testing practices in canada. *J. Syst. Softw.*, 86(5):1354–1376, May 2013.
13. L. C. Kats, R. Vermaas, and E. Visser. Integrated language definition testing: enabling test-driven language development. In *ACM SIGPLAN Notices*, volume 46, pages 139–154. ACM, 2011.
14. R. Lämmel. Grammar testing. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, 2001.

15. Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific c verification with mbeddr. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 539–550. ACM, 2014.
16. M. H. Pa, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.
17. D. Pavletic, S. A. Raza, K. Dummann, and K. Haßlbauer. Testing Extensible Language Debuggers. In T. Mayerhofer, P. Langer, E. Seidewitz, and J. Gray, editors, *Proceedings of the 1st International Workshop on Executable Modeling co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 27, 2015.*, volume 1560 of *CEUR Workshop Proceedings*, pages 34–40. CEUR-WS.org, 2015.
18. D. Pavletic, S. A. Raza, M. Voelter, B. Kolb, and T. Kehrer. Extensible Debuggers for Extensible Languages. *Softwaretechnik-Trends*, 33(2), 2013.
19. D. Pavletic, M. Voelter, S. A. Raza, B. Kolb, and T. Kehrer. Extensible debugger framework for extensible languages. In J. A. de la Puente and T. Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2015 - 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings*, volume 9111 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2015.
20. D. Ratiu and A. Ulrich. Increasing usability of spin-based c code verification using a harness definition language: Leveraging model-driven code checking to practitioners. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017.
21. D. Ratiu and M. Voelter. Automated testing of dsl implementations: Experiences from building mbeddr. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, New York, NY, USA, 2016. ACM.
22. D. Ratiu, M. Voelter, B. Kolb, and B. Schaetz. Using language engineering to lift languages and analyses at the domain level. In *Proceedings the 5th NASA Formal Methods Symposium (NFM'13)*, 2013.
23. D. Ratiu, M. Voelter, B. Kolb, and B. Schätz. Using language engineering to lift languages and analyses at the domain level. In *NASA Formal Methods, 5th International Symposium*, 2013.
24. D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb. Language engineering as enabler for incrementally defined formal analyses. In *Proceedings of the Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FORMSERA'2012)*, 2012.
25. J.-P. Tolvanen and S. Kelly. Model-driven development challenges and solutions - experiences with domain-specific modelling in industry. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development - Volume 1: IndTrackMODELSWARD,*, pages 711–719, 2016.
26. M. Voelter. Language and IDE Development, Modularization and Composition with MPS. In *Generative and Transformational Techniques in Software Engineering*, Lecture Notes in Computer Science. 2011.
27. M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering*. dslbook.org, 2013.
28. M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. *Using C language extensions for developing embedded software: A case study*, volume 50. ACM, 2015.
29. M. Voelter, B. Kolb, T. Szabó, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, Jan 2017.
30. M. Voelter and S. Lisson. Supporting diverse notations in mps'projectional editor. In *GEMOC@ MoDELS*, pages 7–16, 2014.
31. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 20(3):339–390, 2013.
32. M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proceedings of SPLASH Wavefront 2012*, 2012.
33. M. Voelter, D. Ratiu, B. Schätz, and B. Kolb. mbeddr: an extensible C-based programming language and IDE for embedded systems. In *SPLASH '12*, 2012.

34. H. Wu, J. G. Gray, and M. Mernik. Unit testing for domain-specific languages. In W. M. Taha, editor, *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL 2009, Oxford, UK, July 15-17, 2009, Proceedings*, volume 5658 of *Lecture Notes in Computer Science*, pages 125–147. Springer, 2009.
35. H. Wu, R. Monahan, and J. F. Power. Metamodel instance generation: A systematic literature review. *Computing Research Repository (CoRR)*, 2012.
36. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.