Automated Testing of DSL Implementations

Experiences from Building mbeddr

Daniel Ratiu
Siemens CT
daniel.ratiu@siemens.com

Markus Voelter
independent/itemis
voelter@acm.org

ABSTRACT

Domain specific languages promise to improve productivity and quality of software by providing problem-adequate abstractions to developers. Projectional language workbenches like JetBrains MPS allow the definition of modular and extensible domain specific languages, generators and development environments. While recent advances in language engineering have enabled the definition of DSLs and tooling in a modular and cost-effective manner, the quality assurance of their implementation is still challenging. In this paper we present our work on testing the implementation of domain specific languages and associated tools, and discuss different approaches to increase the automation of language testing. We illustrate this based on MPS and our experience with testing mbeddr, a set of domain specific languages and tools on top of C tailored to embedded software development.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Software Quality Assurance

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. Recent advances in language workbenches, the tools used implementing language engineering and associated IDEs, enable a more productive approach to building domain specific tools around DSLs. Besides 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 domain specific and thus can take advantage of the abstractions in the DSL.

Language Workbenches. Language Workbenches 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 development. In a sense, they move language development from the domain of computer science into the domain of software engineering. **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.**

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. Since languages and their implementations are "just" software artifacts, they must be quality assured as well. As languages become more complex, more widespread and developed in shorter cycles, systematic quality assurance is essential. For example, mbeddr[1] [13] is a set of 81 languages and C extensions for embedded software development built using the JetBrains MPS language workbench. It has 1,418 tests in total, with 3,646 assertions. This corresponds to approximately 45,000 lines of code, and is about 50% of the size of the mbeddr implementation itself.

In this paper we focus on the following categories of defects of the implementation of a DSLs and associated tooling:

D1, Front-end Defects The front end of a DSL relates to the definition of the context sensitive constraints and static semantics (type system). If the constraints or typing rules are insufficient, they allow language users to define models which are not meaningful; when they are badly implemented, runtime-errors (e.g. exceptions) occur in the IDE, or the generators may fail or produce non-compiling code.

D2, Back-end Defects Once models are authored, generators are used to synthesize other models, typically with progressively lower levels of abstraction. Due to the abstraction gaps, or because of optimizations implemented in the generators, generators are the most complex part of the language definition and are thus error prone. For example, generators might simply crash on certain input models, they might produce output which does not conform to the target language or us semantically invalid.

D3, Defects of Domain Specific Tooling Domain specific constructs open new possibilities for domain specific

AST'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2896921.2896922

[1] http://mbeddr.com

tools, such as analyzers or debuggers. They are often complex and depend on models at various abstraction levels. Analyzers are prone to errors such as semantic mis-alignment with the generators, or misinterpretation of models.

Contribution. In this paper we categorize various aspects of the quality of DSL implementation and associated tooling that must be tested. We introduce a holistic approach for testing languages implemented in MPS. In particular, we develop an approach to automate the testing of different language and tooling implementation aspects.

Structure of the paper. In the next section we present an overview over MPS and its support for testing different aspects of a DSL implementation. In Section 3 we present our approach to increase the automation of language and tooling testing. At first we focus on domain specific tooling, namely the integration of the CBMC model checker in mbeddr. We then present our strategy to automate testing by synthethising random models (that respect the structures defined by the meta model) and using increasingly complex oracles to detect bugs in the implementation of different language aspects. In Section 4 we discuss the main lessons learned and challenges when testing mbeddr. In Section 5 we present related work, and then conclude the paper.

2. LANGUAGE DEVELOPMENT AND TESTING IN MPS

In MPS, a language implementation consists of several aspects. At the core, these include structure, concrete syntax, constraints, type system and transformations. Additional aspects include data flow, debuggers, interpreters and various specific IDE extensions.

For each of these aspects, MPS ships with a dedicated language; 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 [5].

For reasons of brevity, we will not describe in detail any of these aspect-specific languages and refer the reader to [12, 1, 3]. However, we provide some intuition in the rest of this paragraph. All of the aspect DSLs embed Java expressions or statements but provide different high-level paradigms that are aspect-specific. The structure language is essentially a declarative meta modeling language. The language for the concrete syntax relies on instantiating, composing and parameterizable editor cells. The type system uses typing equations solved by a solver. Transformations use various kinds of templates to map a source AST to a target AST.

In the remainder of this section we discuss the facilities for language testing provided by MPS out of the box.

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. 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 code is rendered with a red background as shown in Fig. 1. 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.

Error Reporting. This kind of test verifies whether MPS detects non-structural programming errors correctly; in other words, they verify that the red squigglies appear on the nodes that have errors. This can be used to test type system rules (an example is shown in Fig. 2), but also any other error message resulting from constraints or checking rules. 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.

Program Inspection. The `NodesTestCase` used for verifying the correct annotation of error messages in Fig. 2 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. 3 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. 3, 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.

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 node specified in the test case.

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

Figure 1: 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.

```
Test case testingTypeChecksForLocalVariables
([ void f() {
     int8 anInt = 42;
     int8 aFloat = <check 3.3 has error>;
     int8 aString = <check "hello" has error>;
   } f (function) ])
```

Figure 2: Testing type system rules is done by writing code that provokes the error, and then asserting that the error actually occurs.

Testing Dynamic Semantics. We want to test that the results of the execution of a model are as expected by the developer when writing the program. In mbeddr, since extensions define their semantics by transformation to C, we test this by generating the mbeddr program to C and then execute the C code. If semantics are implemented by an interpreter, that interpreter can be called from a nodes test case, and asserting over the resulting behavior as part of a test case similar to the one shown in Fig. 3

Semantics testing requires ability the to express the expected behavior (oracle) in the DSL programs. Thus, one of the first C extensions we have developed for mbeddr was the support for unit tests. As the right half of Fig. 4 shows, mbeddr test cases are almost like `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. 4 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 transformations; 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 and they test whether the mbeddr generator 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. INCREASING AUTOMATION IN MPS

This section contains our core contribution: we discuss our approach to increase the automation of testing languages and their associated tools in MPS.

```
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);
  }
```

Figure 3: This test case asserts that the structure of linearly entered expression respects the precedence specified for the + and * operators.

```
statemachine SM initial = S1 {
  in event e()
  in event f()
  state S1 { on e [] -> S2 }
  state S2 { on e [] -> S3 }
  state S3 { on e [] -> S1 }
}
```

```
exported testcase testState {
  SM sm;
  assert(0) sm.isInState(S1);
  test statemachine sm {
    e ➔ S2
    f ➔ S2
    e ➔ S3
    e ➔ S1
} }
```

Figure 4: An example state machine plus a test cases that uses a state machine-specific C extension to express the test for the state machine.

3.1 Languages for Testing Specific Aspects

As mentioned before, MPS 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 dramatically ease the definition of tests and their review.

Experience: Testing Counterexamples. mbeddr integrates formal verification based on the CBMC [4] model checker [11]. Our approach uses domain specific languages for property specification and environment definition. The verification itself is performed on the generated C code. A CBMC-based verification in mbeddr consists of the following steps: 1) mbeddr code is generated to C + CBMC macros, 2) CBMC is run on the generated code and produces a counterexample at C level, 3) the results and counterexamples are lifted back in mbeddr and each lifted step is associated with a node from the original input program.

Fig. 5 illustrates this approach. In the first (left-most) column we show a simple state machine and a verification environment, column two shows the C code generated from the high-level models, column three illustrates the counterexample at C level produced by CBMC, and the last column shows 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 [9].

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 model element from which that code originates. During the early stages of mbeddr development we used a classical approach using JUnit, but the test cases were hard to write and maintain. We then decided to describe the desired shape of counterexamples, as seen by users, by using the DSL illustrated in Figure 6. The counterexample is translated into Java for integration with JUnit and the MPS UI. 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.

Experience: Testing Debuggers. We have also defined a DSL for testing debuggers, and in particular their step over, step into and step out behaviors. The DSL is used to specify which statement executes next when any of these steps are taken. For further details please refer to [10].

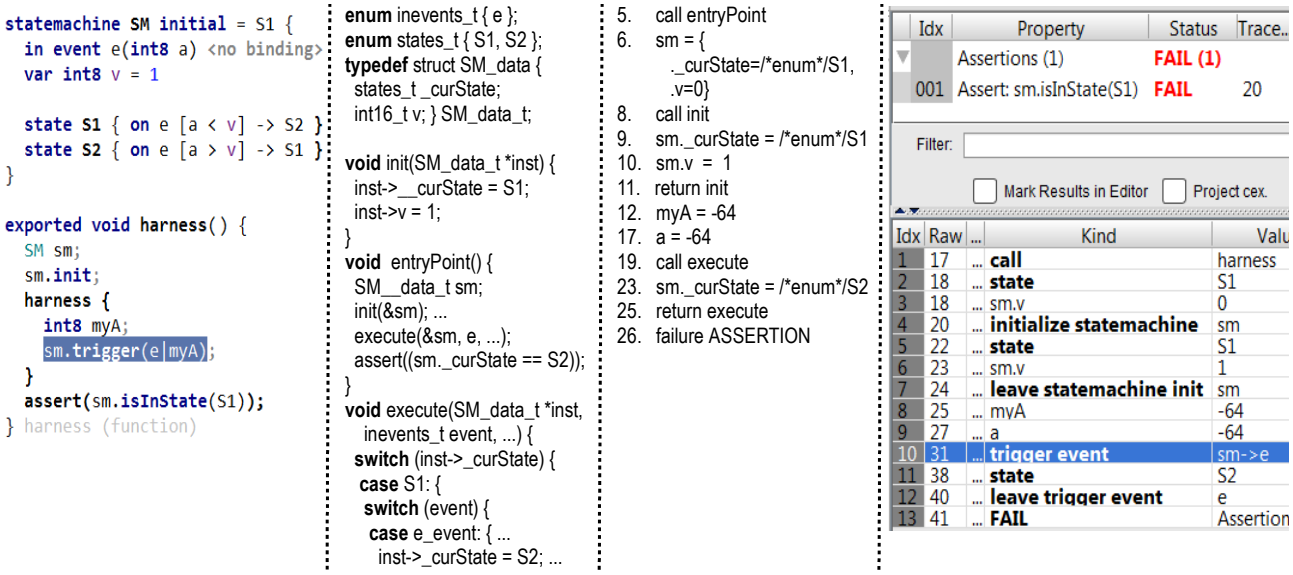

Figure 5: High-level model (left); generated C code; C-level counterexample; lifted counterexample (right)

```
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"));
```

Figure 6: Example of counterexample DSL (top) and its translation into Java (bottom).

3.2 Automated Testing

To increase testing productivity, and hence to increase the quality of language implementations, we increase automation. In particular, we rely on automatically synthesizing mbeddr models and then using several oracles 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 meta-model).
2. Run the context-sensitive constraints and type system checks against these models. Models which are checked correct are 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 found a bug: the generator has produced non-compilable C code for a valid model.
5. If code compiles successfully, we then evaluate a suite of assertions defined by the language engineer that check the functional requirements on the generator by relating input and their corresponding output models.

3.2.1 *Automatic Model Synthesis*

We generate test vectors (i.e., input models) by using concepts from the languages under test. If the language under test can be composed with other existing languages (an crucial feature of MPS and language engineering in general), we test them together by generating models which cover the space of possible compositions.

Experience. mbeddr's implementation of C has 317 meta-model classes, 116 interfaces, as well as 170 relations between them. The state machine language has 41 classes, 7 interfaces, and 38 relations. The state machine language can be composed with C – e.g. state machines can be declared in C modules, C expressions can be used in guards, and C types can be used as in events arguments. For big languages such as mbeddr (including C and all of its extensions), generating non-trivial models requires dealing with an explosion in complexity because the space of constructs and their possible combinations is large. We address this by using existing, manually written models as seeds for the mutation algorithm.

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

Figure 7: Example of a testing configuration

Fig. 7 shows an example of a configuration for generating models. Given a set of models as `starting point`, the algorithm tries for a certain `number of steps` to mutate these models. To perform a model mutation, it randomly chooses a node from within this model and of a certain type (`seed chooser`). It replaces it with another subtree which is synthesized up to a certain `depth` by using meta-classes from the set of `languages of interest`. In mbeddr, we use the models that have been manually written for unit tests of the languages as starting points for obtaining new test-vectors by mutation; this way, no additional effort has to be spent on writing the starting point models.

3.2.2 Robustness of the Front-End

After a model is synthesized, we exercise the functionality of the language (e.g., constraints, type system). We expect that these checks are performed successfully, in the sense that no runtime errors occur. Otherwise we have identified a potential bug in the type checker or constraints, respectively. The models causing these runtime errors must be manually checked – sometimes a particular model cannot be constructed by the user as a consequence of restrictions in the editor, so the error can never occur with end users.

Experience. Before starting with automatic testing, we had a high confidence in the front-end because the errors caused here are "very visible" while using mbeddr; we assumed that most had been found and reported. To our surprise, the automated tests identified several tens of errors for the mbeddr core languages. This is because the random model synthesizer produces models that a typical user may not think of. In Fig. 8-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. 8-right shows the original front-end code as well as the fix. Another

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

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

```
link {event} scope:
  (exists, referenceNode, contextNode, enclosingNode
    node<GenericDotExpression> gde;
    gde = enclosingNode : GenericDotExpression;
    node<> tpe = gde.expression.type;
    return tpe : StatemachineType.machine.inEvents();
  }
```

```
link {event} 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>;
 }
```

Figure 8: Example of defects in the implementation of the front-end.

example of a major bug is when the `typedef` was used in a circular manner (i.e. `typedef a b; typedef b a;`) then mbeddr froze. Of course, circular `typedef`s are not valid C code but mbeddr should report an error and not freeze.

3.2.3 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. 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 or the constraints in the language are insufficient, failing to reject invalid models.

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 hundreds of models, we identified about a dozen robustness issues with their generator. Most of there issues originate from the too weak constraints in the front-end of mbeddr which do not reflect all assumptions made by the generators.

3.2.4 Structural Correctness of Generated Models

It is possible that the output models are not meaningful because they do not comply with the target language definition; 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 identified that most of them originate from lack of constraints in the font-end.

3.2.5 Semantic Correctness of the Generator

To finally ensure semantic correctness (beyond the structural correctness discussed above) we must check the relation between the input models and the target models (similar to black-box testing). To do this, we allow language developers to define assertions that capture the intent of a generator. These assertions originate from the specification of the DSL about how it should be translated into the target language. The assertions, described in a declarative way, can easily capture a part of the semantics of the generator; we provide an example in the next paragraph.

Experience. Generation in MPS is achieved through a sequence of generation steps. Each step transforms a higher-level model into lower-level one until the target language is reached; then, a text generator creates the final output for subsequent compilation. For checking the correctness of the generator in mbeddr, we define assertions on the relation between an input model and the last model from the generation stack. In Figure 9 we present an assertion which

```
foreach Statemachine : sm from: originalModel.nodes(Statemachine)
  exists Function : fun from: outputModel.nodes(Function)
    fun.name.equals(" execute__" + sm.name)
  exists StructDeclaration : sd from: outputModel.nodes(StructDeclaration)
    sd.name.equals(sm.name + "__data")
    foreach StatemachineVariableDeclaration : svd from: sm.localVariables()
      exists Member : mem from: sd.descendants<concept = Member>
        mem.name.equals(svd.name)
```

Figure 9: Example of assertions about relation between input and output models.

checks that for each state machine, an `execute` function and a `struct` declaration are generated, and that for each state machine variable declaration a member is generated into this `struct`. To our surprise, our experience so far with the effectiveness of these semantic checks shows that only very few additional bugs can be identified. A possible explanation is that we have already had a significant number of executable unit tests (like the one presented in Fig. 4) which make sure that the generators work correctly.

4. DISCUSSION

Measuring the Quality of Tests. A common criterion for measuring test quality is the structural coverage of the to-be-tested code. Since the mbeddr languages are mostly developed using MPS-provided DSLs, we measure the structural coverage with respect to the implementation written with these DSLs. We follow a staged approach: 1) concept coverage – ensure that all meta-model classes of the language(s) of interest are instantiated in tests; 2) context sensitive rules – make sure that all constraints, checking and type-system rules from the language definition are used in tests; 3) the coverage of generators; and 4) the coverage of the tooling. MPS provides direct support for 1) and 2). For 3) and 4), we measure the coverage of the Java code[2] which is ultimately generated from all DSLs which implement the languages.

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 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.

DSL Development vs. Compiler Construction. The ability to test languages efficiently, and the resulting need for automation, is especially prominent for DSLs. In GPLs, 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 easily 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 automation is critical.

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 a the 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.

Effectiveness of the Synthesized Models. 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 most effective situations were 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. This assumes that the set of manually written seeds is representative for 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.

5. RELATED WORK

To the best of our knowledge there is no 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. [15] is a seminal work about testing C compilers by randomly synthesizing C programs. Errors which occur during compilation reveal robustness defects in the compiler. 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 had bugs in the translation.

[7] describes language testing in Spoofax. Spoofax supports testing parsers, abstract syntax tree construction, type systems and transformations; essentially the same features that we describe for MPS in Section 2. 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 automation of language tests (as in Section 3.2) in reported in [7].

Xtext, another popular tool for DSL development, does not specifically support the testing of language implementations. However, the Xpect [6] add-on can be used to test

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

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).

Compared to these two approaches, 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 futher automate testing in the context of the mbeddr project.

Grammar Testing. Model Generation. There is considerable work on generating models in order to test grammars or model transformations. [8] represents a seminal work in testing grammars. [14] is a literature review of meta model-based instance generation. Our work is based on MPS, which supports meta-model-oriented definition and composition of languages. Besides being in a different technological space, our work takes a more comprehensive view over language testing: we test multiple language aspects such as abstract syntax, context-sensitive constraints, generators and tooling. Testing the meta-model is only one part. 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.

Testing model transformations. [2] presents a comprehensive overview of properties on model transformations that can be verified. Currently, there is no formal verification approach for MPS generators, and due to the fact that MPS allows the use of plain 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.

6. CONCLUSION AND FUTURE WORK

Adequate testing of DSLs is of paramount importance for their large scale adoption in industrial practice. Furthermore, DSLs evolve quickly, and to avoid regressions in this scenario, a high test coverage and a high degree of automation is a must. The situation become more challenging when languages are developed in a modular manner and different languages can be composed without a priori knowledge.

In this paper we described our experience with testing different aspects of DSLs. We describe the out-of-the-box features provided by MPS for testing structure, editor actions and type systems and constraints. We then introduce our approach for testing the completeness of constraint and type system checks and the robustness of generators that relies on automatic synthesis of models and the use of several test oracles in the transformation-generation-compilation pipeline.

We will continue working on refining our approach. In particular, we will work on allowing language engineers to define more complex assertions when checking the correctness of transformations.

7. 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] 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.

[5] S. Erdweg, T. Storm, M. Völter, et al. The State of the Art in Language Workbenches. In *Software Language Engineering*, LNCS. Springer, 2013.

[6] M. Eysholdt. Executable specifications for xtext. Website, 2014. http://www.xpect-tests.org/.

[7] 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.

[8] R. Lämmel. Grammar testing. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, 2001.

[9] Z. Molotnikov, M. Völter, and D. Ratiu. Automated domain-specific C verification with mbeddr. In *Proc. of the 29th ACM/IEEE Intl. Conference on Automated Software Engineering*, 2014.

[10] D. Pavletic, S. A. Raza, K. Dummann, and K. Hasslbauer. Testing extensible language debuggers. In *Proceedings of 1st International Workshop on Executable Modeling*, 2015.

[11] 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.

[12] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering.* dslbook.org, 2013.

[13] 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.

[14] H. Wu, R. Monahan, and J. F. Power. Metamodel instance generation: A systematic literature review. *Computing Research Repository (CoRR)*, 2012.

[15] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.