DeTeL - A Language for Testing Debuggers

Domenik Pavletic
itemis AG
Stuttgart, Germany
pavletic@itemis.com

Syed Aoun Raza
Stuttgart, Germany
aoun.raza@gmail.com

Kolja Dummann
itemis AG
Stuttgart, Germany
dummann@itemis.com

Kim Haßlbauer
Stuttgart, Germany
kim.hasslbauer@gmail.com

***Abstract*—Debugging is an essential aspect of any language development framework. While this also applies to extensible languages, there, debuggers are even more complex and require therefore more effort on testing. To ease testing of debuggers we propose in this paper *DeTeL* a language for testing debuggers.**

***Index Terms*—Formal languages, Software debugging, Software testing.**

I. Introduction

Software development faces the challenge that General Purpose Languages (GPLs) do not provide the appropriate abstractions for domain-specific problems. Traditionally there are two approaches to overcome this issue. One is to use frameworks that provide domain-specific abstractions expressed with a GPL. This approach has very limited support for static semantics, e. g., no support for modifying constraints or type system. Second approach is to use external Domain-Specific Languages (DSLs) for expressing solutions to domain problems. This approach has some other drawbacks: these DSLs are not inherently extensible. Extensible languages solve these problems. Instead of having a single monolithic DSL, extensible languages enable modular and incremental extensions of a host language with domain specific abstractions [1].

To make debugging extensible languages useful to the language user, it is not enough to debug programs after extensions have been translated back to the host language (using an existing debugger for the base language). A debugger for an extensible language must be extensible as well, to support debugging of modular language extensions at the same abstraction level (extension-level). Minimally, this means that users can step through the constructs provided by the extension and see watch expressions related to the extensions.

Because language extensions can be based on other extensions and languages evolve over time, it is essential to constantly test and monitor if the debugger behavior matches the expected behavior. To ensure the debugging behavior with testing, a GPL can used, however this raises the same issues discussed above. We therefore propose in this paper an extensible language for testing debuggers.

II. mbeddr

mbeddr [2] is an extensible version of C that can be extended with modular, domain-specific extensions. It is built on top of JetBrains Meta Programming System (MPS) [3] and ships with a set of language extensions dedicated to embedded software development. MPS supports the definition, composition and use of general purpose or domain-specific languages.

mbeddr includes an extensible C99 implementation. Further, it also includes a set of predefined language extensions on top of C. These extensions include state machines, components and physical units.

In MPS, language implementations are separated into modular aspects. The major aspects of a language are `Structure`, `Type System`, `Contraints`, `Generator` and `Editor`. However, for building debugging support for a language, `Editor` aspect is irrelevant.

III. Language Extension for Unit Testing

To give an idea of building language and debugger extensions, we first build a language extension for writing unit tests, which is derivded from mbeddr's `unittest` language, and the corresponding debugger extension. Later, we will describe how to test this debugger extension with a DSL.

A. Structure

Fig. 1 shows the language structure: `AssertStatement` is derived from `Statement` and can therefore be used where `Statements` are expected. It contains an `Expression` for the *condition*. `TestCase` holds a `StatementList` that contains the `Statements` that make up the test. Further, to have the same scope as `Function` `TestCase` implements `IModuleContent`. `ExecuteTestExpression` contains a list of `TestCaseRef`, which refer to `TestCases` to be executed.

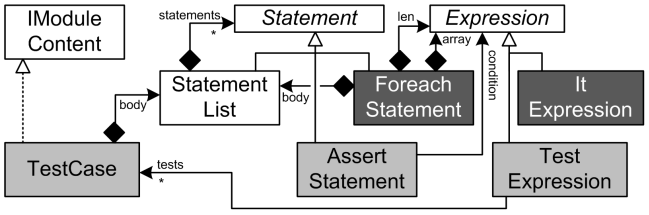

Fig. 1. Language structure

B. Type System and Constraints

`AssertStatemnt` requires a constraint and a type system rule. First restricts the usages only inside `TestCases`, meaning an `AssertStatemnt` can only be used in a `TestCase`:

```
parentNode.ancestor<concept = TestCase, +>.isNotNull
```

Second restricts the type of its child `expr` (*condition*) to `BooleanType`, so only valid *conditions* can be entered:

```
check(typeof(assertStatement.expr) :<=: <BooleanType()>);
```

`ExecuteTestExpression` returns the number of failed unit tests, hence we specify `Int32tType` as its type (see rule below). Later the same type is used in the generator.

```
check(typeof(executeTestExpression) :==: <Int32tType()>);
```

C. Generator

The unit test generator consists of many different transformation rules, which translate code written with the language directly to mbeddr C. Listing 1 shows on the left hand side an example program, written with mbeddr C and the unit test language. The right hand side shows the C program generated from it. While regular mbeddr code is not colored, the boxes show how Abstract Syntax Tree (AST) nodes from the left are translated to C code on the right.

```
int32 main(int32 argc,
    string[] argv) {
  return test[ forTest ];
}



testcase forTest {

  int32 sum = 0;
  assert: sum == 0;
  int32[] nums = {1, 2, 3};
  for(int32_t i=0;i<3;i++){
    sum += nums[i];
  }
  assert: sum == 6;

}
```

```
int32_t main(int32_t argc,
    char *(argv[])) {
  return blockexpr_2();
}

int32_t blockexpr_2(void) {
  int32_t _f = 0;
  _f += test_forTest();
  return _f;
}

int32_t test_forTest() {
  int32_t _f = 0;
  int32_t sum = 0;
  if(!( sum == 0 )) { _f++; }
  int32_t[] nums = {1, 2, 3};
  for(int32_t i=0;i<3;i++){
    sum += nums[i];
  }
  if(!( sum == 6 )) { _f++; }
  return _f;
}
```

Listing 1. Example mbeddr program using the unit test language on the left and the C code that has been generated from it on the right. Empty lines on the left have no meaning, they are used for showing the relationship between high-level and generated code.

IV. MBEDDR DEBUGGER FRAMEWORK

mbeddr comes with a debugger, which allows users to debug their mbeddr code on the abstraction levels of the used languages. For that, each language contributes a debugger extension, which is build with a framework also provided by mbeddr [4]. Debugging support is implemented specificly for the language by lifting the call stack/program state from the base-level to the extension-level (see Fig. 2) and stepping/breakpoints vice versa.

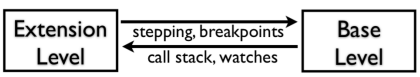

Fig. 2. Flow of debug information between base and extension level [4]

The debugger framework can be separated into two different parts: First, a DSL and a set of interfaces (shown in Fig. 3) for specifying the debugging semantics of language concepts. Second, a runtime for executing those specifications and thus achieving the mapping described in Fig. 2.

In this paper, we focus only on the specification part (see Fig. 3) required for understanding how the debugger extension for unit testing is build. Further details about the architecture and its implemention with MPS are described in [4].

A. Breakpoints

`Breakables` are concepts (e. g., `Statements`) on which we can set breakpoints to suspend the program execution.

B. Watches

`WatchProviders` are translated to low-level watches (e. g., `GlobalVariableDeclaration`) or represent watches on the extension-level. They are declared inside `WatchProviderScopes` (e. g., `StatementList`), which is a nestable context.

C. Stepping

`Steppables` define where program execution must suspend next, after the user *steps over* an instance of `Steppable` (e. g., `Statement`). If a `Steppable` contains a `StepIntoable` (e. g., `FunctionCall`), then the `Steppable` also supports *step into*. `StepIntoables` are concepts that branch execution into a `SteppableComposite` (e. g., `Function`).

All stepping is implemented by setting low-level breakpoints and then resuming execution until one of these breakpoints is hit (approach is based on [5]). The particular stepping behavior is realized through stepping-related concepts by utilizing `DebugStrategies`.

D. Call Stack

`StackFrameContributors` are concepts that have callable semantics on the extension-level or are translated to low-level callables (functions or procedures). While the latter do not contribute any `StackFrames` to the high level call stack, the former contribute at least one `StackFrame`.

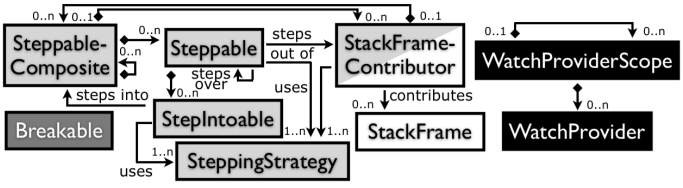

Fig. 3. Meta-model used for specying the debugging semantics of language concepts [4]. Colors indicate the different debugging aspects.

V. DEBUGGER EXTENSION FOR UNIT TESTING

This section describes the implementation of a minimal debugger for the unit testing language, which is defined with mbeddr's debugger specification DSL and the abstractions shown in Fig. 3.

A. Breakpoints

To enable breakpoints on instances of `AssertStatement`, an implementation of the `Breakable` interface is required. Since `AssertStatement` is derived from `Statement` that already implements this interface, thus breakpoints can be set on `AssertStatement`.

B. Call Stack

`TestCase` and `ExecuteTestExpression` implement `StackFrameContributor` that are concepts translated to base-level callables. They contribute StackFrames, each is linked to a base-level stack frame and states whether it is visible in the extension-level call stack or not.

The implementation of `ExecuteTestExpression` links the low-level stack frame to the respective instance (see listing below). Additionally, it declares to hide the stack frame from the high-level call stack.

```
contribute frame mapping for frames.select(name=getName());
```

Similarly the mapping for `TestCase` also requires linking the low-level stack frame to the respective instance. However, it declares to *show* the stack frame in the high-level call stack. Further, we provide the name of the actual `TestCase`, which is represented in the call stack view: Consider Listing 1, where we would lift the name `test_forTest` to `forTest`.

C. Stepping

`AsserStatement` is a `Statement`, which already provides *step over* behavior. However, to be able to *step into* the *condition* we overwrite `Statement`'s *step into* behavior:

```
break on nodes to step-into: this.expr;
```

`break on nodes` searches in `this.expr` (the assert's condition) for instances of `StepIntoable` and contributes their *step into* strategies.

`ExecuteTestExpression` implements `StepIntoable` to allow *step into* the referenced `TestCases`. A minimal implementation puts for each `Testcase` abreakpoint on the first statement in the body:

```
foreach testRef in this.tests {
   break on node: testRef.testcase.body.statements.first;
}
```

D. Watches

Since `ExecuteTestExpression`'s stack frame is not shown in the high-level call stack, none of its watches are mapped. In contrast, stack frames for `TestCases` are visible thus we need to consider its watches. In case of `TestCase`, the `LocalVariableDeclaration _f` has no corresponding representation on the extension-level, therefore we hide it from the watches view (the C watch would be shown otherwise):

```
hide local variable with identifier "_f";
```

VI. Design Decisions

The debugger testing DSL *DeTeL* must allow us to verify at least four aspects: call stack, program state, breakpoints and stepping. To cover these requirements in *DeTeL* we delineate in this section design decisions and their implementation strategy. While we consider some of those design decisions as required (**R**), others are either optional (**O**) or specific to mbeddr (**M**):

R1 Debug state validation: In our test cases we must be able to validate the call stack, and for each of its frames the program state and the location where execution is suspended. The call stack itself is validated by specifying the expected stack frames with their respective names. In terms of program state, we need to verify the names of watches and their respective values, which can either be simple or complex. Finally, a location is specified, where program execution is expected to suspend.

R2 Debug control: In order to test stepping and breakpoints, we require the ability to execute stepping commands (in, over and out) and specify locations where to break.

R3 Language integration: The DSL must integrate with the language extension. This integration is required for specifying in programs under test locations where to break (see R2) and for validating where program execution is suspended (see R1).

O1 Reusability: For writing debugger tests in an efficient way, we expect from *DeTeL* the ability to provide reuse: (1) test data, (2) validation rules and (3) the structure of tests. First covers the ability to have one mbeddr program as test data for multiple test cases. Second refers to single definition and multiple usage of validation rules among different test cases. Finally, third refers to extending test cases and having the possibilty to specialize them.

O2 Extensibility:

This means the language should provide support for contributing new validation rules thus achieving extensibility. Those new rules can be used for testing further debugger functionality not covered by *DeTeL* (e. g., mbeddr's upcoming support for multi-level debugging [6]) or for writing tests more efficiently.

O3 Automated test execution: For fast feedback about newly introduced debugger bugs, we require the ability to integrate our tests into an automatic execution environment (e. g., a build server).

M1 Exchangeable debugger backends:

mbeddr targets the embedded domain where platform vendors require different compilers and debuggers. Hence, we require the ability to run our tests against different debugger backends and on different platforms.

VII. Debugger Testing DSL

DeTeL is integrated in `MPS` and interacts with mbeddr's debugger `API`. While this language is currently tightly coupled to mbeddr, it could in theory interact with a generic debugger `API` and be implemented with a language workbench other than MPS. This section describes the structure of *DeTeL* and the implementation of design decisions discussed in Section VI.

DebuggerTest is the module in which DebuggerTestcases and CallStacks are declared.

A. DebuggerTest

Fig. 4 shows the structure of DebuggerTest: it *contains* ITestContents, currently implemented by DebuggerTestcase and CallStack (described later). By implementing this interface we can also use other concepts inside DebuggerTest (O2). Further, DebuggerTest refers to a Binary (the compiled mbeddr program under test), *imports* of other DebuggerTests (enables reuse O1) and an IDebuggerBackend that specifies the debugger backend to use (M1). Latter is implemented by GdbBackend and allows this way to execute debugger tests with the GNU Debugger (GDB) [7].

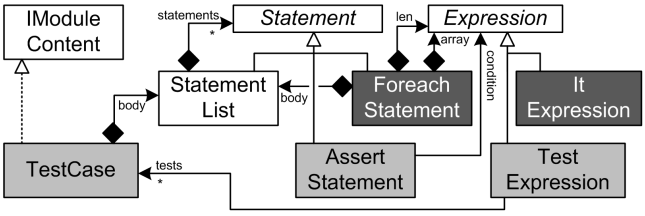

Fig. 4. Structure of DebuggerTests

MPS comes already with the language mps.lang.test for writing type system and editor tests. Its MPS integration allows users to (1) execute tests *automatically* (on the command-line and inside the Integrated Development Environment (IDE)) and (2) get the results of executed tests visualized in a table view. All of that functionality is build for implementations of ITestcase - an interface from mps.lang.test. By implementing this interface in DebuggerTest (our container for DebuggerTestcases), we get the IDE integration and the ability to run our tests *automatically* (O3).

B. CallStack

CallStack implements ITestContent (see Fig. 5) and is used for declaring a stack of IStackFrames. Each frame specifies a *name*, visible *watches* and the *location* where suspended. We have two implementations of IStackFrame: StackFrame and StackFrameExtension (extends a StackFrame). Since a CallStack can extend another CallStack, StackFrameExtension can be used to specialize properties of the extended StackFrame. Additionally, the extending CallStack can declare further StackFrames.

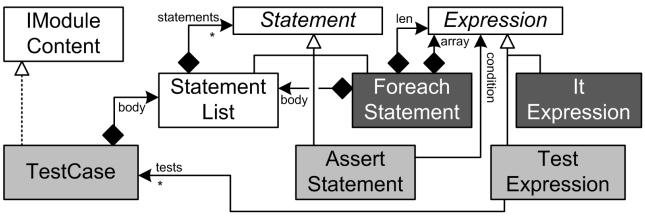

Fig. 5. Structure of CallStacks

IStackFrame has three parts: a *name* (IStackFrameName), a location where suspended (ISuspendLocation) and a list of visible (IWatches).

IStackFrameName has two implementations: AnyStackFrameName does not validate the name, while StackFrameName verifies a specific *name*. Next, ISuspendLocation with its implementations: the location where suspended is not validated with AnyLocation, while ProgramMarkerRef references a ProgramMarker that represents a specific location in an mbeddr program. Instances of ProgramMarker do not influence code generation, they just annotate nodes in the AST. IWatches is implemented by Anywatches and WatchesDeclaration. First does not validate visible watches, while the declaration specifies the *name* of watch and its value with a ValueExpression. This expression again has two implementors: PrimitiveValueExpression for primitive values (e. g., numbers or characters) and ComplexValueExpression for complex values (e. g., arrays or structs).

C. DebuggerTestcase

The structure of DebuggerTestcase is shown in Fig. 6: has a *name*, can *extend* other DebuggerTestcases and can be abstract. Further it holds the following parts: SuspendConfig, SteppingConfig and ValidationConfig. Concrete DebuggerTestcases require at least a SuspendConfig and a ValidationConfig (can be inherited), while an abstract DebuggerTestcase requires none.

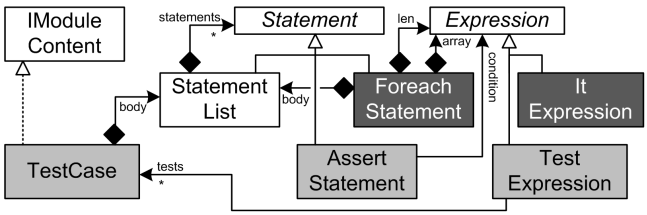

Fig. 6. Structure of DebuggerTestcases

SuspendConfig contains a ProgramMarkerRef that points to *location* inside the mbeddr program under tests. When executing the test, the debugger is suspended at this *location*.

SteppingConfig is optional and contains a list of ISteppingElements that are executed after suspending on *location*. This interface is implemented by ISteppingCommand and SuperConfigElement. Former is again an interface and holds a property *times* that specifies how often the command is executed. Its implementations are StepIntoCommand, StepOverCommand, StepOutCommand – the minimal stepping commands.

ValidationConfig contains a list of IValidationElements. This interface is implemented by the previously described CallStack and CallStackRef, which is used for refering to a CallStack without having the possibility to extend it. Finally, OnPlatform executes contained ValidationConfigs only if tests are executed on the specified Platform (possible platforms are Mac, Unix or Windows).

VIII. Testing the Debugger Extension

The previously described debugger testing DSL is used in this section for testing the debugger extension of unit testing. While those tests do not aim for full test coverage, they concentrate on the essential scenarios to test.

Before writing tests, the program using the unit testing language from Listing 1 is annotated with markers (see listing below). Those markers do not influence the code generation, they are just used by `DebuggerTestcases` to refer to code locations for specifying where to suspend and for verifying where execution suspended.

```
int32 main(int32 argc, string[ ] argv) {
   [return test[forTest];] onReturnInMain
}
testcase forTest {
   [int32 sum = 0;] onSumVarDeclaration
   [assert: sum == 0;] on1stAssertInTestcase
   [int32[ ] nums = {1, 2, 3};] onNumsVarDeclaration
   for(int32_t i=0;i<3;i++) { sum += nums[i]; }
   [assert: sum == 6;] onLastStmntInTestcase
}
```

Next, an empty `DebuggerTest` *UnitTesting* is created (see listing below) that will later contain all `DebuggerTestcases` described in this section. *UnitTesting* tests against the binary *UnitTestingBinary*, which is compiled from Listing 1 and further tells the mbeddr debugger runtime to use the *gdb* debugger backend.

```
DebuggerTest UnitTesting    tests binary: UnitTestingBinary {
                            uses debugger: gdb

}
```

A. Step Into ExecuteTestExpression

For testing *step into* on instances of `ExecuteTestExpression`, in the listing below a `CallStackDeclaration` is created that specifies how the stack must be organized after performing *step into* on *onReturnInMain*. To reuse information and minimize redundance in later `DebuggerTestcases`, two separate `CallStacks` are created: First, *inMain* has a single `StackFrame` that expects (1) execution to be suspended at *onReturnInMain* and (2) two watches to exist – *argc* and *argv*. Second, *inTestcase* extends *inMain* by declaring on top of *main* an addition `StackFrame` *forTest*, which specifies no location, but two watches: *sum* and *nums* (code colored in gray is projected from the extended `CallStack` and not editable).

```
call stack inMain {
   0:main
      location:   onReturnInMain
      watches: {argc, argv}
}

call stack inTestcase extends inMain {
   1:forTest
      location: <any>
      watches: {sum, nums}
   0:main
}
```

After declaring the `CallStacks`, the listing below contains the `DebuggerTestcase` *stepIntoTestcase*, which uses the declaration to verify *step into* for instances of `ExecuteTestExpression`: First, execution is suspended at *onReturnInMain*, next, a single *step into* command is performed before the actual call stack is validated against a custom `CallStack` derived from *inTestcase*. This custom delcaration specializes for the `StackFrame` *forTest* the expected location with *onSumDeclaration*.

```
testcase stepIntoTestcase {
   suspend at:
      onReturnInMain
   then perform:
      step into 1 times
   finally validate:
      call stack stepIntoTestcase extends inTestcase {
         1:forTest
            overwrite location: onSumDeclaration
            watches: {sum, nums}
         0:main
      }
}
```

B. Step into/over AssertStatement

After verifying *step into* for `ExecuteTestExpression`, *step into* and *over* for instances of `AssertStatement` is tested next. Since both stepping commands have the same behavior when performed at *1stAssert*, common information can be extracted into the *abstract* `DebuggerTestcase` *stepOnAssert* shown below: (1) execution is suspended on *1stAssert* and a custom `CallStack` verifies that execution in *forTest* is (2) suspended on *onArrayDecl* and (3) watch *num* has the value zero.

```
abstract testcase stepOnAssert {
   suspend at:
      1stAssert
   finally validate:
      call stack stepOnAssert extends inTestcase {
         1:forTest
            overwrite location:   onArrayDecl
            overwrite watches: {sum=0,nums}
         0:main
      }
}
```

While the first `DebuggerTestcase` *stepIntoAssert* extending *stepOnAssert* performs a *step into*, the other one *stepOverAssert* performs a *step over*:

```
testcase stepIntoAssert extends stepOnAssert {
   then perform:
      step into 1 times
}
testcase stepOverAssert extends stepOnAssert {
   then perform:
      step over 1 times
}
```

C. Step on last Statement in Testcase

The last testing scenario verifies stepping on the last `Statement` (*2ndAssert*) inside a `Testcase` suspends execution on the calling `ExecuteTestExpression` (*onReturnInMain*). The approach for testing this scenario is similar as before: The listing below contains the *abstract* `DebuggerTestcase` *steppingOnLastStmnt*, which suspends exeuction on *2ndAssert* and verifies the actual call stack has the same structure as *inMain*.

```
abstract testcase steppingOnLastStmnt {
   suspend at:
      2ndAssert
   finally validate:
      call stack inMain
}
```

Next, for *step over*, *into* and *out* a separate `DebuggerTestcase` is created, which extend *steppingOnLastStmnt* and specify the additional stepping command:

```
testcase stepOverLastStmnt extends steppingOnLastStmnt {
   then perform:
      step over 1 times
}
testcase stepIntoLastStmnt extends steppingOnLastStmnt {
   then perform:
      step into 1 times
}
testcase stepOutFromLastStmnt extends steppingOnLastStmnt {
   then perform:
      step out 1 times
}
```

IX. Related Work

Wu et al. describe a unit testing framework for DSLs [8] with focus on testing the semantics of the language. However, it is necessary that testing DSLs cover all related aspects of the language e. g., editor (concrete syntax), type system, scoping, transformation rules, and finally the debugger.[1] mbeddr contains tests for the editor, type system, scoping and transformation rules, our work contributes the language for testing the debugger aspect.

Low Level Virtual Machine (LLVM) project [9] comes with a C debugger named Low Level Debugger (LLDB). Test cases for this debugger are written in Python and the unittest framework of Python. While those tests verify the debugger's command line interface and the scripting API, they also test other functionality, such as using the help menu or changing the debugger settings. Further, some of the LLDB tests verify the debugging behavior on different platforms, such as Darwin, Linux or FreeBSD. In contrast, we only concentrate on testing the debugging behavior and we also support writing tests against specific platforms. However, the approach for testing the debugging behavior is derived from the LLDB project: write a program in the source-language (mbeddr), compile it to an executable and debug it through test cases, which verify the debugging behavior.

The GDB provided by the GNU project takes a similar approach as the LLDB: debugger tests cover different aspects of the debugger's functionality and are written in a scripting language [7]. Contrarily, to our approach of testing the debugging behavior for one extensible language, GDB project tests debugging behavior for all of its supported languages, such as C, C++, Java, Ada etc. Further, those tests run on different platforms and different target configurations. Our work supports writing tests against different platforms, but does not allow users to change the target configuration via the DSL.

[1]Specific language workbenches might require testing of additional aspects

X. Summary and Future Work

mbeddr comes with a debugger for extensible languages. To test this debugger, we have introduced in this paper a generic testing DSL. The language is implemented in MPS with a focus on mbeddr, but the underlying approach is applicable for testing any imperative language debugger. Further, we have shown in this paper (1) the implementation of a language extension, (2) how debugging support is build for it and (3) how the debugger is tested with use of our DSL. The language is designed for extensibility, so others can contribute their own context-specific validation rules. In addition, we concentrated on reuse, so test data, test structures and validation rules can be shared among tests.

In the future, we plan to investigate ways for integrating the debugger specification DSL with the DSL for testing the debugger extension. From this integration we expect to (1) gain advances in validating debugger test cases and (2) the possibility to automatically generate test cases from formal debugger specifications (based on work from [10], [11]). In addition, we will continue researching on languages for testing non-functional aspects, such as testing the performance of stepping commands and lifting of program state.

References

[1] M. Voelter, "Language and IDE Development, Modularization and Composition with MPS," in *Generative and Transformational Techniques in Software Engineering*, ser. Lecture Notes in Computer Science, 2011.

[2] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "Mbeddr: An extensible c-based programming language and ide for embedded systems," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12. New York, NY, USA: ACM, 2012, pp. 121–140.

[3] JetBrains, "Meta Programming System," 2015. [Online]. Available: http://www.jetbrains.com/mps

[4] D. Pavletic, M. Voelter, S. A. Raza, B. Kolb, and T. Kehrer, "Extensible debugger framework for extensible languages," in *Reliable Software Technologies - Ada-Europe 2015 - 20th Ada-Europe International Conference on Reliable Software Technologies, Madrid Spain, June 22-26, 2015, Proceedings*, ser. Lecture Notes in Computer Science, J. A. de la Puente and T. Vardanega, Eds., vol. 9111. Springer, 2015, pp. 33–49.

[5] H. Wu, "Grammar-driven Generation of Domain-specific Language Testing Tools," in *20th Annual ACM Special Interest Group on Programming Languages (SIGPLAN) Conference on Object-oriented Programming, Systems, Languages, and Applications*. San Diego, CA, USA: ACM, 2005, pp. 210–211.

[6] D. Pavletic and S. A. Raza, "Multi-Level Debugging for Extensible Languages," *Softwaretechnik-Trends*, vol. 35, no. 1, 2015.

[7] Free Software Foundation, "The GNU Project Debugger," 2015. [Online]. Available: https://www.gnu.org/software/gdb/

[8] H. Wu, J. G. Gray, and M. Mernik, "Unit testing for domain-specific languages," in *Domain-Specific Languages, IFIP TC 2 Working Conference, DSL 2009, Oxford, UK, July 15-17, 2009, Proceedings*, ser. Lecture Notes in Computer Science, W. M. Taha, Ed., vol. 5658. Springer, 2009, pp. 125–147.

[9] LLVM Compiler Infrastructure, "The LLDB Debugger," 2015. [Online]. Available: http://lldb.llvm.org

[10] H. Wu and J. Gray, "Automated generation of testing tools for domain-specific languages." in *ASE*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 2005, pp. 436–439.

[11] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu, "Automatic generation of language-based tools using the LISA system," *Software, IEE Proceedings* -, vol. 152, no. 2, pp. 54–69, 2005.