Testing Debuggers for Extensible Languages

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

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

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

***Abstract*—bla**

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

I. Introduction

Software development faces the challenge that General Purpose Language (GPL) do not provide the appropriate abstractions for many domains in which they are used. Traditionally there are two major approaches to overcome this issue. One is to use frameworks that provide the domain-specific abstractions for the developer. This approach has very limited support for static semantics. Most GPLs do not support modification of constraints or type system. The second is to use external Domain-Specific Languages (DSLs) to give the developer a language for expressing solutions to domain problems. This approach has some other drawbacks. **TODO(***monolithic, all in advanced, central, foo bar***)**. Language engineering with extensible languages provides a solution to theses problems. Instead of having a single monolithic DSL extensible languages enable modular and incremental extension of a host language with domain specific abstractions [1]. Those abstractions are incrementally reduced to lower abstractions until they reach the abstraction level of a GPL.

Debugging programs written with those language extensions at the same abstraction level as they have been defined in the model is crucial to the user. Because of this need, the debugger specification is a important part of the langauge specification. The information how the less domain specific extensions to the executed code has to be in place in the language extension. Description of the debugger behavior is done with a DSL and is part of the language extension. The debugger description heavily relies on the mapping of the language extensions to the executed code.

Because language evolve over time and language extension can be based on other extensions it is essential to constantly test and monitor if the behavior of the debugger matched the expected behavior. In order to do so a language to describe the expected debugger behavior is needed.

II. mbeddr

The mbeddr open-source project[1] focuses on supporting embedded software development. It introduces a set of of modular domain-specific extensions to C and also supports other languages for addressing common problems in software development, e. g., writing documentation with close integration to code or capturing requirements. mbeddr is build using JetBrains Meta Programming System (MPS) language workbench[2]. Fig. 1 shows how mbeddr is organized into layers, with MPSÂăat the bottom and custom language extensions on top. MPS supports the definition, composition and use of general purpose or domain-specific languages. To archive this MPS uses a projectional editor, this means, even if the notation might look textual it is not represented as a sequence of characters, which are transformed into a Abstract Syntax Tree (AST) by parsing. In contrast user actions manipulate the AST directly. The AST is then rendered to the user according to editor specifications/projection rules. These rules are not limited to textual notations, for example can tabular or mathematical notations can be used if appropriated. Since no parsing ambiguities can occur a wide range of languages extension can be supported.

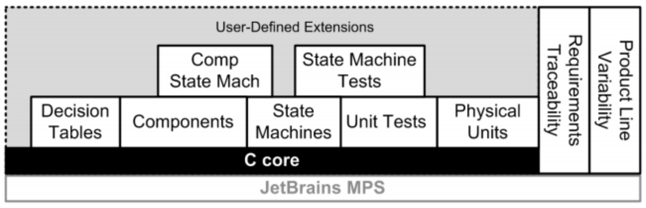

Fig. 1. mbeddr language architecture [2]

A. Languages

mbeddr includes a extensible C99 implementation. In addition to plain C mbeddr also include a set of predefined extension on top of C. These extension include test cases, state machines, components and physical units. In MPS, languages are separated into modular aspects. The major aspects of a language are described below. However, for building debugging support we only care about `Structure`, `Generator` and constraints expressed in `Type System/Contraints`.

Structure: Definition of the AST of the language.

Editor: Projection rules how the AST is presented to the user and how the user interacts with the program.

Type System/Contraints: Static semantics of the language.

Generator: Dynamic semantics of the language, transforms the model into executable code.

[1] http://mbeddr.com

[2] https://www.jetbrains.com/mps/

B. Unit Test Language Extension

In this section we will build a language extension for writing test cases with mbeddr.[3] In the later sections we will show how to build debugging support for this langauge and how to test its debugging support.

Structure:

Fig. 2 shows the language structure, which we discuss now in detail. `AssertStatement` is derived from `Statement` and can therefore be used where `Statements` are epected. It contains an `Expression`, which represents the *condition*. `TestCase` holds a `StatementList`, which contains the `Statements` that make up the test. Further, `TestCase` implements `IModuleContent`, which makes the concept usable on top level of mbeddr `Modules` (a compilation unit). Other subconcepts of this marker interface are `GlobalVariableDeclartion` or `Function`. An `ExecuteTestExpression` is an `Expression` and contains a list of `TestCaseRef`. Those references refer to `TestCase` to be executed.

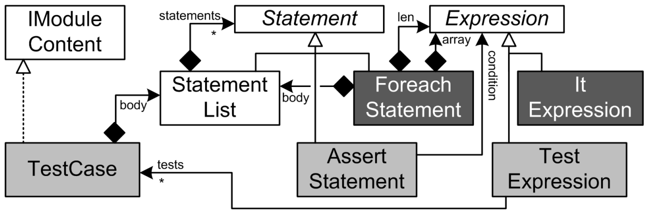

Fig. 2. Language structure

Type System/Constraints:

For implementing our `AssertStatemnt` we require a constraint and a type system rule. First restricts the usages only inside `TestCases`, meaning an `AssertStatemnt` must always live inside (considering the `AST`) a `TestCase`:

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

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

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

Since `ExecuteTestExpression` is an expression, which returns a value (the number of failures), we specify `Int32tType` as its type (see rule below). We will later use the same type inside the code generator.

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

Generator:

Our generator consists of many different transformation rules, which translate code written with our language directly to mbeddr C. Listing 1 shows on the left hand side an example program, written with mbeddr C and our language extension. The right hand side shows the resulting C program. While regular mbeddr code is not colored, the boxes show how AST nodes from the left are translated to C code on the right.

[3]The language design is derivded from mbeddr's `unittest` language.

```
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.

III. mbeddr Debugger

mbeddr comes with a debugger, which allows user to debug their mbeddr code on the abstraction levels of the used languages. Since mbeddr is extensible, the debugger is extensible as well for enabling debugging of new language extensions.

mbeddr programs can be composed of different languages, located on different abstractions levels. Language extensions used inside those programs contribute so called debugger extensions, which provide debugging support for the language itself. This 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. 3) and stepping/breakpoints vice versa.

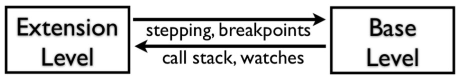

Fig. 3. Flow of debug information between base and extension level [3]

A. Framework

The debugger framework can be separated into two different parts: First, a DSL and a set of interfaces (shown in Fig. 4) for specifying the debugging semantics of language constructs. Second, a runtime for executing those specifications and this way performing the mapping described in the Fig. 3.

You can find further information about the architecture and how it is implemented with MPS in [3]. In this section we only discuss the specification part, which is essential for understanding how to build a debugger extensions for our language example.

Breakpoints `Breakables` are language consturcts on which we can set breakpoints (e. g., `Statements`).

Watchables A `WatchProvider` is either translated to a lower level watchable (e. g., `GlobalVariableDeclaration`) or represents a watchable on the extension-level. Each of them lives inside an `WatchProviderScope` (e. g., `StatementList`), which is nestable.

Stepping Log level breakpoints are used for realizing stepping (approach is based on [4]). Concrete stepping implementations are encapsulated in `SteppingStrategies`. Those strategies are contributed by `Steppables`, on those we can *step over* or *into* (e. g., `ExpressionStatement`). However, *step into* is relevant if they contain `StepIntoables`, something to step into (e. g., `FunctionCall`). Finally, a `Steppable` always lives inside a `SteppableComposite` (e. g., `StatementList`), which is nestable.

Call Stack A `StackFrameContributor` is something that has callable semantics on the extension-level or is translated to a lower-level callable. While the latter category does not contribute any `StackFrames` to the high level call stack, the other category contributes one to many `StackFrames`.

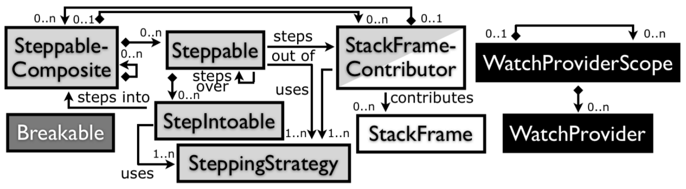

Fig. 4. Meta-model used for specying the debugging semantics of language constructs [3]. Colors indicate the different aspects.

B. Building Debugging Behavior

We describe in this section the implementation of a minimal debugger for our unit test lanuage. The debugger framework comes with a DSL for specifying the debugging behavior. We will use this DSL and the abstractions shown in Fig. 4 for building the debugging support.

Breakpoints We want to be able to set breakpoints on instances of `AssertStatement`. For being able to, we only need to implement the marker interface `IBreakable`. Since this concept is derived from `Statement` which already implements this interface, we can already set breakpoints on instances of `AssertStatement`.

Call Stack For lifting the call stack we need to specify, which concepts have callable semantics or are generated to C functions. `TestCase` has callable semantics and is trasnalted to a C function. In contrast, `ExecuteTestExpression` does not have callable semantics, but is translated to a C function. We therefore implement `IStackFrameContributor` in both concepts.

The listing below shows how we link instances of `ExecuteTestExpression` to low level stack frames. In addition to this, we tell the debugger runtime to hide the stack frame from the high-level call stack.

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

The mapping for `TestCase` is more complex. Here we also start similarly by linking the low level stack to the `TestCase` instance. But, we tell the debugger runtime to show the stack frame in the high-level call stack. Further, we provide the name of the actual `TestCase`, which is shown in the call stack view: Consider our example Listing 1, where we would lift the frame `test_forTest` to `forTest`.

Stepping As described in Section III-A, stepping is implemented by setting low-level breakpoints on the location where execution should suspend after performing a stepping command. `AsserStatement` is a `Statement`, which already provides *step over* behavior. However, we want to be able to *step into* `expr`, in case it contains something to step into. Therefore 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 inside the AST) for instances of `StepIntoable` and contributes their *step into* strategies. Next, in `ExecuteTestExpression`, we overwrite the *step into* behavior in a similar way.

For being able to *step into* `TestCases` we implement `StepIntoable` in `TestCaseRef`. In a minimal implementation we simply put a breakpoint on the first statement in the `Testcase`'s body:

```
break on node: this.testcase.body.statements.first;
```

Watches `ExecuteTestExpression` is translated to C variables, however, those are not considered since the corresponding stack frame is not shown in the call stack. In contrast, `TestCase` generates the `LocalVariableDeclaration` *_fails*, which we hide from the watchables view (the C watchable would be shown otherwise):

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

IV. Design Decisions

Our testing DSL targets debuggers for imperative extensible languages. Those debuggers usually provide at least the following functionality: show the call stack and program state in relation to the languages used, suspend the debugger on breakpoints and offer the possibility to step through the code. Our testing language must allow us to test at least those four aspects. Based on those requirements we have defined some design decision, which we will later implement in our language. While we consider some of those decisions as required, others are either optional or specific to mbeddr.

A. Required

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 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 watchables and their respective values, which can either be simple or complex. Finally, a location is specified, where the debugger is expected to suspend.

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

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

B. Optional

O1 Reusability: For writing debugger tests in an efficient way, we expect from the language the ability to reuse: (1) test data, (2) validation rules and (3) the structure of tests. First covers the ability to have one mbeddr program to debug and write different debugger tests against it. Second refers to reusing a set of validation rules in different test cases. Consider different test cases testing stepping behavior inside the `main` function of a C program. As long as we stay inside the function, the call stack (stack frames, not the location where suspended) would be the same for each test case. Hence, we could write the C program and the call stack validation rule once, and refer to it from each of our test cases. Finally, third, since we could even think about writing an abstract test case, which is specialized by our concrete test cases.

O2 Extensibility: As discussed in R1, we expect from the DSL the ability to validate the program state, location where suspended and the call stack. In addition to this, we require the ability to extend the language for contributing new validation rules. Those new rules could be used for testing further debugger functionality not covered by the testing language (e. g., mbeddr's upcoming support for multi-level debugging [5]) or for writing tests more efficiently.

O3 Automated test execution: For getting fast feedback about bugs introduced into debuggers, we require the ability to integrate our tests into an automatic execution environment. E.g., this environment can be an Integrated Development Environment (IDE) or a command-line tool.

C. mbeddr-specific

M1 Exchangeability of debugger backends: mbeddr targets the embedded domain and C. In this domain, target platforms and vendors require different compilers and debuggers. Hence, we require the ability to run our tests against different debugger backends and on different platforms. To achieve this, we expect from the DSL the ability to specify the debugger backend and the platform against/on which tests should executed.

V. Testing DSL

Our language for testing debuggers is integrated in `MPS` and interacts with mbeddr's debugger `API`. While our language could in theory interacts with a generic debugger `API`, we have not yet considered efforts for investigating that direction.

A. Language Design

MPS comes already with a languae for writing type system and editor tests. Its MPS integration allows users to (1) execute tests automatically from within the IDE and (2) get the results of executed tests visualized in a particular view component. All of that functionality is build for implementors of `ITestCase`. By implementing this interface in our , We get the same functionaltiy for free (O3), by implementing this interface in `DebuggerTest`, which serves as container for `DebuggerTestCases`.

`IDebuggerTestContent` `DebuggerTestReference` `CallStackDeclaration` `WatchablesDeclaration`

`DebuggerTestcase` -> *abstract* -> -> *extends* -> `DebuggerTestcaseReference` -> `SuspensionPointConfiguration` -> `SteppingConfiguration` -> `ValidationConfiguration`

DSL zum beschreiben von Debugger verhalten, die ist erweiterbar integriert Debugger mit C + Extensions Beschreibt das verhalten auf der Abstraction der Erweiterung von C und nicht auf C level

B. Testing Example

Sample Test fÃijr das Sample For Each im TestCase call from test expression CallStack Variablen

```
[int32 sum = 0;] onAssignment
```

VI. Related Work

Wu et al. describe a unit testing framework for DSLs [6]. While they concentrate on testing the semantics of the language, we believe testing DSLs should not only cover the semantics, but also other language aspects: editor (concrete syntax), type system, scoping, transformation rules, and finally the debugger.[4] All of those aspects form part of the language implementation and should therefore be tested as well. While mbeddr comes with tests for the editor, editor, type system, scoping and transformation rules, we contribute in this paper a language for testing the debugger aspect, which is used in mbeddr for testing the debugger.

The Low Level Virtual Machine (LLVM) project [7] comes with a C debugger named Low Level Debugger (LLDB). Tests for this debugger are written in Python and with the unittest framework that comes with 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, however, we also support writing tests against specific platforms. 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.

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

The GNU Debugger (GDB) provided by the GNU project takes a similar approach as the LLDB: debugger tests are cover different aspects of the debugger's functionality and are written in a scripting language [8]. While we concentrate on testing the debugging behavior for one extensible language, the 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. In contrast, we only also support writing tests against different platforms, but do not allow users to change the target configuration via the DSL.

VII. 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 and for mbeddr, but the approach can be applied 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. We have designed the language for extensibility, so others can contribute their own project-specific validation rules. In addition, we concentrated on reuse, so test data, test structures and validation rules and can shared among tests.

In the future, we will investigate ways for integrating the debugger specification DSL with the DSL for debugger testing. 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 [9], [10]). 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. While we will continue our work on testing debuggers, we will also research on methodologies for building debuggers for extensible languages, i. e., on ways to derive debuggers from code generators and transformation rules.

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. [Online]. Available: http://doi.acm.org/10.1145/2384716.2384767

[3] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-19584-1_3

[4] 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.

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

[6] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03034-5

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

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

[9] 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. [Online]. Available: http://dblp.uni-trier.de/db/conf/kbse/ase2005.html#WuG05

[10] 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. [Online]. Available: http://dx.doi.org/10.1049/ip-sen:20041317