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 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.

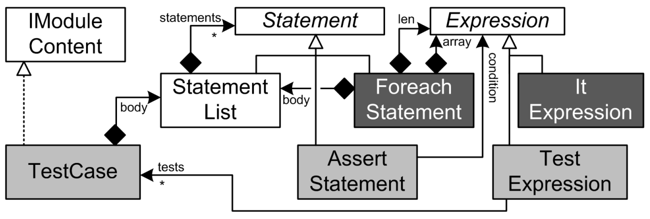

Fig. 1. Language structure

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.

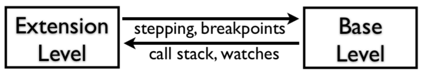

Fig. 2. Flow of debug information between base and extension level [4]

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`.

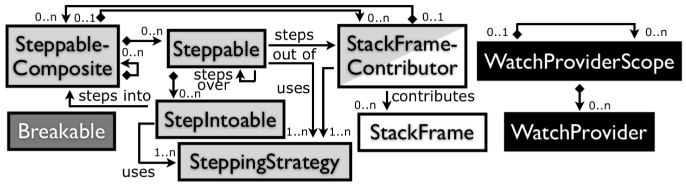

Fig. 3. Meta-model used for specying the debugging semantics of language concepts [4]. Colors indicate the different debugging aspects.

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. Similar steps are taken to impment *step into* for `ExecuteTestExpression`.

`TestCaseRef` implements `StepIntoable` to allow *step into* the referenced `TestCase`. A minimal implementation puts a breakpoint on the first statement in the `Testcase` body:

```
break on node: this.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 `TestCase`s 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 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. The language for testing those debuggers must allow us to verify 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 (**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 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).

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 [6]) 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.

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.

VII. Debugger Testing DSL

Our language for testing debuggers 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, however, we have not yet investigated efforts for going into this direction.

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 IDE) and (2) get the results of executed tests visualized in a table view. All of that functionality is build for implementors 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).

Structure

`DebuggerTest` *contains* `IDebuggerTestContents`, which is implemented by `DebuggerTestCase`. This way we can instanciate `DebuggerTestCases`, but also other implementors of this interface (enables extensibility O2). Further, a `DebuggerTest` specifies an *mbeddr binary* to debug (compiled with debug symbols before executing tets), a *C debugger backend* to use (M1) and *imports* of other `DebuggerTests` (enables reuse O1).

`CallStackDeclaration` is another implementor of `IDebuggerTestContent` and is used for declaring a stack of `IStackFrames`. Each frame specifies a *name*, visible *watchables* and the *location* where suspended. We currently have two implementors of this interface: `StackFrameDeclaration` and `SubStackFrame`, which extends a declaration. Since a `CallStackDeclaration` can extend another declaration `CallStackDeclaration`, we use `SubStackFrame` to specialize properties of the extended `StackFrameDeclaration`. In addition, the extending `CallStackDeclaration` can declare additional `CallStackDeclarations` in the stack.

`IWatchables`, `WatchablesDeclaration`, `AnyWatchables` `ISuspendLocation`, `AnyLocation`, `ProgramMarkerReference` `IStackFrameName`, `AnyStackFrameName`, `SpecificStackFrameName`

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

`DebuggerTestReference`

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

VIII. Testing the Debugger Extension

1. step into TestcaseRef should suspend inside a Testcase

```
DebuggerTest UnitTesting    tests binary: ArrayInitTest
                            uses debugger: gdb
   call stack inMain {
      0:main
         location:   onReturnInMain
         watchables: {argc, argv}
   }

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

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

2. step into/over assert should suspend on array declaration

```
abstract testcase stepOnAssert {
   suspend at:
      1stAssert
   finally validate:
      call stack inTestcase extends inMain {
         1:forTest
            overwrite location:   onArrayDecl
            overwrite watchables: {sum=0,nums}
         0:main
      }
}
testcase stepIntoAssert extends stepOnAssert {
   then perform:
      step into 1 times
}
testcase stepOverAssert extends stepOnAssert {
   then perform:
      step over 1 times
}
```

```
abstract testcase suspendOnArrayDecl {
   finally validate:
      call stack inTestcase {
        1:forTest
          location: <any>
          watchables: <any>
        0 : main
          location: <any>
          watchables: {argc, argv}
      }
}
testcase stepIntoTestcase extends inTestcase {
   suspend at:
      firstArrayVarAssignment
   then perform:
      step into 1 times
}
```

3. step over/out last assert should suspend in main function

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

`[int32 sum = 0;] onAssignment`

IX. Related Work

Wu et al. describe a unit testing framework for DSLs [7]. 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.[1] 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 [8] comes with a C debugger named Low Level Debugger (LLDB). Tests for this debugger are written in Python and with the unit-test 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.

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 [9]. 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.

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 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 [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. 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.

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

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

[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] 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

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

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

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

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