Testing Extensible Language 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*—Extensible languages allow incremental extensions of a host language with domain specific abstractions. Debuggers for such languages must be extensible as well to support debugging of different language extensions at their corresponding abstraction level. As such languages evolve over time, it is essential to constantly verify their debugging behavior. For this purpose, a General Purpose Language (GPL) can be used, however this increases the complexity and decreases the readability of tests. To reduce continuous verification effort, in this paper, we introduce *DeTeL*, an extensible Domain-Specific Language (DSL) for testing extensible language debuggers.**

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

I. Introduction

Software development faces the challenge that 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. The second approach is to use external 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 users can step through constructs provided by the extension and see watch expressions (e. g., variables) related to the extensions.

Because language extensions can be based on other extensions and languages evolve over time, it is essential to constantly test if debugger behavior matches the expected behavior. To test debugging behavior, a GPL can be used, however this raises the same issues discussed above. We therefore propose in this paper *DeTeL* (Debugger Testing Language), an extensible DSL 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. 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 aspects. The major aspects are `Structure`, `Type System`, `Constraints`, `Generator` and `Editor`. However, for building debugging support, the `Editor` aspect is irrelevant.

III. Language Extension for Unit Testing

To give an idea of building language and debugger extensions, we first build *MUnit*, a language for writing unit tests, and a 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.

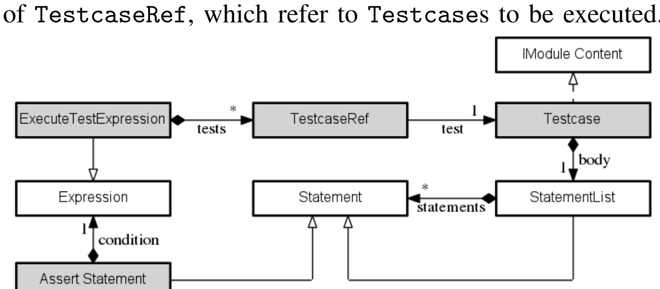

Fig. 1. Language structure

B. Type System and Constraints

`AssertStatement` requires a constraint and a type system rule. It restricts the usages only inside `Testcases`, meaning an `AssertStatement` can only be used in a `Testcase`:

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

It also 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 *MUnit* 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 *MUnit*. The right hand side shows the C program generated from it. While regular mbeddr C code is not colored, the boxes indicate 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 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

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 built with a framework also provided by mbeddr [4]. Those extensions are always language-specific in contrast to domain-specific debuggers (e. g., the moldable debugger [5]), which provide application-specific debug actions and views on the program state. Hence, debugging support is implemented specifically 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.

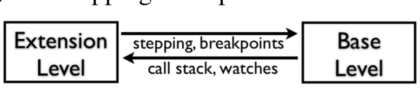

Fig. 2. Flow of debug information between base and extension level [4]

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 section, we provide an overview of the specification part (see Fig. 3) that is required for understanding how the debugger extension for *MUnit* is built. While this paper concentrates on testing debuggers for extensible languages, we have published another paper [4] describing details about the debugger framework and its implementation with MPS.

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., `Argument`) 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 [6]). 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 (e. g., `Functions`). While the latter do not contribute any `StackFrames` to the high level call stack, the former contribute at least one `StackFrame`.

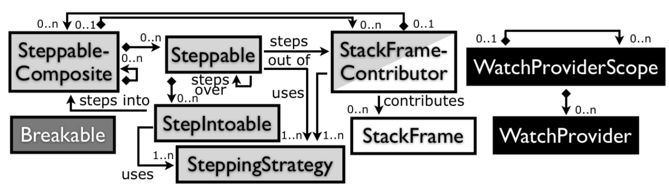

Fig. 3. Meta-model used for specifying the debugging semantics of language concepts [4]. Colors indicate the different debugging aspects

V. Debugger Extension for the MUnit Language

This section describes the implementation of a debugger extension for the *MUnit* language. This extension is defined with the mbeddr debugger specification DSL and the abstractions of the debugging meta-model shown in Fig. 3.

A. Breakpoints

To enable breakpoints on `AssertStatements`, an implementation of the `Breakable` interface is required. `AssertStatement` is derived from `Statement` that already implements this interface, thus breakpoints are already supported.

B. 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, and is therefore not shown (specified in listing below).

The mbeddr debugger framework uses a pessimistic approach for lifting watches: those that should not be shown in the `UI` are marked as *hidden*. Otherwise, the debugger shows the low-level watch (in this case the C local variable _f) with its respective value.

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

C. Stepping

`AssertStatement` 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 *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 a breakpoint in each `Testcase`:

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

D. Call Stack

`Testcase` and `ExecuteTestExpression` are translated to base-level callables and therefore implement `StackFrameContributor`. 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). Further, it hides the frame from the high-level call stack, since `ExecuteTestExpression` has no callable semantics.

```
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:

```
String frameName = "test_" + this.name;
contribute frame mapping for frames.select(name=frameName);
```

Further, we provide the name of the actual `Testcase`, which is represented in the call stack view: Consider Listing 1, where we would show the name `forTest` instead of `test_forTest`.

VI. Requirements

The debugger testing DSL 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 requirements. While we consider some of those requirements as required (**R**), others are either context (**CS**) or mbeddr specific (**MS**).

A. Required

R1 Debug state validation: Changes in generators can modify names of generated procedures or variables and this way, e. g., invalidate program state lifting in the debugger. For being able to identify those problems, we need a mechanism to validate the call stack, and for each of its frames the program state and the location where execution is suspended. For the call stack, a specification of expected stack frames with their respective names is required. In terms of program state, we need to verify the names of watches and their respective values, which can either be simple or complex. Further, a location specifies where program execution is expected to suspend and tests can be written for a specific platform.

R2 Debug control: Similarly as in R1, generator changes also affect the stepping behavior. Consider changing the `FunctionCall` generator to inline the body of called functions instead of calling them. This change would require modifications in the implementation of *step into* as well. For being able to identify those problems, we need the ability to execute stepping commands (in, over and out) and specify locations where to break.

R3 Language integration: The DSL must integrate with language extensions. 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).

B. Context Specific

CS1 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. The first covers the ability to have one mbeddr program as test data for multiple test cases. The second refers to single definition and multiple usage of validation rules among different test cases. Finally, the third refers to extending test cases and having the possibility to specialize them.

CS2 Extensibility: Languages 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 [7]) or for writing tests more efficiently.

CS3 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., an `IDE` or a build server).

C. Mbeddr Specific

MS1 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 open-source and is shipped as part of mbeddr [8]. It is integrated in MPS and interacts with the mbeddr debugger `API`. *DeTeL* is currently tightly coupled to mbeddr, however it could interact with a generic debugger `API` and could be implemented independent of MPS. This section describes the structure of *DeTeL* and the implementation of requirements discussed in Section VI. The language syntax is not documented, but can easily be derived by looking at its editor definitions in MPS.

A. DebuggerTest

Fig. 4 shows the structure of `DebuggerTest`, which is a module that *contains* `IDebuggerTestContents`, currently implemented by `DebuggerTestcase` and `CallStack` (described later). This interface facilitates extensibility inside `DebuggerTest` (CS2). Further, `DebuggerTest` refers to a `Binary`, which is a concept from mbeddr representing the compiled mbeddr program under test (R3), the *imports* of `IDebuggerTestContents` from other `DebuggerTests` (CS1) and an `IDebuggerBackend` that specifies the debugger backend (CS2, MS1). The later is implemented by `GdbBackend` and allows this way to run debugger tests with the GNU Debugger (GDB) [9].

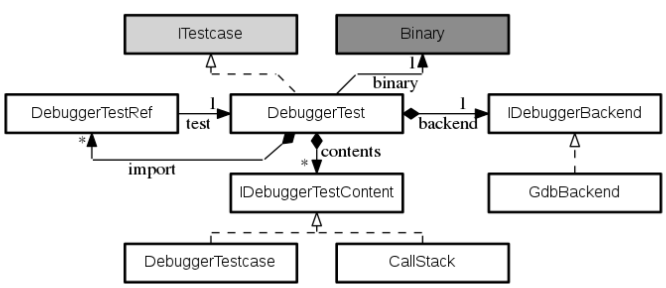

Fig. 4. Structure of `DebuggerTest`

MPS already contains the language `mps.lang.test` for writing type system and editor tests. This allows (1) automatic execution of tests on the command-line and (2) visualization of test results in a table view inside MPS. All of that functionality is built for future implementations of `ITestcase` - an interface from `mps.lang.test`. By implementing this interface in `DebuggerTest` (our container for `DebuggerTestcases`), we benefit from available features (CS3).

B. CallStack

`CallStack` implements `IDebuggerTestContent` (see Fig. 5) and contains `IStackFrames` (CS2, R1), which has two implementations: `StackFrame` and `StackFrameExtension`. An *extend*ing `CallStack` inherits all `StackFrames` from the extended `CallStack` in the form of `StackFrameExtensions`, with the possibility of specializing inherited properties (CS1), and can declare additional `StackFrames`.

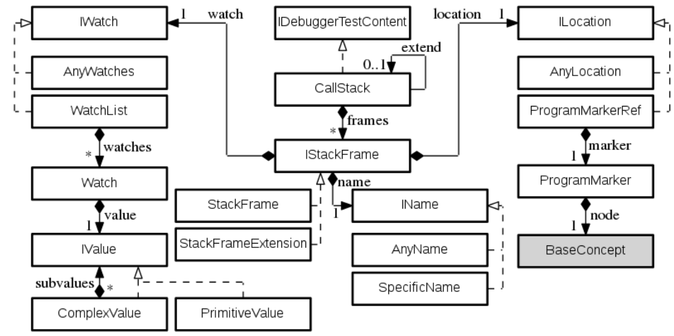

Fig. 5. Structure of `CallStack`

`IStackFrame` has three parts, each with two different implementations: a *name* (`IName`), a location where program execution should suspend (`ILocation`) and visible *watches* (`IWatches`).

`IName` implementations: `SpecificName` verifies the specified *name* matches the actual and `AnyName` ignores it completely. `ILocation` implementations: `AnyLocation` that does not perform any validation and `ProgramMarkerRef` that refers via `ProgramMarker` to a specific location in a program under test (R3). These markers just annotate nodes in the AST and have no influence on code generation. `IWatch` implementations: `AnyWatches` performs no validations and `WatchList` contains a list of `Watches`, each specifies a *name*/*value* (`IValue`) pair. The *value* can be either `PrimitiveValue` (e. g., numbers) or `ComplexValue` (e. g., arrays).

C. DebuggerTestcase

Fig. 6 shows the structure of `DebuggerTestcase`: it can *extend* other `DebuggerTestcases` (CS1), has a *name*, and can be `abstract`. Further it contains the following parts: `SuspendConfig`, `SteppingConfig` and `ValidationConfig`. Concrete `DebuggerTestcases` require a `SuspendConfig` and a `ValidationConfig` (can be inherited), while an `abstract` `DebuggerTestcase` requires none of these.

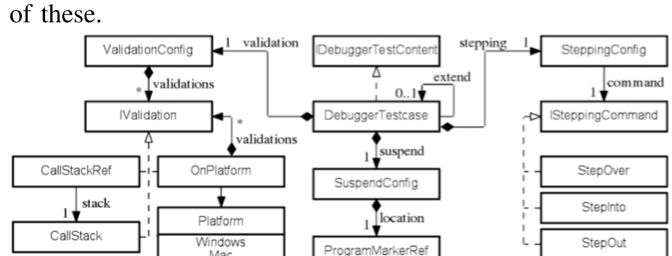

Fig. 6. Structure of `DebuggerTestcase`

`SuspendConfig` contains a `ProgramMarkerRef` that points to the first program *location* where execution suspends (R2).

`SteppingConfig` is optional and contains a list of `ISteppingCommands` (CS2) that are executed after suspending on *location* (R2). This interface is implemented by `StepInto`, `StepOver`, and `StepOut` (each performs the respective command n *times*).

`ValidationConfig` contains a list of `IValidations` (CS2, R1), implemented by `CallStack`, `CallStackRef` and `OnPlatform`. `CallStackRef` refers to a `CallStack` that cannot be modified. Finally, `OnPlatform` specifies a `Platform` (*Mac*, *Unix* or *Windows*) and contains *validations*, which are only executed on the specific platform (R1).

VIII. Writing Debugger Tests

In this section, we describe an application scenario where we apply *DeTeL* to test the debugger extension of `MUnit`.

Before writing debugger tests, we first take the program using `MUnit` from Listing 1 and annotate it in Listing 2 with `ProgramMarkers`. Those markers are later used by `DebuggerTestcases` for specification and verification of code locations where program execution should suspend.

```
int32 main(int32 argc, string[ ] argv) {
   [return test[forTest];] onReturnInMain
}
int32 add(int32 a, int32 b) {
   [return a+b;] inAdd
}
testcase forTest {
   [int32 sum = 0;] onSumDeclaration
   [assert: sum == 0;] firstAssert
   [int32[ ] nums = {1, 2, 3};] onArrayDecl
   for(int32_t i=0;i<3;i++) { sum += nums[i]; }
   [assert: sum == 6;] secondAssert
}
```

Listing 2. Annotated program

Next, in the Listing 3 a stub of `DebuggerTest` *UnitTesting* is created that will later contain all `DebuggerTestcases` described in this section. *UnitTesting* tests against the `Binary` *UnitTestingBinary*, which is compiled from Listing 2. Additionally, it instructs the debugger runtime to execute tests with the `GdbBackend`.

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

Listing 3. `DebuggerTest` stub

A. Step Into ExecuteTestExpression

For testing *step into* on instances of `ExecuteTestExpression`, in the Listing 4, we create a `CallStack` that specifies the stack organization after performing *step into* on *onReturnInMain*. To reuse information and minimize redundancy in subsequent `DebuggerTestcases`, two separate `CallStacks` are created: First, *csInMainFunction* contains a single `StackFrame` that expects (1) program execution to suspend at *onReturnInMain* and (2) two `Watches` (*argc* and *argv*). Second, *csInTestcase* extends *csInMainFunction* by adding an additional `StackFrame` *forTest* on top of the `StackFrameExtension` *main* (colored in gray). This `StackFrame` specifies two `Watches` (*sum* and *nums*) and no specific location (`AnyLocation`).

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

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

Listing 4. `CallStack` declarations

Listing 5 contains the `DebuggerTestcase` *stepIntoTestcase*, which uses the `CallStack` *csInTestcase* to verify *step into* for instances of `ExecuteTestExpression`. As a first step, program execution is suspended at *onReturnInMain*, next, a single `StepInto` is performed before the actual call stack is validated against a custom `CallStack` derived from *csInTestcase*. This custom declaration specializes the `StackFrame` *forTest* i. e., program execution is expected to suspend at *onSumDeclaration*.

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

Listing 5. *Step into* `ExecuteTestExpression`

B. Step into/over AssertStatement

After verifying *step into* for `ExecuteTestExpression` in the previous section, we now test *step into* and *over* for `AssertStatement`. Both stepping commands have the same result when performed at *firstAssert*, hence common test behavior is extracted into the *abstract* `DebuggerTestcase` *stepOnAssert* as shown in Listing 6: (1) program execution suspends at *firstAssert*, (2) a custom `CallStack` verifies program execution suspended in *forTest* on *onArrayDecl* and (3) the `Watch` *num* holds the `PrimitiveValue` zero.

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

Listing 6. Abstract `DebuggerTestcase`

The `DebuggerTestcase` *stepIntoAssert* extending *stepOnAssert* performs a `StepInto` command and *stepOverAssert* performs a `StepOver`:

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

Listing 7. Extending DebuggerTestcases

C. *Step on last Statement in Testcase*

The last testing scenario verifies that stepping on the last `Statement` (*secondAssert*) inside a `Testcase` suspends execution on the `ExecuteTestExpression` (*onReturnInMain*). Again, we create an *abstract* `DebuggerTestcase` *steppingOnLastStmnt* that suspends execution on *secondAssert* and verifies that the actual call stack has the same structure as `CallStack` *csInMainFunction*:

```
abstract testcase steppingOnLastStmnt {
   suspend at:
      secondAssert
   finally validate:
      call stack csInMainFunction
}
```

Listing 8. Assumptions after suspending program execution in *main*

Next, separate `DebuggerTestcases` are created, each for *step over*, *into* and *out*, which extend *steppingOnLastStmnt* and specify only the respective `ISteppingCommand`:

```
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
}
```

Listing 9. Test stepping commands on last `Statement` in `Testcase`

In each `DebuggerTestcase` from the listing above execution suspends on the same `Statement` (*OnReturnInMain*), although different stepping commands are performed. Remember, since *secondAssert* does not contain any children of type `StepIntoable` (e. g., `FunctionCall`), performing a *step into* on the `Statement` has the same effect as a *step over*.

IX. Executing Debugger Tests

Our test cases from the previous section are generated to plain Java code and can be executed in MPS with an action from the context menu. This functionality is obtained by implementing `ITestcase` in `DebuggerTest` (see Section VII-A). By executing this action, test results are visualized in a table view, provided by MPS: for each `DebuggerTestcase`, the result (success or fail) is indicated with a colored bubble and a text field shows the process output.

As indicated by a green bubble on the left side of Fig. 7, all of our previously written `DebuggerTestcases` pass. We show in the next section how language evolution will invalidate the debugger definition and this way cause all of our tests to fail.

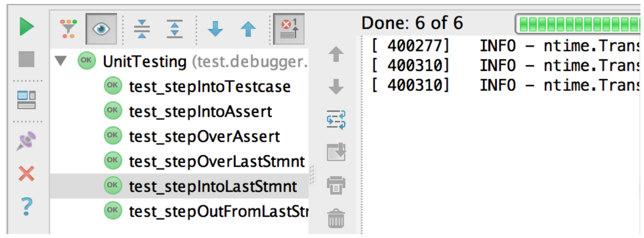

Fig. 7. Successful execution of `DebuggerTestcases` in MPS

X. Language Evolution

The previous sections have shown how to build a language extension for mbeddr in MPS, define a debugger for this extension and use `DeTeL` to test its debugging behavior. This section demonstrates how `DeTeL` is used to identify invalid definitions in debugger extensions after evolving the language.

A. *Evolving MUnit*

In this section we modify the MUnit generator to demonstrate how this affects the debugger. Currently, the generator reduces a `Testcase` to a `Function`: its name is prefixed with *test_*, followed by the `Testcase` name (see Listing 1). We now change this generator, so the `Function` name is prefixed with *testcase_*, instead of *test_*. The listing below shows how our example program from Listing 1 is now generated to C.

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

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

```
int32_t testcase_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 10. C code that has been generated with the modified `Testcase` generator for the example program from Listing 1

Because of our generator modification, `Testcases` are now generated to `Functions` with a different identifier as before. However, we have not updated the debugger extension, therefore, the call stack construction for all `Testcases` fails and this way all of our `DebuggerTests` fail as well (see Fig. 8). Although those debugger tests fail, they are still valid, since they are written on the abstraction level of the languages, not the generator. The next section shows how we update the debugger extension to solve the call stack construction.

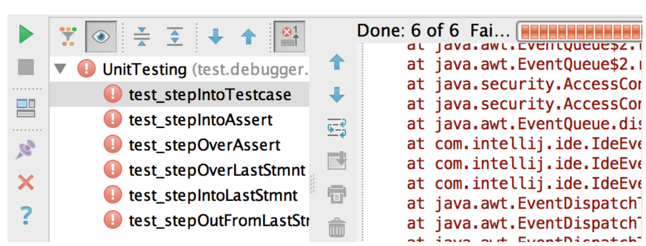

Fig. 8. Failing `DebuggerTestcases` after modifying the generator

B. Updating the Debugger Extension

The *MUnit* debugger extension tries to lift for each `Testcase` a stack frame whose name is prefixed with *test_*, followed by the name of the respective `Testcase` (see Section V-D). However, due to our generator modification, this frame is not present and therefore the whole call stack construction fails with an error. To solve this problem, we update the name used for matching the stack frame name:

```
String frameName = "testcase_" + this.name;
contribute frame mapping for frames.select(name=frameName);
```

Other aspects, such as stepping, breakpoints or watches are not affected by the generator modification and hence do not need to be changed. Therefore, after fixing the call stack lifting for `Testcase` our debugger tests pass again.

XI. Related Work

Wu et al. describe a unit testing framework for DSLs [10] with focus on testing the semantics of the language. However, from our perspective, it is necessary to have testing DSLs for all aspects of the language definition, 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.

The Low Level Virtual Machine (LLVM) project [11] comes with a C debugger named Low Level Debugger (LLDB). Test cases for this debugger are written in Python and the unit test framework of Python. While those tests verify the command line interface and the scripting Application Programming Interface (API) of the debugger, 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 or Linux. In contrast, we only concentrate on testing the debugging behavior, but also support writing tests for specific platforms. Our 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 debugger takes a similar approach as the LLDB: tests cover different aspects of the debugger functionality and are written in a scripting language [9]. Contrarily, to our approach of testing debugging 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 target configurations. Our work supports writing tests against different platforms, but does not allow users to change the target configuration via the DSL.

XII. Summary and Future Work

The mbeddr extensible language comes with an extensible debugger. To test this debugger, we have introduced in this paper a generic and extensible testing DSL. The language is implemented in MPS with 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 [12], [13] and [14]). 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.

[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.

[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] A. Chis, T. Gîrba, and O. Nierstrasz, “The moldable debugger: A framework for developing domain-specific debuggers,” in *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, 2014, pp. 102–121.

[6] 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.

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

[8] B. Kolb, M. Voelter, D. Ratiu, D. Pavletic, Z. Molotnikov, K. Dummann, N. Stotz, S. Lisson, S. Eberle, T. Szabo, A. Shatalin, K. Miyamoto, and S. Kaufmann, “mbeddr.core - An extensible C,” https://github.com/mbeddr/mbeddr.core, GitHub repository, 2015.

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

[10] 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.

[11] LLVM Compiler Infrastructure, “The LLDB Debugger,” 2015. [Online]. Available: http://lldb.llvm.org

[12] 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.

[13] 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.

[14] H. Wu, J. Gray, and M. Mernik, “Grammar-driven generation of domain-specific language debuggers.” *Software: Practice and Experience*, vol. 38, no. 10, pp. 1073–1103, 2008.