

# Testing Extensible Language Debuggers

1st International Workshop on Executable Modeling

Domenik Pavletic, Syed Aoun Raza, Kolja Dummann and Kim Haßlbauer

© Pavletic et. al. 2015

# Agenda

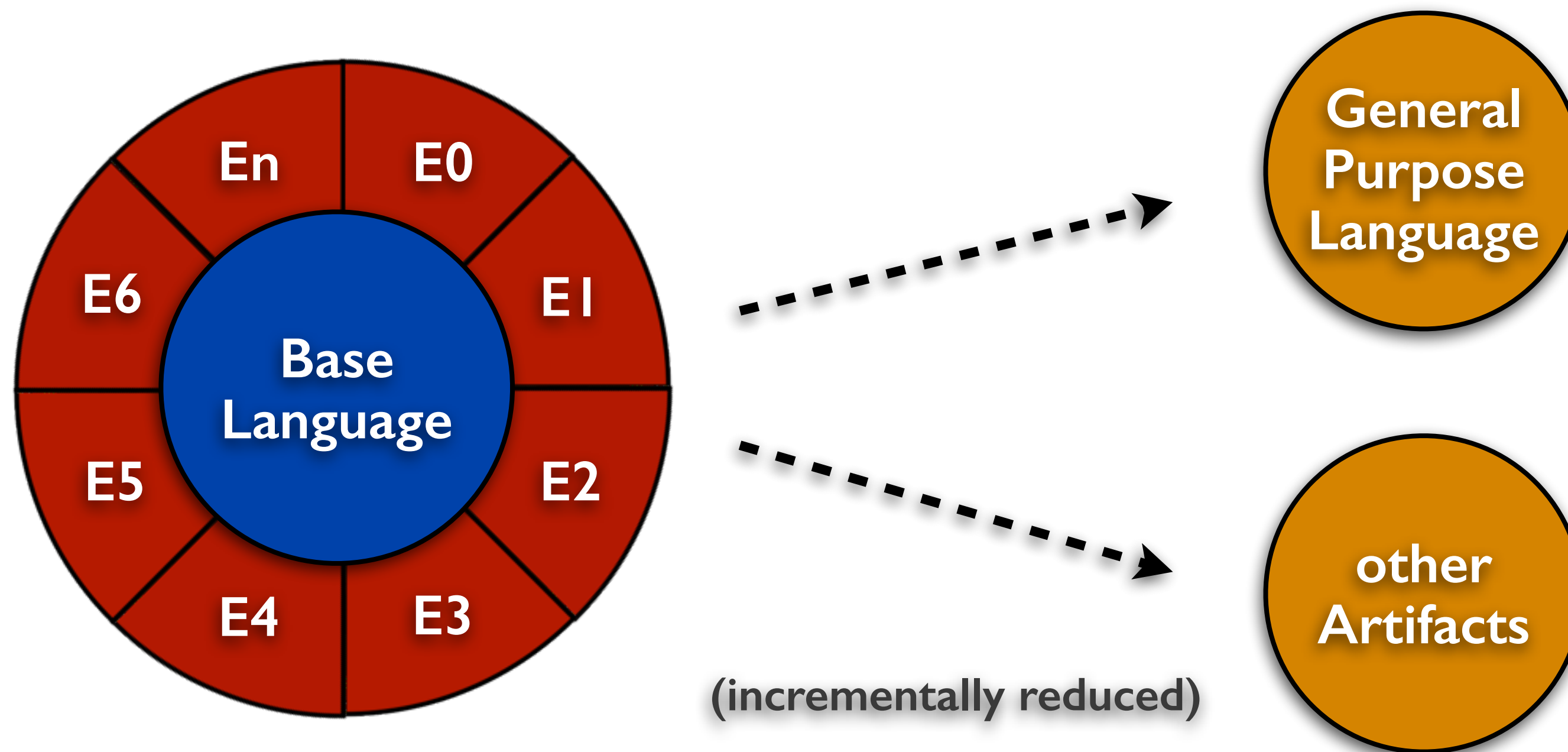
- \* Extensible Languages and Debuggers
- \* Requirements on the Testing DSL
- \* Testing Debugger Extensions

# **Extensible Languages**

*the context*

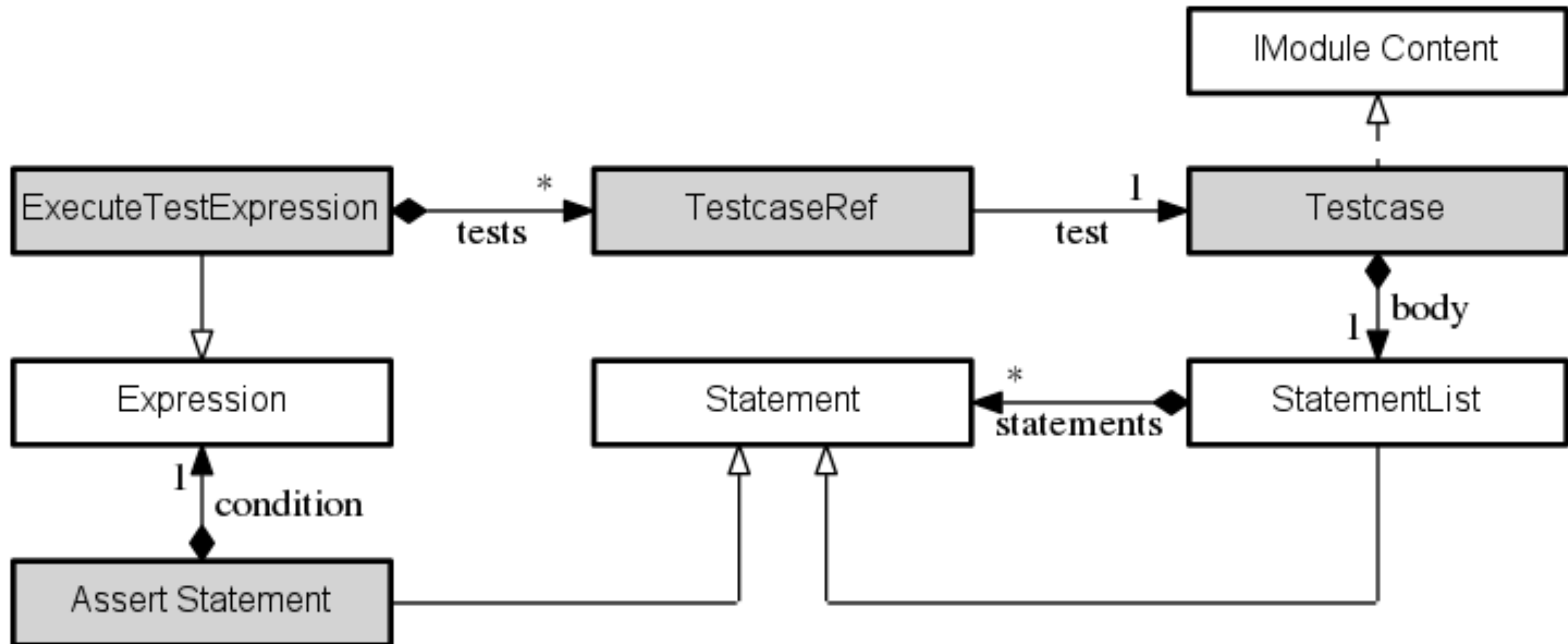
# Extensible Languages

*enable integration of new language extensions*



# Example Language Extension

*we extend mbeddr with unit testing*



# Code Generator

*generating an example program*

Extension-  
Level

```
1 int32 main(int32 argc,  
2   string[] argv) {  
3   return test[ forTest ];  
4 }  
5  
6  
7  
8  
9  
10  
11  
12 testcase forTest {  
13  
14   int32 sum = 0;  
15   assert: sum == 0 ;  
16   int32[] nums = {1, 2, 3};  
17   for(int32_t i=0;i<3;i++){  
18     sum += nums[i];  
19   }  
20   assert: sum == 6 ;  
21  
22 }
```

```
1 int32_t main(int32_t argc,  
2   char *(argv[])) {  
3   return blockexpr_2();  
4 }  
5  
6 int32_t blockexpr_2(void) {  
7   int32_t _f = 0;  
8   _f += test_forTest();  
9   return _f;  
10 }  
11  
12 int32_t test_forTest() {  
13   int32_t _f = 0;  
14   int32_t sum = 0;  
15   if(!( sum == 0 )) { _f++; }  
16   int32_t[] nums = {1, 2, 3};  
17   for(int32_t i=0;i<3;i++){  
18     sum += nums[i];  
19   }  
20   if(!( sum == 6 )) { _f++; }  
21   return _f;  
22 }
```

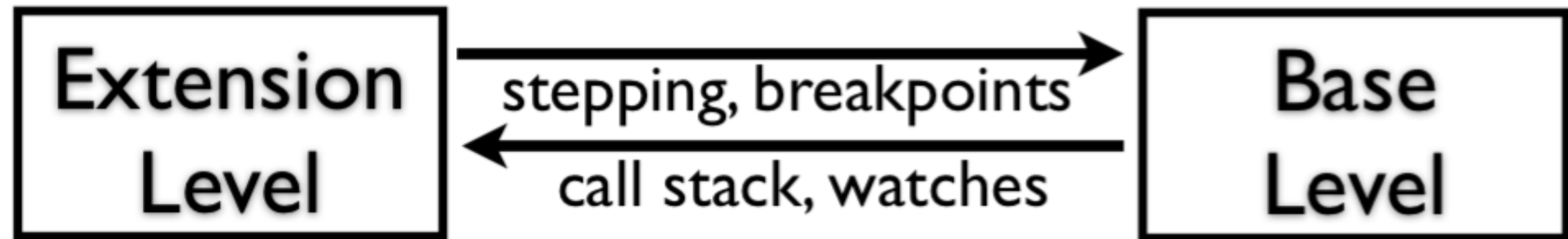
Base-Level

# Extensible Language Debuggers

*how we build them for mbeddr*

# Debugger Extensions in mbeddr

*debug information is lifted/mapped*



- Languages contribute Debugger Extensions
- mbeddr Debugger Framework
  - meta-model, DSL and debugger runtime
- Base-Level Debugger (GDB) used



# mbeddr Debugger

## *debugging on extension-level*

The screenshot displays the mbeddr debugger interface. At the top, a tab labeled 'Main' is visible. The main area shows the source code of a C program. The code is as follows:

```
exported int32 main(int32 argc, string[] argv) {  
    return test[forTest; (unittest)];  
}  
main (function)  
  
testcase forTest {  
    int32 sum = 0;  
    assert(0) sum == 0;  
    int32[] nums = {1, 2, 3};  
    for (int32 i = 0; i < 3; i++) {  
        sum += nums[i];  
    } for  
    assert(1) sum == 6;  
} forTest(test case)
```

A red dot on the left margin indicates a breakpoint is set at the line `assert(0) sum == 0;`, which is highlighted in blue. Below the code editor, the 'Debug' window shows the current session for '(default) SMHelloWorld'. The 'Debugger' tab is active, and the 'Console' is empty. The 'Frames' pane shows the current stack frame: `forTest():22 Main(test.ex.ext.statemachine.helloWorld@tests)`. The 'Thread' pane shows the thread name: `main():12 Main(test.ex.ext.statemachine.helloWorld@tests)`. The 'Variables' pane shows the current state of variables: `nums = [3]` and `sum = 0`.

# Building Debuggers in mbeddr

## *lifting the call stack*

lift stack frame name

```
12 testcase forTest {
13
14     int32 sum = 0;
15     assert: sum == 0 ;
16     int32[] nums = {1, 2, 3};
17     for(int32_t i=0;i<3;i++){
18         sum += nums[i];
19     }
20     assert: sum == 6 ;
21
22 }
```

```
12 int32_t test_forTest() {
13     int32_t _f = 0;
14     int32_t sum = 0;
15     if(!( sum == 0 )) { _f++; }
16     int32_t[] nums = {1, 2, 3};
17     for(int32_t i=0;i<3;i++){
18         sum += nums[i];
19     }
20     if(!( sum == 6 )) { _f++; }
21     return _f;
22 }
```

```
public void contributeFrameMappings(list<IMWMappingStackFrame> frames) {
    string name = "test_" + this.name;
    contribute frame mapping for frames.selectFrame(name=name) { << ... >> };
}
```

# Building Debuggers in mbeddr

## *mapping stepping behavior*

```
1 int32 main(int32 argc,  
2   string[] argv) {  
3   return test[ forTest ] ;  
4 }  
5  
6  
7  
8  
9  
10  
11  
12 testcase forTest {  
13  
14   int32 sum = 0 ;  
15   assert: sum == 0 ;  
16   int32[] nums = {1, 2, 3};  
17   for(int32 t i=0:i<3:i++){
```

step-into

```
1 int32_t main(int32_t argc,  
2   char *(argv[1])) {  
3   return blockexpr_2() ;  
4 }  
5  
6 int32_t blockexpr_2(void) {  
7   int32_t _f = 0 ;  
8   _f += test_forTest();  
9   return _f ;  
10 }  
11  
12 int32_t test_forTest() {  
13   int32_t _f = 0 ;  
14   int32_t sum = 0 ;  
15   if(!( sum == 0 )) { _f++; }  
16   int32_t[] nums = {1, 2, 3};  
17   for(int32 t i=0:i<3:i++){
```

step-into

```
public void contributeStepIntoStrategies(list<IDebugStrategy> resultStrategies)  
  overrides ISteppable.contributeStepIntoStrategies {  
  this.tests.forEach({~testRef => break on node: testRef.test.body.statements.first; });  
}
```

**Debugger Extensions are  
written manually and  
therefore error-prone**

*testing is required ...*

# Using a DSL to test Debugger Extension

*the important requirements ..*

# Requirements

*the important ones ...*

## 1. Test debugging behavior

- \* Program state, call stack, breakpoints & stepping

## 2. Reuse information

- \* Test data, validation rules & structure of tests

## 3. Execute tests automatically

- \* IDE & build server

# Using the Testing DSL

*we test step-into for Testcases ...*

# Testing Debugger Extensions

*1. annotate the program under test*

```
0 Main constraints  
model test.ex.ext.statemachine.helloWorld imports nothing
```

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

```
testcase forTest {  
  [int32 sum = 0; ]onSumDecl  
  [assert(0) sum == 0; ]firstAssert  
  [int32[] nums = {1, 2, 3}; ]onArrayDecl  
  for (int32 i = 0; i < 3; i++ ) {  
    sum += nums[i];  
  } for  
  [assert(1) sum == 6; ]secondAssert  
} forTest(test case)
```



# Testing Debugger Extensions

## 2. *specify an expected CallStack: after step-into*

```
int32 main(int32 argc, string[] argv) {  
    [return test [forTest;] ]onReturnInMain  
} main (function)  
testcase forTest {  
    [int32 sum = 0; ]onSumDecl  
    [assert(0) sum == 0; ]firstAssert  
    [int32[] nums = {1, 2, 3}; ]onArrayDecl  
    for (int32 i = 0; i < 3; i++ ) {  
        sum += nums[i];  
    } for  
    [assert(1) sum == 6; ]secondAssert  
} forTest(test case)
```

```
call stack csInTest {
```

```
💡 | 1 forTest  
   | 0 main  
}
```

---

**name:**

forTest

**location:**

**<any location>**

**watches:**

**watches inForTest {**

sum

nums

**}**

# Testing Debugger Extensions

## 3. write the *DebuggerTestcase*

```
int32 main(int32 argc, string[] argv) {
  [return test [forTest;]; ]onReturnInMain
} main (function)
testcase forTest {
  [int32 sum = 0; ]onSumDecl
  [assert(0) sum == 0; ]firstAssert
  [int32[] nums = {1, 2, 3}; ]onArrayDecl
  for (int32 i = 0; i < 3; i++ ) {
    sum += nums[i];
  } for
  [assert(1) sum == 6; ]secondAssert
} forTest(test case)
```

```
test case stepIntoTestcase {
  suspend at:
    onReturnInMain
  then perform:
    step into 1 times
  finally validate:
    call stack OnSumDecl extends csInTest {
      1 forTest
      0 main
    }
}
inherited name:
  forTest
overwrite inherited location:
  onSumDecl
inherited watches:
  watches inForTest {
    sum
    nums
  }
```

# Testing Debugger Extensions

## 4. execute the test

The screenshot shows a debugger window titled "UnitTesting" with a yellow header bar. The header bar contains the text: "Debugger Test UnitTesting tests binary: UnitTestingBinary uses debugger: gdb imports: <none>". Below the header bar, there is a code block with the following content:

```
call stack csInTestcase {  
  1 forTest  
  0 main  
}  
test case stepIntoTestcase {  
  suspend at:  
    onReturnInMain  
  then perform:  
    step into 1 times  
  finally validate:  
    call stack csOnSumDeclInTestcase extends csInTestcase {  
      1 forTest  
      0 main  
    }  
}
```

At the bottom of the window, there is a "Run" section with a toolbar and a table of test results. The toolbar includes icons for play, eye, step over, step into, step out, and a red X icon. The table shows the following results:

Test	Time...	Usage...	Usag...	Usage...	Results
Total:	3,087 s	-238...	9288...	9049...	...
test.deb	3,087 s	-238...	9288...	9049...	...
test_s	3,087 s	-238...	928808 Kb	49...	Pass...

# Language Evolution

*your debugger extension will break ...*

# Evolving the Testcase Generator

*prefix of generated C Function is changed*

Extension-  
Level

```
1 int32 main(int32 argc,  
2   string[] argv) {  
3   return test[ forTest ];  
4 }  
5  
6  
7  
8  
9  
10  
11  
12 testcase forTest {  
13  
14   int32 sum = 0;  
15   assert: sum == 0 ;  
16   int32[] nums = {1, 2, 3};
```

Base-Level

```
1 int32_t main(int32_t argc,  
2   char *(argv[])) {  
3   return blockexpr_2();  
4 }  
5  
6 int32_t blockexpr_2(void) {  
7   int32_t _f = 0;  
8   _f += testcase_forTest();  
9   return _f;  
10 }  
11  
12 int32_t testcase_forTest() {  
13   int32_t _f = 0;  
14   int32_t sum = 0;  
15   if(!( sum == 0 )) { _f++; }  
16   int32_t[] nums = {1, 2, 3};
```

# Evolving the Testcase Generator

*debugger test fails*

```
UnitTesting x
Debugger Test UnitTesting      tests binary: UnitTestingBinary
                               uses debugger: gdb
                               imports: <none>

call stack csInTestcase {
  1 forTest
  0 main
}
test case stepIntoTestcase {
  suspend at:
    onReturnInMain
  then perform:
    step into 1 times
  finally validate:

```

```
Run UnitTesting
Done: 1 of 1 Failed: 1
at java.security.AccessController.doPrivileged(Native Method)
at java.security.AccessControlContext$1.doIntersectionPrivilege(AccessControlContext.java:85)
at java.awt.EventQueue.dispatchEvent(EventQueue.java:685)
at com.intellij.ide.IdeEventQueue.defaultDispatchEvent(IdeEventQueue.java:3)
at com.intellij.ide.IdeEventQueue._dispatchEvent(IdeEventQueue.java:3)
at com.intellij.ide.IdeEventQueue.dispatchEvent(IdeEventQueue.java:3)
at java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.java:3)
at java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.java:3)
at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:3)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:3)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:3)
```

→ Tests are fine, call stack construction is invalid!

# Evolving the Testcase Generator

*call stack construction fails*

Debugger Extension must use different name:

```
public void contributeFrameMappings(list<IMWMappingStackFrame> frames) {  
    string name = "testcase_" + this.name;  
    contribute frame mapping for frames.selectFrame(name=name) { << ... >> };  
}
```

**return „any Questions?“;**