*14*

## Testing DSLs

*All the aspects of DSL implementation we have discussed so far need to be tested to keep them stable. In this chapter we address testing of the language syntax, the constraints and the semantics, as well as some of the editor services, based on examples with Xtext, MPS and Spoofax. We conclude the chapter with a brief look at "testing" a language for appropriateness relative to the domain.*

DSL testing is a multi-faceted problem, since it needs to address all the aspects of the DSL implementation we have discussed so far. In particular, this includes the syntax, the constraints and type system, as well as the execution semantics (i.e. transformations or interpreters). Here are some examples:

* Can the syntax cover all required sentences? Is the concrete syntax "correct"?
* Do the scopes work correctly?
* Do the constraints work? Are all "wrong" programs actually detected, and is the right error message attached to the right program element?
* are the semantics correct? Do transformations, generators and interpreters work correctly?
* Can all programs relevant to the users actually be expressed? Does the language cover the complete domain?

### 14.1 Syntax Testing

Testing the syntax is simple in principle. Developers simply try to write a large set of relevant programs and see if they can be expressed with the language. If not, the concrete syntax is incomplete. We may also want to try to write "wrong" programs and check that the errors are detected, and that meaningful error messages are reported.

|  |  |
| --- | --- |
| *An Example with Xtext* The following piece of code is the fundamental code that needs to be written in Xtext to test a |  |
| DSL program using the Xtext testing utilities1. It is a JUnit 4 |  |
| test case (with special support for the Xtext infrastructure2), so it can be run as part of Eclipse’s JUnit integration. |  |

|  |
| --- |
| @RunWith(XtextRunner.**class**)  @InjectWith(CoolingLanguageInjectorProvider.**class**) **public class** InterpreterTests **extends** XtextTest {  @Test  **public void** testET0() **throws** Exception { testFileNoSerializer("interpreter/engine0.cool", " tests.appl", "stdparams.cool" ); }  } |

|  |  |
| --- | --- |
| The single test method loads the **interpreter/engine0.cool** program, as well as two more files which contain elements referenced from **engine0.cool**. The **testFileNoSerializer** method loads the file, parses it and checks constraints. If either |  |
| parsing or constraint checking fails, the test fails3. |  |
| On a more fine-grained level it is often useful to test partial sentences instead of complete sentences or programs. The following piece of Xtext example code tests the **CustomState** parser rule: |  |

|  |
| --- |
| @Test  **public void** testStateParserRule() **throws** Exception { testParserRule("state s:",  "CustomState" );  testParserRule("state s: entry { do fach1->anOperation }",  "CustomState" );  testParserRule("state s: entry { do fach1->anOperation }", "State" ); } |

The first line asserts that the string **state s:** can be parsed with the **CustomState** parser rule. The second line passes in a more complex state, one with a command in an entry action. Line three tries the same text with the **State** rule, which itself calls the **CustomState**[[1]](#footnote-1).

*An Example with Spoofax* Spoofax supports writing tests for language definitions using a testing language. Consider the following test suite:

|  |
| --- |
| **module** example **language** MoblEntities  **test** empty **module** [[**module** foo]] **parse succeeds test** missing layout (**module** name) [[modulefoo]] **parse fails** |

The first two lines specify the name of the test suite and the language under test. The remaining lines specify positive and negative test cases concerning the language’s syntax. Each test case consists of a name, the to-be-tested code fragment in double square brackets, and a specification that determines what kind of test should be performed (**parsing**) and what the expected outcome is (**succeeds** or **fails**). We can also specify the expected abstract syntax based on the ATerm textual notation:

**test** empty **module** (AST) [[**module** foo]] **parse to** Module("foo", [])

Instead of specifying a complete abstract syntax tree, we can only specify the interesting parts in a pattern. For example, if we only want to verify that the definition list of an empty module is indeed empty, we can use **\_** as a wildcard for the module name:

**test** empty **module** (AST) [[**module** foo]] **parse to** Module(\_, [])

Abstract syntax patterns are particularly useful for testing operator precedence and associativity:

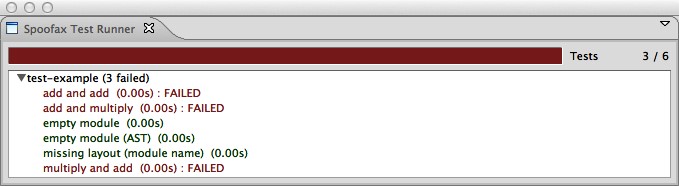
**test** multiply and add [[1 + 2 \* 3]] **parse to** Add(\_, Mul(\_, \_)) **test** add and multiply [[1 \* 2 + 3]] **parse to** Add(Mul(\_, \_), \_) **test** add and add [[1 + 2 + 3]] **parse to** Add(Add(\_, \_), \_)

Alternatively, we can specify an equivalent concrete syntax fragment instead of an abstract syntax pattern:

**test** multiply and add [[1 + 2 \* 3]] **parse to** [[1 + (2 \* 3)]] **test** add and multiply [[1 \* 2 + 3]] **parse to** [[(1 \* 2) + 3]] **test** add and add [[1 + 2 + 3]] **parse to** [[(1 + 2) + 3]]

A test suite can be run from the *Transform* menu. This will open the *Spoofax Test Runner View*, which provides information about failing and succeeding test cases in a test suite. Fig. 14.1 shows an example. Additionally, we can also get instant feedback while editing a test suite. Tests can also be evaluated outside the IDE, for example as part of a continuous integration setup.

*Syntax Testing with MPS* Syntax testing in the strict sense is not useful or necessary with MPS, since it is not possible to “write text that does not parse”. Invalid programs cannot even



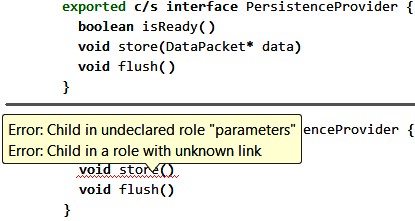
be entered. However, it is useful to write a set of programs which the language developer considers relevant. While it is not possible to write syntactically invalid programs, the following scenario is possible (and useful to test): a user writes a program with the language in version 1. The language evolves to version 2, making that program invalid. In this case, the program contains unbound language concepts or “holes”. By running the model checker (interactively or via **ant**), such problems can be detected. Fig. 14.2 shows an example.

Figure 14.2: *Top:* an interface expressed

### 14.2 Constraints Testing

Testing of constraints is essential, especially for languages with complex constraints, such as those implied by type systems. The goal of constraints testing is to ensure that the correct error messages are annotated to the correct program elements, if those program elements have a constraint or type error.

*An Example with Xtext* A special API is necessary to be able to verify that a program which makes a particular constraint fail actually annotates the corresponding error message to the respective program element. This way, tests can then be written which assert that a given program has a specific set of error annotations.

The unit testing utilities mentioned above also support testing constraints. The utilities come with an internal Java DSL that supports checking for the presence of error annotations after parsing and constraint-checking a model file.

|  |
| --- |
| @Test  **public void** testTypesOfParams() **throws** Exception { testFileNoSerializer("typesystem/tst1.cool", "tests.appl", "stdparams. cool");  assertConstraints( issues.sizeIs(3) ); // 1 assertConstraints( issues.forElement(Variable.**class**, "v1"). // 2 theOneAndOnlyContains("incompatible type") ); // 2 assertConstraints( issues.under(Variable.**class**, "w1"). // 3 errorsOnly().sizeIs(2).oneOfThemContains("incompatible type") ); // 3  } |

|  |  |  |  |
| --- | --- | --- | --- |
| We first load the model file that contains constraint errors (in this case, type system errors). Then we assert the total number | | |  |
| of errors in the file to be three (line 1)5. Next, in line 2, we | | |  |
| check that the instance of **Variable** named **v1** has exactly one error annotation, and that it has the text "incompatible type" in the error message. Finally, in line 3 we assert that there are exactly two errors anywhere under (i.e. in the subtree below) a **Variable** named **w1**, and one of these contains "incompatible type" in the error message. Using the fluent API style shown by these examples, it is easy to express errors and their locations in the program. If a test fails, a meaningful error message is output that supports localizing (potential) problems in the test. The following is the error reported if no error message is found that contains the substring *incompatible type*: | | |  |
| junit.framework.AssertionFailedError: <no id> failed  - failed oneOfThemContains: none of the issues contains substring ’incompatible type’  at junit.framework.Assert.fail(Assert.java:47) at junit.framework.Assert.assertTrue(Assert.java:20) ... |

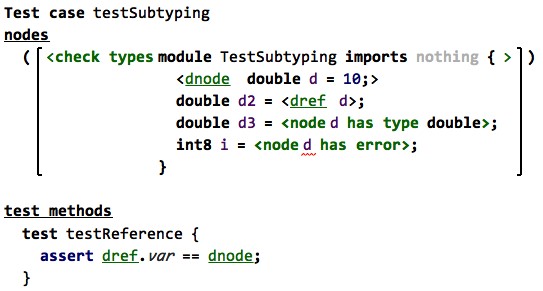
|  |
| --- |
| junit.framework.AssertionFailedError: <no id> failed   * no elements of type com.bsh.pk.cooling.coolingLanguage.Variable named ’v1’ found * failed oneOfThemContains: none of the issues contains substring ’incompatible type’   at junit.framework.Assert.fail(Assert.java:47) ... |

|  |  |
| --- | --- |
| Scopes can be tested in the same way: we can write example programs where references point to valid targets (i.e. those in scope) and invalid targets (i.e. not in scope). Valid references |  |
| may not have errors, invalid references must have errors6. |  |
| *An Example with MPS* MPS comes with the **NodesTestCase** for testing constraints and type system rules (Fig. 14.3). It supports special annotations to express assertions on types and errors, directly in the program. For example, the third line of the nodes section in Fig. 14.3 reads **var double d3 = d** without annotations. This is a valid variable declaration in mbeddr C. After this has been written down, annotations can be added. They are rendered in green (gray in the hardcopy version of the book). Line three asserts that the type of the variable **d** is **double**, i.e. it tests that variable references assume the type of the referenced variable. In line four we assign a **double** to an |  |

A test may also fail earlier in the chain of filter expressions if, for example, there is no **Variable** named **v1** in the program.

More output is provided in this case:

**int**, which is illegal according to the typing rules. The error is detected, hence the red underline. We use another annotation to assert the presence of the error.



In addition to using these annotations to check types and typ-

ing errors, developers can also write more detailed test cases about the structure or the types of programs. In the example we assert that the **var** reference of the node referred to as **dref** points to the node labeled as **dnode**. Note how labels (green, underlined) are used to add names to program elements so they can be referred to from test expressions. This approach can be used to test scopes. If two variables with the same name are defined (e.g., because one of them is defined in an outer block, and we assume that the inner variable shadows the outer variable of the same name), we can use this mechanism to check that a reference actually points to the inner variable.

Fig. 14.4 shows an example.

*An Example with Spoofax* In Spoofax’ testing language, we can write test cases which specify the number of errors and warnings in a code fragment:

|  |
| --- |
| **test** duplicate entities [[ **module** foo **entity** X {} **entity** X {} ]] 1 **error**  **test** lower case **entity** name [[ **module** foo **entity** x {}  ]] 1 **warning** |

Additionally, we can specify parts of the error or warning messages using regular expressions:

**test** duplicate entities [[

**module** foo

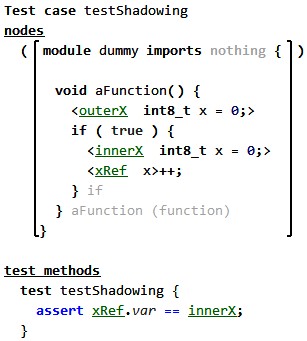


Figure 14.4: Labels and test methods can be used to check that scoping works. In this example, we check shadowing of variables.

|  |
| --- |
| **entity** X {} **entity** X {}  ]] 1 **error** /duplicate/ |

|  |  |
| --- | --- |
| Here, **/duplicate/** is a regular expression that matches error messages like "Duplicate definition of X". As in Xtext and MPS, we can test scopes by means of correct and incorrect references. Alternatively, we can specify the source and target of a link in a test case:   |  | | --- | | **test** property reference [[ **module** foo **entity** X { [[p]]: int **function** f(q: int) {  r: int = 0;  **return** [[p]];  }  }  ]] **resolve** #2 **to** #1  **test** parameter reference [[ **module** foo **entity** X {  p: int **function** f([[p]]: int) {  r: int = 0;  **return** [[p]];  }  }  ]] **resolve** #2 **to** #1  **test** variable reference [[ **module** foo **entity** X {  p: int **function** f(q: int) { [[p]]: int = 0; **return** [[p]];  }  }  ]] **resolve** #2 **to** #1 |   These cases use double square brackets to select parts of the program and specify the expected reference resolving in terms of these selections.  *14.3 Semantics Testing*  Fundamentally, testing the execution semantics of a program involves writing assertions against the *execution* of a program[[2]](#footnote-2). |

In the simplest case this can be done the following way:

* Write a DSL program, based on an understanding what the program is expected to do.
* Generate the program into its executable representation.

to what we do in unit testing "normal" programs. We write a system in a programming language X, and then write more code in X that states assertions about what the system does.

* *Manually* write unit tests (in the target language) that assert the generated program’s behavior based on the understanding of what the DSL program should do.

|  |  |
| --- | --- |
| test cases on DSL level results in more concise and readable |  |
| tests.9 |  |
| The same approach can be used to test execution semantics based on an interpreter, although it may be a little more difficult to manually write the test cases in the target language; the interpreter must provide a means to "inspect" its execution so that we can check whether it is correct. If the tests are written in the DSL and the interpreter executes them along with the core program, the approach works well.  Strictly speaking, the approach discussed here tests the semantics of a *specific* program. As always in testing, we have to write many of these tests to make sure we have covered |  |
| all10 of the possible executions paths through a generator or |  |
| interpreter. If we do that, the set of tests implicitly tests the generator or interpreter – which is the goal we want to achieve in semantics testing.  If we have several execution backends, such as an interpreter *and* a compiler, it must be ensured that both have the same semantics. This can be achieved by writing the tests in the DSL and then executing them in *both* backends. By executing enough tests, we can get a high degree of confidence that the semantics of the backends are aligned. |  |
| *Testing an Interpreter with Xtext* The cooling language provides a way of expressing test cases for the cooling programs within the cooling language itself11. These tests are executed with an interpreter inside the IDE, and they can also be exe- |  |
| cuted on the level of the C program, by generating the program *and* the test cases to C12. The following code shows one of the |  |

Notice how we do *not* test the structure or syntax of the generated artifact. Instead we test its meaning, which is exactly what we *want* to test. An important variation of this approach is the following: instead of writing the unit tests manually in the target language, we can also write the tests in the DSL, assuming the DSL has syntax to express such tests8. Writing the

simplest possible cooling programs, as well as a test case for that program:

**cooling program** EngineProgram0 **for** Einzonengeraet **uses** stdlib { **var** v: **int event** e1

test whether a program written in the DSL works correctly. This is in fact why the interpreter and the test sublanguage have been built in the first place: DSL users should be able to test the programs written in the DSL.

|  |
| --- |
| **init** { **set** v = 1 }  **start**:  **entry** { **set** v = v \* 2 } **on** e1 { **state** s2 } **state** s2:  **entry** { **set** v = 0 }  }  **test** EngineTest0 **for** EngineProgram0 { **assert**-**currentstate**-**is** ^**start** // 1 **assert**-**value** v **is** 2 // 2 **step** // 3 **event** e1 // 4 **step** // 5 **assert**-**currentstate**-**is** s2 // 6 **assert**-**value** v **is** 0 // 7  } |

The test first asserts that, when the program starts, it is in the **start** state (line 1 in the comments in the test script). We then assert that **v** is **2**. The only reasonable way in which **v** can become **2** is that the code in the **init** block, as well as the code in the entry action of the **start** start, have been executed13. We

then perform one step in the execution of the program in line 3. At this point nothing should happen, since no event was

triggered. Then we trigger the event **e1** (line 4) and perform another **step** (line 5). After this step, the program must transition to the state **s2**, whose entry action sets **v** back to 0. We assert both of these in lines 6 and 7.

These tests can be run interactively from the IDE, in which case assertion failures are annotated as error marks on the program, or from within JUnit. The following piece of code shows how to run the tests from JUnit.

|  |
| --- |
| @RunWith(XtextRunner.**class**)  @InjectWith(CoolingLanguageInjectorProvider.**class**) **public class** InterpreterTests **extends** PKInterpreterTestCase {  @Test  **public void** testET0() **throws** Exception { testFileNoSerializer("interpreter/engine0.cool",  "tests.appl", "stdparams.cool" ); runAllTestsInFile( (Model) getModelRoot()); }  } |

The code above is basically a JUnit test that inherits from a base class that helps with loading models and running the interpreter. We call the **runAllTestsInFile** method, passing in the model’s root element. **runAllTestsInFile** is defined by the **PKInterpreterTestCase** base class, which in turn inherits from **XtextTest**, which we have seen before. The method iterates over all tests in the model and executes them by creating and running a **TestExecutionEngine**[[3]](#footnote-3).

|  |
| --- |
| **protected void** runAllTestsInFile(Model m) {  CLTypesystem ts = **new** CLTypesystem(); EList<CoolingTest> tests = m.getTests(); **for** (CoolingTest test : tests) {  TestExecutionEngine e = **new** TestExecutionEngine(test, ts); **final** LogEntry logger = LogEntry.root("test execution"); LogEntry.setMostRecentRoot(logger);  e.runTest(logger);  }  } |

The cooling programs are generated to C for execution in the refrigerator. To make sure the generated C code has the same semantics as the interpreter, we simply generate C code from the test cases as well. In this way the same tests are executed against the generated C code. By ensuring that all of them work in the interpreter and the generator, we ensure that both behave in the same way.

*Testing a Generator with MPS* The following is a test case expressed using the testing extension of mbeddr C. It contributes **test case**s to modules15. **testMultiply** is the actual test case.

It calls the to-be-tested function **times2** several times with different arguments and then uses an **assert** statement to check for the expected value.

|  |
| --- |
| **module** UnitTestDemo {  int32 main(int32 argc, int8\*[ ] argv) { **return test** testMultiply;  }  **test case** testMultiply {  **assert** (0) times2(21) == 42; **assert** (1) times2(0) == 0; **assert** (2) times2(-10) == -20;  }  int8 times2(int8 a) {  **return** 2 \* a;  }  } |

Note that, while this unit testing extension can be used to test any C program, we use it a lot to test the generator. Consider the following example:

**assert** (0) 4 \* 3 + 2 == 14;

One problem we had initially in mbeddr C was to make sure that the expression tree that was created while manually entering expressions like **4 \* 3 + 2** is built correctly in terms of operator precedence. If the tree was built incorrectly, the generated code could end up as **4 \* (3 + 2)**, resulting in 20. So we’ve used tests like these to implicitly test quite intricate aspects of our language implementation[[4]](#footnote-4).

We have built much more elaborate support for testing various other extensions. It is illustrative to take a look at two of them. The next piece of code shows a test for a state machine:

|  |
| --- |
| **exported test case** test1 { **initsm**(c1);  **assert** (0) **isInState**<c1, initialState>; **test statemachine** c1 { start -> countState step(1) -> countState step(2) -> countState step(7) -> countState step(1) -> initialState  }  } |

**c1** is an instance of a state machine. After initializing it, we assert that it is in the **initialState**. We then use a special **test statemachine** statement, which consists of event/state pairs: after triggering the event (on the left side of the **->**) we expect the state machine to go into the state specified on the right side of the **->**. We could have achieved the same goal by using sequences of **trigger** and **assert** statements, but the syntax used here is much more concise.

The second example concerns mocking. A mock is a part of a program that can be used in place of the real one. It simulates some kind of environment of the unit under test, and it can also verify that some other part of the system under test behaves as expected17. We use this with the components extension.

|  |  |
| --- | --- |
| The following is a test case that checks if the **client** uses the **PersistenceProvider** interface correctly. Let’s start by taking a look at the interface: |  |

**interface** PersistenceProvider { **boolean** isReady() **void** store(DataPacket\* data) **void** flush()

}

|  |  |
| --- | --- |
| The interface is expected to be used in the following way: clients first have to call **isReady**, and only if that method returns **true** are they supposed to call **store**, and then after any number of calls to **store**, they have to call **flush**. Let us assume now we want to check if a certain client component uses the interface |  |
| correctly18. Assuming the component provides an operation |  |

**run** that uses the persistence provider, we could write the following test:

**exported test case** runTest {

client.run(); // somehow check is behaved correctly

}

also supports protocol state machines which support the declarative specification of valid call sequences.

|  |  |
| --- | --- |
| To check whether the client behaves correctly, we can use a mock. Our mock specifies the *incoming* method calls it expects to see during the test. We have provided a mocking extension to components to support the declarative specification of such expectations. Here is the mock:   |  | | --- | | **exported mock component** PersistenceMock { **ports**:  **provides** PersistenceProvider pp  **expectations**:  **total no**. **of calls**: 4 **sequence** {  0: pp.isReady **return false**;  1: pp.isReady **return true**;  2: pp.store {  0: parameter data: data != null }  3: pp.flush  }  } |   The mock provides the **PersistenceProvider** interface, so any other component that **requires** this interface can use this component as the implementation. But instead of actually implementing the operations prescribed by **PersistenceProvider**, we specify the sequence of invocations we expect to see. We expect a total number of 4 invocations. The first one is expected to be to **isReady**. We return **false**, expecting the client to try again later. If it does, we return **true** and expect the client to continue with persisting data. We can now validate the mock as part of the test case:  **exported test case** runTest { client.run();  **validate mock** persistenceMock  }  If the **persistenceMock** saw behavior different from the one specified above, the **validate mock** statement will fail, and with it the whole test[[5]](#footnote-5). |

One particular challenge with this approach to semantics testing is that, if an assertion fails, you get some kind of **assertion XYZ failed at ABC** output from the running test case. To understand and fix the problem, you will have to navigate back to the **assert** statement in the DSL program. If you have many failed assertions, or just generally a lot of test program output, this can be tedious and error-prone. For example, the following piece of code shows the output from executing an mbeddr **test case** on the command line:

./TestHelperTest

$$runningTest: running test () @TestHelperTest:test\_testCase1

:0#767515563077315487

$$FAILED: \*\*\*FAILED\*\*\* (testID=0) @TestHelperTest:f:0#9125142491355884683

$$FAILED: \*\*\*FAILED\*\*\* (testID=1) @TestHelperTest:f:1#9125142491355901742

|  |  |
| --- | --- |
| We have built a tool in mbeddr that simplifies finding the message source. You can paste arbitrary text that contains error |  |
| messages into a text area (such as the example above) on the left in Fig. 14.5. Pressing the **Analyze** button will find the nodes that created a particular message20. You can then click on the node to select it in the editor. |  |



*Testing Interpreters and Generators with Spoofax* Spoofax’ testing language also supports testing transformations. We use it to test interpreters, assuming that the interpreter is implemented as a transformation from programs to program results. For example, the following tests address a transformation **eval-all**, which interprets expressions:

**test** evaluate addition [[1+2]] **run** eval-all **to** [[3]] **test** evaluate multiplication [[3\*4]] **run** eval-all **to** [[12]] **test** complex evaluation [[1+2\*(3+4)]] **run** eval-all **to** [[15]]

To test generators, we can rely on Spoofax’ testing support for builders. For example, the following tests use a builder **generate-and-execute**, which generates code from expressions, runs the code, and returns the result of the run as a string:

**test** generate addition [[1+2]] **build** generate-and-execute **to** "3" **test** generate multiplication [[3\*4]] **run** generate-and-execute **to** "12" **test** generate evaluation 1 [[1+2\*(3+4)]] **run** generate-and-execute **to** "15"

*Structural Testing* What we suggested in the previous subsection tests the execution semantics of programs written in DSLs, and, if we have enough of these tests, the correctness of the transformation, generator or interpreter. However, there is a significant limitation to this approach: it only works if the DSL actually specifies behavior! If the DSL only specifies structures and cannot be executed, the approach does not work. In this case you have to perform a structural test. In principle, this is simple: you write an example model, you generate it21,

|  |  |
| --- | --- |
| and then you inspect the resulting model or test for the expected structures. Depending on the target formalism, you can use regular expressions, XPath expressions or OCL-like expres- |  |
| sions to automate the inspection22. |  |
| Structural testing can also be useful to test model-to-model transformations23. Consider the example in Section 11.2.2. There, we inserted additional states and transitions into whatever input state machine our transformation processed. Testing this via execution invariably tests the model-to-model transformation as well as the generator (or interpreter). If we wanted to test the model-to-model transformation in isolation, we have to use structural testing, because the result of that transformation itself is not yet executable. The following piece of Xtend code could be used to check that, for a specific input program, the transformation works correctly: |  |

|  |
| --- |
| // run transformation **val** tp = p.transform  // test result structurally **val** states = tp.states.filter(typeof(CustomState)) assert( states.filter(s|s.name.equals("EMERGENCY\_STOP")).size == 1 )  **val** emergencyState = states.findFirst(s|s.name.equals("EMERGENCY\_STOP")) states.findFirst(s|s.name.equals("noCooling")).eAllContents.  filter(typeof(ChangeStateStatement)).  exists(css|css.targetState == emergencyState) |

This program first runs the transformation, and then finds all **CustomState**s (those that are not start or stop states). We then assert that in those states there is exactly one with the name **EMERGENCY\_STOP**, because we assume that the transformation has added this state. We then check that in the (one and only) **noCooling** state there’s at least one **ChangeStateState- ment** whose target state is the **emergencyState** we had retrieved above24.

### 14.4 Formal Verification

Formal verification can be used in addition to semantics testing in some cases. The fundamental difference between testing and verification is this: in testing, each test case specifies *one* particular execution scenario. To get reasonable coverage of the whole model or transformation, you have to write and execute a lot of tests. This can be a lot of work, and, more importantly,

1. Notice that we don’t write an algorithmic check that closely resembles the transformation itself. Rather, we test a specific model for the presence of specific structures. For example, we explicitly look for a state called **noCooling** and check that this one has the correct **ChangeStateStatement**.

you may not think about certain (exceptional) scenarios, and hence you may not test them. Bugs may go unnoticed.

|  |  |
| --- | --- |
| these algorithms in detail is beyond the scope of this book. Also, formal verification has inherent limitations (e.g., the halting problem) that can only be solved by testing. So testing and verification each have sweet spots: neither can fully replace the other. However, it is very useful to know that verification approaches exist, especially since, over the last couple of years, they have become scalable enough to address real-world problems. In this section we look at two examples from mbeddr: model checking and SMT solving.  *Model Checking State Machines* Model Checking is a verification technique for state machines. Here is how it works in | SAT solving, SMT solving or abstract execution. |
| principle26: |  |
| * Some functionality is expressed as a state machine. * You then specify *properties* about the behavior of the state machine. Properties are expressions that have to be true for every execution of the state machine27. |  |
| * You then run the model checker with the state machine and the properties as input. * The output of the model checker either confirms that your |  |
| properties hold, or it shows a counter example28. |  |
| Conceptually, the model checker performs an exhaustive search during the verification process. Obviously, the more complex your state machine is, the more possibilities the checker has to |  |
| address – a problem known as *state space explosion*. With finite memory, this limits scalability, because at some point you will out of memory, or the verification will run for an unacceptably long time. In reality the model checker does *not* perform an exhaustive search; clever algorithms have been devised that are semantically equivalent to an exhaustive search, but don’t actually perform one29. This makes model checking scalable and fast enough for real-world problems, although there is still a limit in terms of input model complexity30. |  |

Verification checks *the whole program* at once. Various nontrivial algorithms are used to do that25, and understanding

The interesting aspect of model checking is that the properties you specify are not just simple Boolean expressions such as *each state must have at least one outgoing transition, unless it is*

*a stop state*. Such a check can be performed statically, as part of the constraint checks. The properties addressed by model checkers are more elaborate and are often typically expressed in (various flavors of) temporal logic31. Here are some examples, expressed in plain English:

• *It is always true that after we have been in state X we will eventually be reaching state Y*. This is a *Fairness* property. It en-

sures that the state machine does not get stuck in some state forever. For example, **state Y** may be the green light for pedestrians, and **state X** could be the green light for cars.

* *Wherever we are in the state machine, it is always possible to get into state X*. This is a *Liveliness* property. An example could be a system that you must always be able to turn off.
* *It is not ever possible to get into state X without having gone through state Y before*. This is a *Safety* property. Imagine a state machine where entering state **X** turns the pedestrian lights green and entering state **X** turns the car lights red.

|  |  |
| --- | --- |
| The important property of these temporal logic specifications is that quantifiers such as *always*, *whenever* and *there exists* are available. Using these, one can specify global truths about the *execution* of a system – rather than about its structure32. |  |
| Model checking does come with its challenges. The input language for specifying state machines as well as specifying the properties is not necessarily easy to work with. Interpreting the results of the model checker can be a challenge. And for some of the tools, the usability is really bad33.  To make model checking more user friendly, the mbeddr C language provides a nice syntax for state machines, then generates the corresponding representation in the input language of the model checker34. The results of running the model checker |  |
| are also reinterpreted in the context of the higher-level state machine. Tool integration is provided as well: users can select the context menu on a state machine and invoke the model checker. The model checker input is generated, the model checker is executed, and the replies are rendered in a nice table in MPS. Finally, we have abstracted the property specification language by providing support for the most important |  |
| idioms35; these can be specified relatively easily (for example |  |
| **never <expr>** or **always eventually reachable <state>**). Also, a number of properties are automatically checked for each state machine. |  |

Let’s look at an example. The following code shows a state machine that represents a counter. We can send the **step** event into the state machine, and as a consequence, it increments the **currentVal** counter by the **size** parameter passed with the event. If the **currentVal** becomes greater than **LIMIT**, the counter wraps around. We can also use the **start** event to reset the counter to **0**.

|  |
| --- |
| **verifiable statemachine** Counter { **in events** start()  step(**int**[0..10] size)  **local variables int**[0..100] currentVal = 0 **int**[0..100] LIMIT = 10  **states** ( **initial** = initialState ) **state** initialState {  **on** start [ ] -> countState { }  } **state** countState {  **on** step [currentVal + size > LIMIT] -> initialState { } **on** step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size;  }  **on** start [ ] -> initialState { }  }  } |

Since this state machine is marked as **verifiable**, we can run the model checker from the context menu36. Fig. 14.6 shows

the result of running the model checker.

Here is a subset of the properties it has checked successfully (it performs these checks for all states/transitions by default):

|  |  |
| --- | --- |
| State ’initialState’ can be reached | SUCCESS |
| Variable ’currentVal’ is always between its defined bounds | SUCCESS |
| State ’countState’ has deterministic transitions | SUCCESS |
| Transition 0 of state ’initialState’ is not dead | SUCCESS |

The first one reports that NuSMV has successfully proven that the **initialState** can be reached somehow. The second one reports that the variable **currentVal** stays within its bounds (notice how **currentVal** is a bounded integer). Line three reports that in **countState** it never happens that more than one transition is ready to fire at any time. Finally, it reports that no transitions are dead, i.e. each of them is actually used at some point.

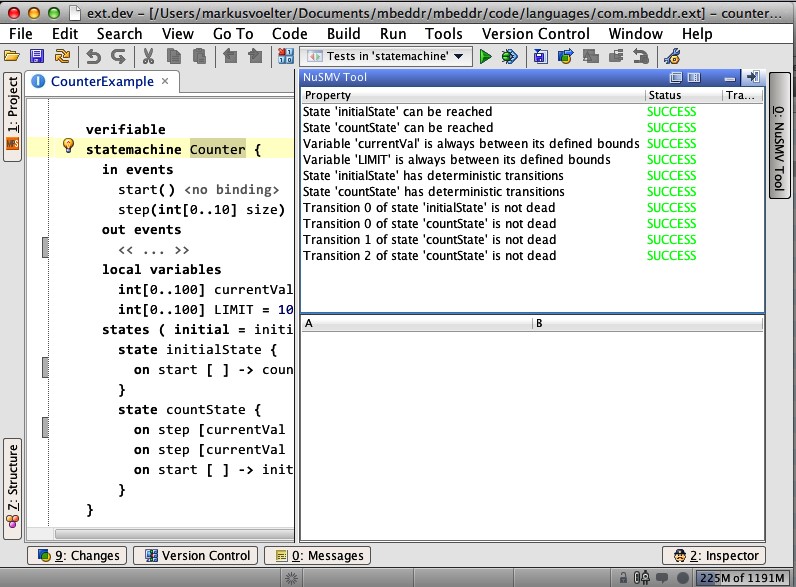
Let’s provoke an error. We change the two transitions in **countState** to the following:

**on** step [currentVal + size >= LIMIT] -> initialState { } **on** step [currentVal + size <= LIMIT] -> countState { currentVal = currentVal + size;

}

We have changed the **>** to a **>=** in the first transition37. Running the model checker again, we get, among others, the following messages:

|  |  |  |
| --- | --- | --- |
| State ’countState’ contains nondeterministic transitions | FAIL | 4 |

This means that there is a case in which the two transitions are non-deterministic, i.e. both are possible based on the guard, and it is not clear which one should be fired. The **4** at the end means that the execution trace to this problem contains four steps. Clicking on the failed property check reveals the problematic execution trace:

|  |  |
| --- | --- |
| State initialState  LIMIT | 10 |
| currentVal | 0 |
| State initialState in\_event: start start()  LIMIT | 10 |
| currentVal | 0 |
| State countState in\_event: step step(10)  LIMIT | 10 |
| currentVal | 0 |
| State initialState  LIMIT | 10 |
| currentVal | 10 |

This is one (of potentially many) execution traces of this state machine that leads to the non-determinism: **currentVal** is 10, and because of the **>=**, both transitions could fire.

In addition to these default properties, it is also possible to specify custom properties. Here are two examples, expressed using the property patterns mentioned earlier:

The first one expresses that we want the model checker to prove that a specific Boolean condition will never be true. In our example, we check that the **LIMIT** really is a constant and is never (accidentally) changed. The second one specifies that wherever we are in the execution of the state machine, it is still possible (after an arbitrary number of steps) to reach the **initialState**. Both properties hold for the example state machine.

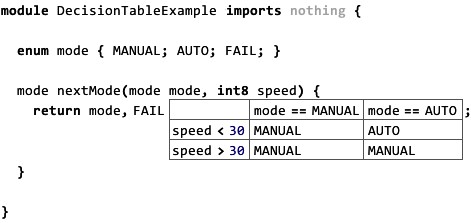
|  |
| --- |
| **verification conditions**  **never** LIMIT != 10  **always eventually reachable** initialState |

*SAT/SMT Solving* SAT solving, which is short for satisfiability solving, concerns the satisfiability of sets of Boolean equations. Users specify a set of Boolean equations and the solver tries to assign truth values to the free variables so as to satisfy all specified equations. SAT solving is an NP-complete problem, so there is no analytic approach: exhaustive search (implemented, of course, in much cleverer ways) is the way to address these problems. SMT solving (Satisfiability Modulo Theories) is an extension of SAT solving that allows other constructs in addition to logical operators – the most frequently used being linear arithmetic, arrays or bit-vectors.

As an example, SMT solving can be used to verify mbeddr’s decision tables. A decision table has a set of Boolean conditions as row headers, a set of Boolean conditions in the column headers, as well as arbitrary values in the content cells. A decision table essentially represents nested **if** statements: the result value of the table is that content cell whose row and column header are **true**. Fig. 14.7 shows an example.

SMT solving can be used to check whether all cases are handled. It can detect whether combinations of the relevant variables exist for which no combination of row header and column header expressions match; in this case, the decision table would not return any value.

SAT and SMT solvers have some of the same challenges as model checkers regarding scalability: a low-level and limited input language and the challenge of interpreting and under-



|  |  |
| --- | --- |
| standing the output of a solver. Hence we use the same approach to solve the problem: from higher-level models (such as the decision table) we generate the input to the solver, run it, and then report the result in the context of the high-level language.  *Model Checking and Transformations* A problem with model verification approaches in general is that they verify only the model. They can detect inconsistencies or property violations as a consequence of flaws in the program expressed with a DSL. However, even if we find no flaws in the model on the DSL level, the generator or interpreter used to execute the program may still introduce problems. In other words, the behavior of the actual running system may be different from the (proven correct) behavior expressed in the model. There are three ways to address this:  • You can test your generator manually using the strategies suggested in this chapter. Once you trust the generator based on a sufficiently large set of tests, you then only have to verify the models, since you know they will be translated correctly. |  |
| • Some tools, for example the UPAAL model checker38, can |  |
| also generate test cases39. These are stimuli to the model, |  |
| together with the expected reactions. You can generate those into your target language and then run them in your target language. This is essentially an automated version of the first approach. |  |

* Finally, you can verify the generated code. For example, there are model checkers for C. You can then verify that the properties that hold on the DSL level also hold on the level

|  |  |
| --- | --- |
| of the generated code. This approach runs into scalability issues relatively quickly, since the state space of a C program is much larger than the state space of a well-crafted state |  |
| machine40. However, you can use this approach to verify the |  |
| generated code based on a sufficient set of relatively small test cases, making sure that these cover all aspects of the generator. Once you’ve built trust in the generator in this way, you can resort to verifying just the DSL models (which scales better).  *14.5 Testing Editor Services*  Testing IDE services such as code completion (beyond scopes), quick fixes, refactorings or outline structure has some of the challenges of UI testing in general. There are three ways of approaching this:  • The language workbench may provide specific APIs to hook into UI aspects to facilitate writing tests for those. |  |
| • You can use generic UI testing tools41 to simulate typing and |  |

clicking in the editor, and checking the resulting behavior.

* Finally, you can isolate the algorithmic aspects of the IDE behavior (e.g., in refactorings or quick fixes) into separate modules (classes) and then unit test those with the techniques discussed in the rest of this chapter, independent of the actual UI.

In practice, I try to use the third alternative as much as possible: for non-trivial IDE functionality in quick fixes and refactorings, I isolate the behavior and write unit tests. For simple things I don’t do any automated tests. For the actual UI, I typically don’t do any automated tests at all, for three reasons: (1) it is simply too cumbersome and not worth the trouble; (2) as we use the editor to try things out, we implicitly test the UI; and (3), language workbenches are frameworks which, if you get the functionality right (via unit tests), provide generic UIs that can be expected to work.

In the remainder of this subsection we show examples of the case in which the language workbench provides specific APIs to test the IDE aspects of languages.

*An Example with MPS* In a parser-based system, you can always type anything. So even if the IDE functionality (particularly regarding code completion) is broken, you can still *type* the desired code. Also, the editing experience of typing code is always the same, fundamentally: you type linear sequences of characters. In a projectional editor, this is not the case: you can only enter things that are available in the code completion menu, and the editing experience itself relies on the editor implementation42. Hence it is important to be able to test editor

behavior.

MPS supports this with a special DSL for editor testing (see Fig. 14.8). Note how MPS’ language composition facilities allow embedding the subject DSL into the DSL for describing the editor test case.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | *An Example with Xtext/Xpect* Xpect is a framework for inte- |  | | gration testing of Xtext DSLs developed by Moritz Eysholdt43. |  | | It can be used for testing various language aspects, among | **/** | | them, for example, code completion44. It does so by embed- |  | |

ding test expectations as comments inside the program to be tested. Here is an example based on a Hello World grammar

(literally):

|  |
| --- |
| Model: greetings+=Greeting\*;  Greeting:  ’Hello’ name=**ID** ’!’; |

The following piece of code shows an example program that includes Xpect statements that test whether code completion works as expected:

helpers for IDE service testing such as content assist or builder tests. They run with, and often even without, an SWT display (e.g., headless).

|  |
| --- |
| // XPECT\_TEST org.example.MyJUnitContentAssistTest END\_TEST  // XPECT contentAssist at |Hel --> Hello Hello Peter!  // XPECT contentAssist at |! --> !  Hello Heiko! |

The Xpect processor processes all comments that start with **XPECT**. In this case, we test the content assist (e.g., code completion) functionality. Let us look at the details:

* The **contentAssist** refers to the kind of test to be executed (details on this below).
* **at** is a keyword for improved readability and has no semantic impact.
* **|Hel** and **|!** instruct the test to search for occurrences of **Hel** and **!** somewhere in the code after the **XPECT** statement. The pipe **|** marks the assumed cursor position relative to **Hel** and **!** where the content assist should be triggered.
* The part after **->** marks the expectation of the test. In the first test, content assist is expected to suggest the keyword **Hello**, and in the second test the exclamation point is expected.

Xpext is in fact a generic infrastructure for integration tests.

As you can see from the example above, the test references a JUnit test class45: **org.example.MyJUnitContentAssistTest**.

|  |  |
| --- | --- |
| The term **contentAssist** is actually the name of a test method inside that class. Everything from an Xpect comment after the **XPECT** keyword is passed as parameters into the test method. | runner which allows you to execute Xpect tests as JUnit test; integration into IDEs such as Eclipse and CI environments is ensured. |
| The test method can do whatever it wants as long as it produces a string as the output. This string is then compared with the expectation, the text behind the **->**. While **contentAssist** is predefined in Xpect-provided unit test base classes, you can define your own methods. Since the actual testing is based on string comparison, the system is easily extensible. The following language aspects can be tested with Xpect46: | 46 The general idea behind Xpect is the separation of test data, test expectations and test setup from implementation details. The test data consists of DSL documents written in the language that you want to test. The test expectations are anything you might want the test to verify and which can be expressed as a string. The setup may declare other DSL documents that the test depends |

* The AST created from a DSL document.
* Messages and locations of error and warning markers.
* Names returned by scopes.
* Proposal items suggest by content assist features (as the example above shows).

on, including Eclipse project setups. Since all these details are hidden, the DSL *user* can potentially understand or even write the test cases, not just the DSL developer.

* Textual diffs that were created by applying refactorings or quick fixes.
* Textual output of a code generator (but use with caution, since generated text may be too fragile).

|  |  |
| --- | --- |
| use the **|** to select offsets inside a DSL document. Xpect also makes it easy to locate the offending expectation if case a test fails: since all expectations are represented as strings, if a test fails, the Eclipse JUnit view provides a comparison dialog that shows all differences between the actual test result and the test expectation. Xpect also makes it easy to specify even large test data scenarios, possibly consisting of multiple languages and multiple Eclipse projects. Finally, since Xpect code is embedded into the DSL code to be tested, you can use your DLS’s Xtext editor to edit your test data. In plain JUnit tests you would have to embed snippets of your DSL documents into Java string literals, which won’t provide any tool support for | ties such as content assist or scoping, you will have to navigate/point/refer to to a model element after you have created the test data. This leads to boilerplate code in Java-based tests. |
| your language at all48. |  |

Results from interpreters or execution of generated code. By embedding test expectations into the subject programs, Xpect implicitly solves the navigation problem47 by allowing you to

*An Example with Spoofax* Spoofax’ testing language supports testing editor services such as reference resolution and content completion. For reference resolution, we mark a definition and a use site in a test case with **[[...]]**. We can refer to these markers by numbers **#1** and **#2**, specifying which marked element should refer to the other marked element. For example, the following test cases mark the name of an entity **A** in its declaration and in the type of a property:

|  |
| --- |
| **test** entity type reference (1) [[ **module** foo entity [[A]] {}  entity B { a: [[A]]  }  ]] **resolve** #2 **to** #1  **test** entity type reference (2) [[ **module** foo  entity B { a: [[A]]  } entity [[A]] {}  ]] **resolve** #1 **to** #2 |

ing the program to be tested, there is no tool support for the Xpect syntax and the expectations. The reason is that Xtext does not support language embedding: there is no way to easily define a composed language from the subject DSL and Xpect. While this limitation is not a problem for Xpect itself (after all, its syntax is extremely simple), it may be a problem for expectations with a more complex syntactic structure. Of course, a specialized editor could be developed (based on the default Xtext editor) that provides code completion for the Xpect code in the DSL program comments. But that would require hand-coding and would be quite a bit of work.

The first test case addresses a backward reference, where the second marked name should resolve to the first marked name. The second test case addresses forward reference, where the first marked name should resolve to the second marked name.

For content completion, we mark only one occurrence, and specify one of the expected completions:

|  |
| --- |
| **test** entity type reference (1) [[ **module** foo entity SomeEntity {}  entity A { a: [[S]]  }  ]] **complete to** "String"  **test** entity type reference (2) [[ **module** foo entity SomeEntity {}  entity A { a: [[S]]  }  ]] **complete to** "SomeEntity" |

Refactorings are tested in a similar fashion. The selected part of the code is indicated with square brackets, and the name of the refactoring is specified in the test:

|  |
| --- |
| **test** Rename refactoring [[ **entity** [[Customer]] {  } **entity** Contract { client : Customer  }  ]] **refactor** rename("Client") **to** [[ **entity** Client {  }  **entity** Contract { client : Client  }  ]] |

### 14.6 Testing for Language Appropriateness

A DSL is only useful if it can express what it is supposed to express. A bit more formally, one can say that the coverage of the DSL relative to the target domain should be 100%. In practice, this questions is much more faceted, though:

* Do we actually understand completely the domain the DSL is intended to cover?
* Can the DSL cover this domain completely? What does

"completely" even mean? Is it ok to have parts of the sys-

tem written in *LD*−1, or do we have to express everything with the DSL?

* Even if the DSL covers the domain completely: are the abstractions chosen appropriate for the model purpose?
* Do the users of the DSL like the notation? Can the users work efficiently with the notation?

It is not possible to answer these questions with automated tess. Manual reviews and validation relative to the (explicit or tacit) requirements for the DSL have to be performed. Getting these aspects right is the main reason why DSLs should be developed incrementally and iteratively.

1. . [↑](#footnote-ref-1)
2. e [↑](#footnote-ref-2)
3. The **TestExecutionEngine** is a wrapper around the interpreter for cooling programs that we have discussed before. [↑](#footnote-ref-3)
4. . [↑](#footnote-ref-4)
5. s [↑](#footnote-ref-5)