Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Department of Computer Science
Information Security Group

Semester Thesis Winter Semester 2006/2007

Automatic Generation of JUnit Test-Harnesses

Manfred Stock

March 6, 2007

Supervising Assistants: Achim Brucker and Jürgen Doser
Supervising Professor: Prof. Dr. David Basin

Abstract

Testing is an important activity in the development of (software) systems. This thesis presents an extension of the Generic Code Generator to support the generation of JUnit test-harnesses. It shows the design of a generic test suite and how such a test suite can automatically be generated from a given UML class diagram of the system. Because class diagrams only contain the structure of a system but not the semantics of its interfaces, this also includes a translation of OCL contracts to Java code. The generated code verifies such constraints and makes use of the Dresden OCL Standard Library.

Contents

1 Introduction

1.1 Motivation

Software engineering increasingly depends on tools which ease and support the development process. The process itself consists of different phases each of which requires a different set of tools: Requirements engineering and design are often done with the help of CASE-tools such as ArgoUML [17], the actual implementation is supported by modern integrated development environments such as Eclipse [6] and the verification may rely upon unit testing frameworks such as JUnit [7].

To assist a security-aware development, the Information Security Group of the Computer Science Department at the ETH Zurich is developing a UML-based tool-chain which supports access control specifications using SecureUML, formal analysis with HOL-OCL and automatic code generation. See Figure 1.1 for an overview of the tool-chain.

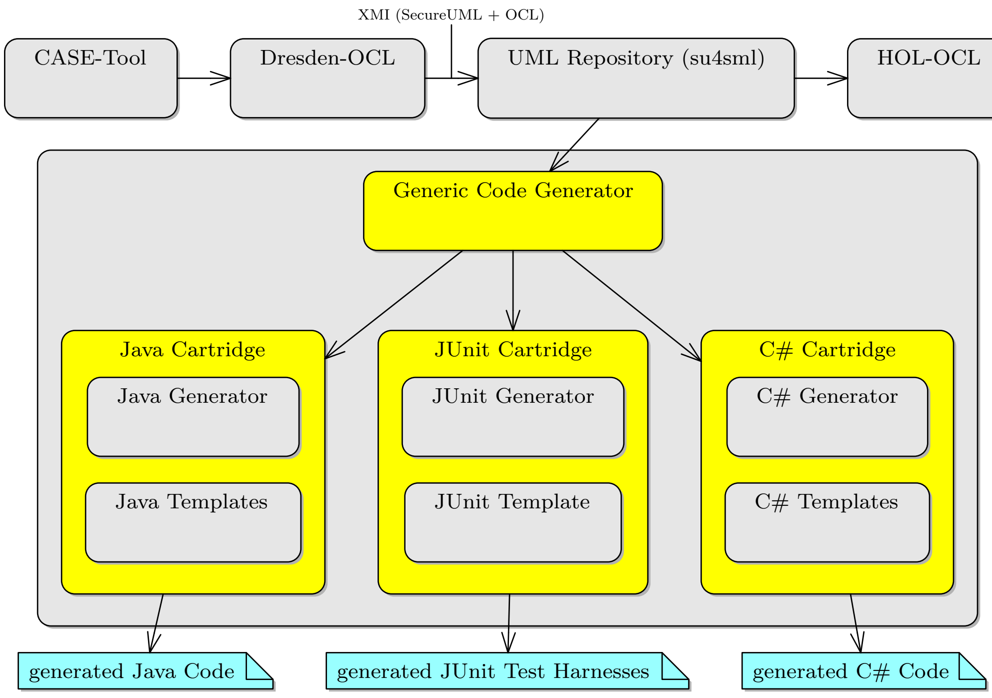

Figure 1.1: Overview of the tool-chain

As automatic code generation does not generate the complete implementation of a system, failures may exist in the final system even if the model was provably correct because a programmer may have introduced faults. Then again, even a complete and automatic generation of an implementation may exhibit failures when run on a real system. One way to find failures is software testing and because testing generally plays an important role in the development of software systems, the automatic generation of test-harnesses is a sensible extension of the tool-chain. Such an extension is also expedient as test-harnesses for different projects and unit tests for different classes often share a similar structure and since it bridges the design and the validation and verification phase of the development process: Some constraints which have been defined on the model by using, for example, OCL may automatically be leveraged for the use in unit tests. On the other hand, the structure of the system as specified in the model can be reflected in automatically generated skeletons of unit tests which should be completed by the programmer and may already contain constraint checking code. Figure 1.1 also shows where the aforementioned extension, which is presented in this thesis, fits into the tool-chain.

1.2 Background

1.2.1 Modeling

The design process that is used when a new system is built normally starts with the construction of an abstract model of this new system. A widely used language to model software systems is the Unified Modeling Language (UML). It provides several diagram types to model different aspects of the system: Structure diagrams such as class diagrams are used to model the structure of the system whereas behaviour diagrams like state machine diagrams and interaction diagrams such as sequence diagrams are used to model the dynamic behaviour. An example of a simple class diagram may be found in Figure 1.2: It shows a package `university` with three classes where `Professor` and `Student` are specialisations of the `Person` class. All the classes contain private fields which store the state of the object and several public methods which allow other objects to modify or query the current state.

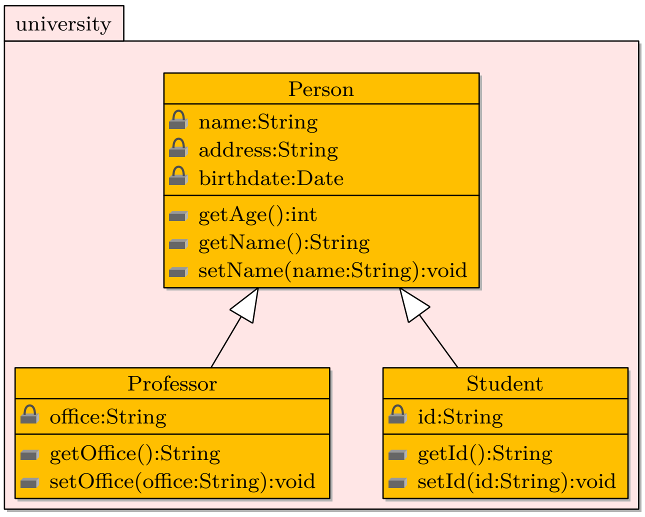

Figure 1.2: Example of a class diagram

Object Constraint Language

Class diagrams do not contain any explicit information about the semantics of the operations of the modelled classes. Such interface specifications may be added by using the Object Constraint Language (OCL) [12] which is a declarative language for the description of rules which apply to UML models. OCL can be used to specify pre- and postconditions of operations and invariants of classes – see Listing 1.1 for an example of such contracts for the classes of Figure 1.2.

1.2.2 Unit Testing

In the context of object oriented programming, the unit of programming elements is a class. Unit tests are testing the interface of a unit by comparing the actual to the intended behaviour. This can only find failures but not faults. Tests nevertheless might give an indication of where one

```
package university
context Person
    inv fields_nonnull: self.birthdate->notEmpty() and
                        self.name->notEmpty() and
                        self.address->notEmpty()

context Person::getAge():Integer
    post positive_age: result >= 0

context Person::setName(name:String):void
    pre name_given: name->notEmpty()
    post name_set: self.name = name

context Student
    inv id_nonnull: self.id->notEmpty()

-- context Professor...
endpackage
```

Listing 1.1: An OCL example

has to look for the actual fault which was the cause of the failure, therefore they are important during the implementation and the validation phase. In addition, a thorough test suite, which is a collection of unit tests, can provide a good sense for the overall quality of the code. It may as well simplify and endorse changes to existing code since the changed code can easily be tested for regressions. A good test is also executable without any interaction which allows for a fully automatic run of the whole test suite after every build of the software system.

A test-harness is a collection of software and test data which is prepared for the execution of unit tests. The software in this case is a unit testing framework, such as JUnit, which is used by the actual unit tests and the test data is often directly written into those unit tests but could also be provided separately, which simplifies modifications.

JUnit

JUnit [7] is presumably the most famous unit testing framework for Java and was written by Erich Gamma and Kent Beck. It originates from Kent Beck's SUnit [2] for Smalltalk which is the original source for a whole family of unit testing frameworks collectively known as xUnit [21]. For an example of a simple JUnit test, see Listing 1.2. JUnit 4 provides, among others, the following facilities:

- Several annotations, for example:
  - `@Test` is used the to tell the framework that the following method shall be executed as a unit test.
  - `@Before` or `@After` can be used to annotate methods which should be executed before or after every `@Test` method. They can be used to setup or tear down the environment.
  - `@BeforeClass` and `@AfterClass` are similar, but they are only executed once, namely before the class with the test cases is instantiated by the framework for the first time and after the last test case of the class has been executed, respectively. An application for these methods would be the setup of database connections and other tasks which involve more resources and therefore should not be executed before or after every single test case.
  - `@Ignore` can be used to mark a `@Test` method to be ignored by the framework. It will nevertheless show up in the statistics but it will be marked as ignored. This is, for example, useful if the test or the tested feature is not completely implemented yet.

```java
package university;

import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.assertEquals;

public class PersonTest {

    private Person p;

    /**
     * Setup method which creates a Person object.
     */
    @Before
    public void setup(){
        p = new Person();
    }

    /**
     * The actual unit test.
     */
    @Test
    public void getAgeTest(){
        // Test if the age equals 20
        assertEquals(p.getAge(),20);
    }
}
```

Listing 1.2: Example of a simple JUnit test

- Test runners which can be used to execute the tests and collect statistics about successful, ignored and failed tests. JUnit provides only simple, text-based test runners because modern software development environments such as Eclipse contain test runners which are integrated into their graphical user interface.

- Some useful assertions such as `org.junit.Assert.assertEquals()` which can be used to check the actual result against the expected one.

To make use of JUnit, the programmer needs to provide the actual unit tests and some test data. In most cases, special setup and tear down methods will be required too, as well as a test strategy which describes among other things the layout of the test suite.

The sequence of actions during the execution of unit tests such as the one given in Figure 1.2 can be described as follows: A test runner looks for `@Test` annotated methods in a given class which are not annotated with `@Ignore`. If there are such methods, it executes the static `@BeforeClass` annotated methods which should initialise more expensive testing setups. This initialisation could include, for example, the setup of database connections or the parsing of configuration files. Then the test runner creates a first new instance of the class with the test cases and executes the `@Before` annotated methods followed by a call to the actual `@Test` annotated method which implements the first test case. JUnit creates a new instance of the class with the test cases for every execution of an actual test case. When the test has finished, the test runner executes the `@After` annotated methods and records the result, that is, whether an unexpected exception was thrown or whether an assertion was triggered, both of which result in a failed test case. If there are no more test cases in the class, the `@AfterClass` annotated methods get called and the results are returned. Depending on the actual test runner, these may be displayed as text or graphically in a GUI.

1.2.3 Workflow

An overview of the intended workflow can be found in Figure 1.3. It would usually start by creating a UML model with the help of a CASE-tool such as ArgoUML and by specifying OCL constraints for the model. The outcome of this step currently needs to be combined into one file using the Dresden OCL Toolkit and is then processed by the Generic Code Generator which emits source code stubs. These stubs are then customised through the user by adding the actual implementation and other details which could not be generated automatically. During this phase, periodic executions of the unit tests are used to verify the implementation. Finally everything should work as intended and the system can be shipped.

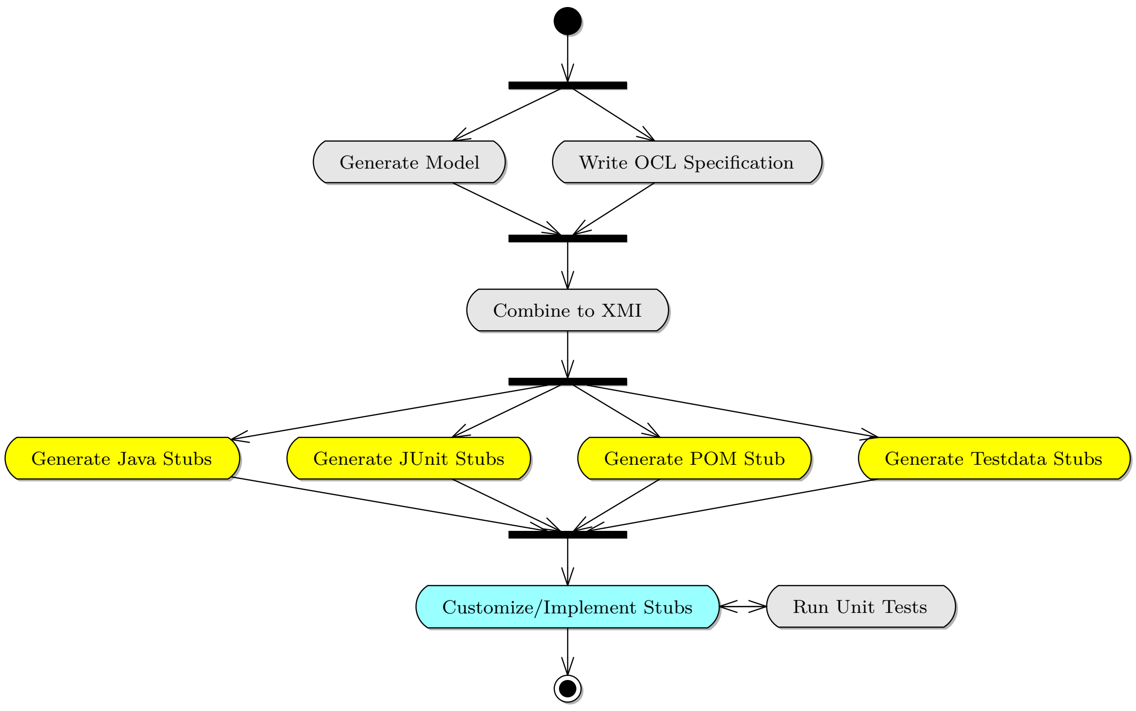

Figure 1.3: The intended workflow

1.3 Related Work

This thesis is about the automatic generation of JUnit test-harnesses and also includes the implementation of a library for access to test data. Furthermore, it comprises the automatic generation of program code which verifies interface specifications that are given in the form of OCL pre- and postconditions and invariants. Several projects in these areas already existed and the most important ones for the context of this thesis will be mentioned in the next sections.

1.3.1 Unit Testing

There is a large amount of unit testing frameworks available many of which are listed in an overview on Wikipedia [20]. This thesis focuses on *JUnit* [7] which is presumably the most famous unit testing framework for Java and more thoroughly described in section 1.2.2. Nevertheless, the presented concepts are mostly independent of JUnit and Java and therefore could be implemented for other languages as well.

Apache Maven [18] and build tools in general may also be put in the context of unit testing as they can execute unit tests as part of the build cycle. Since Maven is used in this thesis, some of its useful features and an extension of the code generator to support the generation of basic project descriptor files for Maven will briefly be described in section 3.2.3.

1.3.2 Access to Test Data

There are also many frameworks or libraries which facilitate the access to test data from within unit tests, so only those which inspired the work of this thesis will be mentioned. One of them was *JTestCase* [15] which is a framework that can be used to separate unit tests from the test data. It uses an XML based file format to store the test data and provides object oriented means to access this data from within the unit tests. The other one was *DDSteps* [14] which uses external test data in Excel or XML files that is injected into test cases which are then run once for each row of data. This thesis takes a similar approach but does not use XML for the storage of test data.

1.3.3 Interface Specifications

Since the tool-chain mentioned in section 1.1 is UML-based and already had support for the *Object Constraint Language (OCL)* [12] which is part of the UML standard, this was the specification language used in this thesis. *Design by Contract* [11] is a paradigm for designing computer software which prescribes the use of checkable interface specifications. It was first implemented in the Eiffel programming language which is organised around the concepts of Design by Contract. The *Java Modeling Language (JML)* [9] is an interface specification language which is specific for Java and also follows the Design by Contract paradigm. In JML, specifications are directly written into the Java source code files in the form of special comments.

Dresden OCL Toolkit

One part of this thesis consists of the translation from OCL constraints to Java code that should be executed during unit tests. This generated code uses the *Dresden OCL Toolkit* [1] which contains the implementation of an OCL Standard Library for Java. The library implements the predefined primitive types and also the collection types of OCL, including the operations defined on these types. The toolkit also comprises a code generator and a tool to combine the XMI description file of UML models with external OCL specifications into one common XMI file.

1.3.4 Automatic Code Generation

The *Generic Code Generator (GCG)* [4] which is used and extended in this thesis was the subject of an earlier semester thesis. It uses a template-driven approach to code generation and so called cartridges as an extension mechanism.

JMLUnit [3] was developed in the context of JML, the Java Modeling Language, and makes use of JML interface specifications to automatically generate test oracles. It employs JUnit to execute the generated unit tests. Test data is provided in an abstract class and the actual test extends this class, which has the advantage that the concrete class may be overwritten without losing the test data. This approach inspired the class hierarchy of the generated test suite which will be presented in section 2.2.1.

A different approach is taken by *JUnitDoclet* [8] which generates test suites with skeletons of test cases based on the application source code. It therefore does not start with a specification or model of the system but with the source code of the actual implementation. For each package, it generates a test suite containing the test cases of its classes. Every public method gets a test method, and getter/setter tests are combined into one test case. The generated tests can be modified and the changes are preserved over regenerations of the test suite.

2 Design

2.1 Design of a Generic Test Strategy

There are different strategies for unit and regression testing one could come up with, each with its own advantages and disadvantages. The approach taken in this thesis tries to separate test data from the actual tests and employs one test class for every class of the system which has non-private methods. There is one test method for every non-private method of the tested class.

The separation of test data and actual tests allows changes to the test data without the need to modify the tests themselves. It also simplifies these modifications for the tester and relaxes dependencies on the implementation language of the system because of the higher degree of abstraction in the test data representation. This of course leads to the need of means to access test data from within unit tests, which is fulfilled by a library that parses a file containing test data and provides access to this test data.

2.2 Layout of the Test Suite

The test strategy from section 2.1 implies some requirements for the overall layout of the test suite: Because there should be a test method for every non-private method, access to protected and default-access methods must be possible, therefore the test class must be in the same package as the tested class. Though this could be relaxed as these methods are not part of the public interface of a class, it is a useful feature for the programmer and may increase test coverage.

Furthermore, a file format is required to allow for the separation of tests and test data. Such a format should fulfil two goals:

1. Easily editable: The files must be editable by programmers and testers and should therefore not involve too much syntactic overhead. This is the reason why XML fell flat for this task.

2. Simple to parse and generate: Because these files are parsed during every run of the test suite, they should not involve too much processing overhead. On the other hand, they should be easy to generate as well since it would be useful if a skeleton was generated automatically. Such a skeleton then could already contain test data generated by HOL-TestGen.

As mentioned in section 1.2.2, setup and tear down methods might be required by the tests. Since such methods often cannot be generated automatically, there needs to be a way to preserve these methods over successive automatic generations of the test suite (which would be required whenever changes in the model occur which need to be reflected in the test suite). This is also valid for custom methods used to compare the actual and the expected output. The solution of this problem was inspired by JMLUnit [3]: They generate an abstract and a concrete class at first and only overwrite the concrete class in later runs of the code generator. In the next few subsections, the actual layout of the test suite will be explained more in-depth using the example of section 1.2.1.

2.2.1 Class Hierarchy

For every class of the system, there are two automatically generated classes in the test suite which serve different purposes:

1. An abstract class, which can and often must be customised by the user. This class contains all the custom setup and tear down methods as well as custom methods to check the actual outputs. One task which must be fulfilled by this class is the initialisation of an instance of the class under test which is used as the target object to call the currently tested method on. In addition, this class may contain methods which produce inputs and expected outputs which are not of a basic data type such as integer or string. In the case that the automatically generated test suite does not allow certain aspects or methods of the system to be tested easily, is is also possible to insert custom `@Test` methods into this class which are also executed by JUnit test runners.

2. A concrete class, which contains all the automatically generated `@Test` methods corresponding to the non-private methods of the tested class. This class should not be modified because it gets overwritten on subsequent runs of the code generator. Besides the test methods, this class contains code which checks OCL pre- and postconditions as well as invariants of the tested class: If a precondition fails with the given test data, the test is not executed as this test data was not valid, but if the postcondition or the invariant fails after executing the tested method, there obviously is a deviation from the expected behaviour which results in a failed test case. The concrete class is also responsible for the initialisation of the library that provides access to the test data. It must therefore contain the path to the test data, the name of the tested class and the names of its non-private methods.

See Figure 2.1 for the class hierarchy which would be generated for the classes of Figure 1.2. All generated classes are in the same package as the tested class to allow access to protected and default-access methods. The concrete classes extend the abstract ones and are used by JUnit test runners. They implement the `TestDataUser` interface to give some helper classes of the test data access library access to information about the currently tested class.

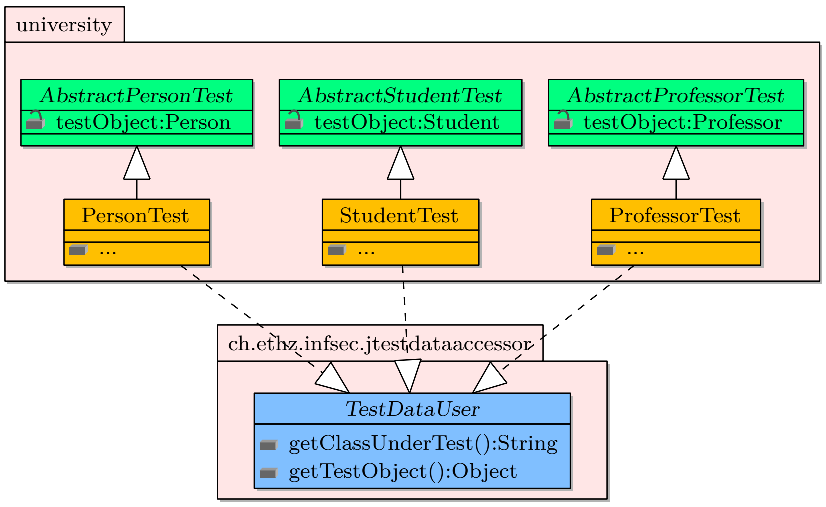

Figure 2.1: Class hierarchy of the generated test suite

2.2.2 Directory Layout

The directory layout which is generated by the code generator for Java and JUnit was adopted from Apache Maven [18]: Maven uses a standard directory layout which separates the implementation from the tests and generally follows best practices. Besides the advantage that a project using this layout can easily be built using Maven, the layout seems to make sense as it separates different aspects of the system such as the implementation from tests and general documentation like the project website. For the class diagrams in Figure 1.2 and Figure 2.1, the directory layout would look like the one in Listing 2.1. The listing also shows where the test data resides.

```
src/
|--main/
|   `--java/
|       `--university/
|           |--Person.java
|           `--...
`--test/
    |--java/
    |   `--university/
    |       |--AbstractPersonTest.java
    |       |--...
    |       |--PersonTest.java
    |       `--...
    `--resources/
        `--university/
            |--TestdataPerson
            `--...
```

Listing 2.1: An example of the directory tree

2.2.3 Test Data Format

Because tests of functions usually take some input and compare the output to a given value, this must be supported by the file format. An apparent problem is the difficulty of representing complex object structures in a simple file format. This can be solved by specifying methods producing those object structures in the test data file. The actual methods would be implemented in the abstract classes mentioned in section 2.2.1, which has the advantage that the test data file does not contain language specific test data.

Since the result or outcome of a method might not be checkable by a default verifier from JUnit, it should be possible to declare custom verifiers in addition to the default ones. They would then be used to actually check the result.

Another necessary feature is the ability to specify setup and tear down methods for each tested method, which would be called before and after every test execution with a given set of test data. JUnit's annotations are not useable here because JUnit does not know about the different sets of test data – for JUnit, several sets of test data still look like one test case because they are all used in one `@Test`-annotated method.

If overloaded methods should be tested as well, there must be a way to specify the parameter types of the tested method because the method name is not enough to identify them. An optional description or name for the test case would certainly be useful in case of a failed test as this information could be included in exceptions or assertions.

To combine all these requirements, a proprietary file format was devised. Listing A.1 in the appendix shows the grammar for the format. It is loosely based on the well-known INI-file format which contains different sections and several key-value-pairs in these sections. See Listing 2.2 for an example.

The sections contain two different scopes, namely a global one which sets general options for the execution of a test case and may contain the following keys:

`resulttype`: The return type of the tested method if it is not `void`.

`inputtypes`: A comma separated list with parameter types of the tested method, if any.

`setup`: Setup method which will be called each time before the tested method is executed with a new set of test data.

`teardown`: Tear down method analogous to the setup method.

```
[getAge]
resulttype = int;         # Type of the result
setup = setup0();         # Setup the object
{
    result = 9;           # Expected result
    checker = EQUALS;     # Check of result
    comment = "Test if the result is 9";
}

[setName]
inputtypes = String;      # Type of the input
setup = setup1();
{
    input = "Foo";        # Provided input
    checker = resultcheck0();
}
```

Listing 2.2: Example of a file with testdata

The other scope is the one containing the test data and is enclosed in curly braces. There may be several such blocks with different sets of test data and they may contain the following keys:

`input`: A comma separated list of inputs. It may contain values of basic types such as integers or strings, but also names of methods which return a suitable value.

`result`: The expected result or a method which generates it. This could also be an exception if the tested method should throw one for the given input.

`checker`: A default checker provided by JUnit such as `EQUALS` or a method which verifies the result.

`comment`: A comment to describe the current set of test data.

Comments in the test data file start with the `#`-sign and they continue until the end of the line. The title of a section matches the name of the method the test data sets in its body are for. Since different sets of test data may require different setup and tear down methods, there may be more than one section with the same title.

2.3 Code Generator Extension for JUnit Support

The existing Generic Code Generator (GCG) from [4] primarily supported the code generation for C# and some variants thereof, but had only tentative support for Java, hence several improvements of the Java cartridge and template were needed. Because the generation of JUnit test-harnesses also involved some other requirements which were not fulfilled yet, multiple small extensions of the code generator were implemented during this thesis. One example is an alternate way to open files in a template such that existing files with the same name are not overwritten. This is required to preserve changes made by the user as mentioned in section 2.2.

2.3.1 Test Cartridge

To support a new output language, the code generator requires a new template and probably a new cartridge which supports additional list, string or boolean variables. Since JUnit tests are written in Java, the JUnit cartridge extends the Java cartridge and adds some new variables which are only required in JUnit tests. Section 3.2.2 lists these new variables.

2.4 Pre- and Postcondition Checking

OCL contracts for methods and class invariants can be used to verify that the actual behaviour of the system matches the intended one. As this is exactly what testing is all about, it is sensible to integrate constraint checking code into unit tests. One way to achieve this is the use of the Dresden OCL Toolkit as mentioned in section 1.3.3, namely the OCL Standard Library implementation which is part of the toolkit. There are two options for the actual integration of the toolkit since it already contains a constraint code generator for Java:

1. Use the GCG to generate Java stubs or take existing code and then apply the Dresden OCL code generator to inject constraint checking code. This approach has the advantage that Dresden OCL does all the OCL handling, but it changes (and therefore requires) the code of the actual implementation which might not be an option in certain cases.

2. Extend the GCG to generate Java code that makes use of the Dresden OCL Standard Library. This code could be placed in the actual implementation of the system (with the same problem as above) or directly into the unit tests. The benefit of this approach is that both options persist - the JUnit code generator could insert the code into the unit tests, or a special Java template could generate Java stubs containing the constraint checking code. The drawback is that the implementation might be fairly limited as a complete implementation would go beyond the scope of a semester thesis.

The approach which was actually taken is the second one.

3 Implementation

3.1 jtestdataaccessor

The `jtestdataaccessor` is the Java library which parses the test data file into a data structure which can be used in unit tests. It uses the ANTLR parser generator [13] for the construction of the actual parser and JUnit assertions to signal failed tests. Besides providing access to test data which is described below, the library also contains other functionality which is useful in the given context:

Invocation of the tested method: Since the test data file may contain several sections for the same method and therein different sets of test data, the same method of the tested class might need to be executed fairly often. Because the code which implements this would look the same in every case, it has been put in a method of a helper class which is used by the actual unit tests. This method also takes care of calling setup and tear down methods as well as initiating the verification of the result (see below). In the case of an exception during the execution of the tested method, it is caught and handled appropriately:

- If the precondition of the method failed, the exception is ignored as the test data was not valid, but a message gets printed which indicates this.
- If the postcondition or the invariant failed, the standard JUnit `org.junit.Assert.fail()` assertion gets triggered which results in a failed test case.
- If an exception of the expected type was thrown, the method discards it. This behaviour is similar to JUnit when the `@Test` annotation was specified with an optional parameter as follows: `@Test(expected = <Class of the expected exception>.class)`. In this case, JUnit requires the exception to be thrown, else the test failed. Such a feature is useful to also test the exceptional behaviour of a class.

Verification of the result: Because one can specify a verifier for the result in the test data file, there is a method which executes this verifier with the result of the tested method. It delegates most of its work to JUnit assertions, but it may also call a user defined function.

3.1.1 Lexer/Parser

The lexer gets the characters of the test data file as input and converts them into lexical tokens. While processing the tokens from the lexer, the parser then constructs the internal tree to represent the test data in an object oriented data structure. Since the input is user defined, it may contain errors, therefore some sanity checks are done during parsing and exceptions thrown if they fail.

3.1.2 Data Structure

For the internal storage of the test data, a data structure was required which could represent all the elements of a test data file. A simplified diagram of the devised structure can be found in Figure 3.1. It consists of two main parts: (1) The `TestDataAccessor` class as entry point to the library among the classes in the top level package which represent the different scopes in the test data file and (2) the classes of the `nodes` package which stand for the right hand sides of the keys mentioned in section 2.2.3. In more detail, the classes of the diagram fulfil the following tasks:

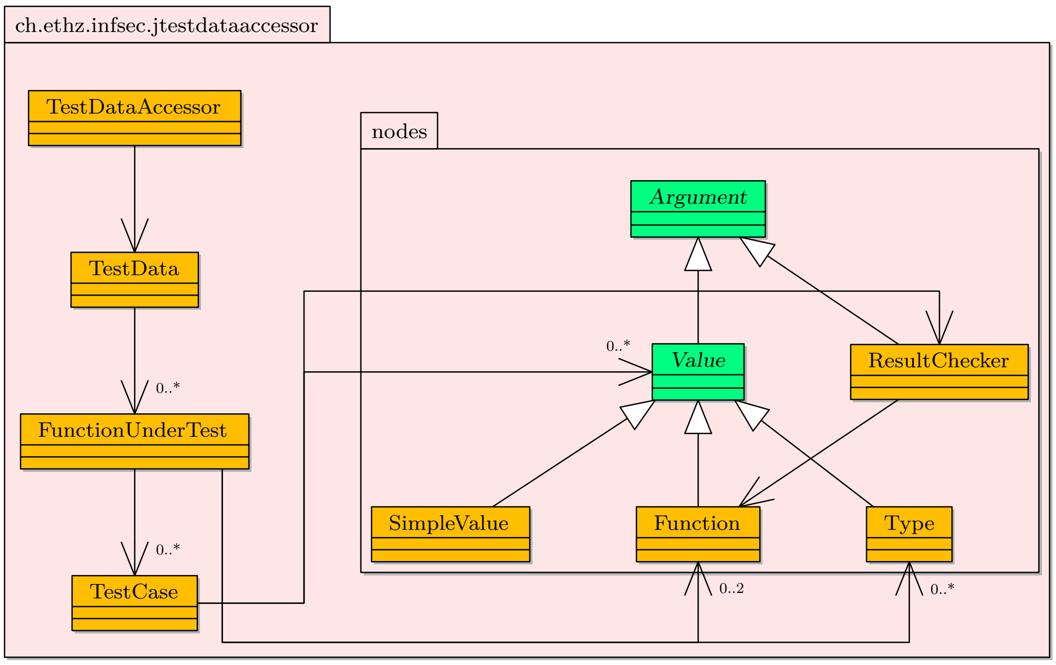

Figure 3.1: Simplified class diagram of the `jtestdataaccessor` data structure

`TestDataAccessor`: Upon instantiation with the path to a file containing test data, the file is parsed into the internal data structure. This class therefore encapsulates a `TestData` object which is passed to the parser for initialisation and it provides access to the test data for a given function.

`TestData`: This class represents the root of the test data data structure. It stores a list of `FunctionUnderTest`-objects for each function which had at least one section in the test data file.

`FunctionUnderTest`: Since this class represents a section of the test data file, it stores references to setup/tear down `Function` objects as well as a return `Type` object and a list of parameter `Type` objects. It also contains a list of `TestCase` objects because a section may contain several sets of test data that are parsed into such a list.

`TestCase`: A test case consists of input and expected output, so this class stores a list of `Value` parameters to the function and a result `Value` which constitutes the expected result. Besides this, is also contains a reference to a `ResultChecker` object which is used to verify the result and a comment which describes the test data stored in this object.

`Argument`: This abstract class is the root of the `nodes` hierarchy and stands for the right hand side of a definition in the test data file.

`Value`: Since the right hand side of a definition may interchangeably contain values of basic types, “function calls” or names of types, the objects which represent those values need to have a common interface to get the actual value at runtime. This is fulfilled by the abstract `Value` class which specifies such an interface.

`SimpleValue`: Basic values such as integers or strings are stored in instances of this class.

`Function`: Several keys of the test data file accept “function calls” in the arguments where the value gets retrieved at runtime by calling a method of the abstract class of the test-harness. Another application is the specification of setup and tear down methods which need to be executed before or after the execution of the tested method, respectively. The actual function call is done using the reflection features of Java.

`Type`: The declaration of input and return types required a class for the representation of class objects, therefore this class searches for a class with a given name and returns the corresponding class object. Because instances of basic data types are different from normal objects in Java, they require special handling when retrieving their class object. A solution for this problem is a simple mapping from the name of the basic data type to the corresponding class object.

`ResultChecker`: This class makes up the verifier of a `TestCase` which is either a default verifier from JUnit or a user defined function.

3.2 Code Generator Extension

As mentioned in section 2.3, the generic code generator required some extensions and a new cartridge for JUnit support. Because several minor improvements were just implemented on demand and not specific to this thesis, they will not be mentioned in this report.

3.2.1 Java Support

The existing support for Java was only very basic and therefore not useful. Code generation for classes was working to some degree, but certain information such as the visibility of methods was not used, so the generated code was quite different from the model.

Extensions to the Java cartridge include among others:

- A new list variable `parent_interface_list` to allow the iteration over parent interfaces of a class or interface which is required for proper code generation when interfaces are used.
- Some new boolean variables like `operation_is_void` to check for example if an operation is `void` or `operation_has_arguments` to test if it has parameters.
- New string variables to include code for pre- and postconditions as well as invariants, to insert the name of the current parent interface (`$parent_interface$`) and to define a stub for the return value of a method (`$returnvalue_stub$`). The latter is useful if one wants to generate code that can immediately be compiled because a Java compiler complains if a non-void method does not return a value.

The Java template was mostly rewritten since it had only support for classes and was setting the visibility of methods and attributes to `public`, no matter what the specification stated. Now it supports classes, interfaces and enumerations and takes care of correctly specifying the visibility. For methods which do return a value, it also inserts a `return` statement which returns a default value matching the return type, that is a numeric return type results in 0, a string return type in an empty string and objects in `null`.

3.2.2 JUnit Support

For the actual JUnit code generation, the cartridges are plugged together such that the JUnit cartridge inherits the functionality of the Java cartridge, therefore all the features from the latter are available in the Java cartridge as well. The Java cartridge thus only contains few extensions to the template language:

- A new list variable `unique_operation_list` which contains every operation name only once since several overloaded methods have the same name. This is no problem when the parameter types are used, but because there is only one unit test per method name, this would result in a collision.

- To check if a class is testable, a boolean variable `isTestable` was introduced. It is false for interfaces, abstract classes and classes which do not contain methods. Additionally, `operation_isNotPrivate` was added since private methods cannot be tested as well.
- Because the validity verification of contracts is slightly different when they need to be embedded into unit tests, string variables which override those from the Java cartridge were introduced to generate code that is adapted to the unit tests. The main difference is that the code is not directly embedded in the classes that must fulfil the constraints, but in the unit tests themselves. Therefore attribute accesses and method calls cannot be done on `this` but on a local instance of the tested class.

The template for JUnit is fairly complex as it has to create three different files for each testable class:

Concrete class: This class contains a `@BeforeClass` annotated method which sets up the environment, that is, it creates a `TestDataAccessor` object with the test data for the tested class and it also sets up a helper class.

For each testable method, there is a `@Test` annotated method which initiates the test using a function of the helper class by passing it the name of the tested method.

Because contracts must be checked before and after the execution of a tested method, for each such method a wrapper function is created which contains the code to check pre- and postconditions. Additionally, a method that verifies the invariant is generated and called after the postcondition checks.

Abstract class: As this class will be modified by the user, it only contains a reference to an instance of the tested class which will be used to call the tested methods. This way it is mostly independent from changes to the model and does not need to be updated.

Testdata stub: The generated stub already contains a section for each non-private method of the tested class which is provided with values for the `resulttype` and `inputtypes` keys since these values are known at code generation time. Overloading of methods is thus supported as well. The test case scope can be generated with some sensible values, too: If the method does not have any parameters, there cannot be an `input` key, and if the method is `void`, `result` is not required, either (except if exceptional behaviour shall be tested). All keys without value which are generated as stubs are commented since the parser would not accept them.

3.2.3 POM Support

POM stands for Project Object Model and is the XML file format used in Apache Maven [18] as a project descriptor. A basic POM file contains a group and artifact id as well as the version number of the artifact. Furthermore, a dependency section is used to specify other artifacts the project depends on which are automatically fetched, locally cached and included in the class path when Maven is run. There are many other useful features and a large number of plug-ins available which can simply be listed in the POM that will be retrieved and used automatically (sometimes depending on the specified goal Maven gets called with). The advantage of downloading dependencies and plug-ins on demand is that they do not need to be installed locally by hand nor stored in the revision control system.

As mentioned in section 2.2.2, the directory layout of the generated test suite was adopted from Maven, so a new cartridge and template pair to generate a basic POM file is another sensible extension of the code generator. Because the `jtestdataaccessor` was also built with Maven and since it is possible to inject arbitrary JAR files such as the Dresden OCL Standard Library into the local Maven repository, dependencies on these libraries can be specified in the generated POM. This allows an immediate initial compilation of the system and the test suite using Maven.

The template for the POM generation did not require any new features, thus the cartridge is only minimal. It was nevertheless created since future versions of the `su4sml` UML repository (see

Figure 1.1) may provide more information which could be used for the group and artifact ids of the POM or the project name. These values are currently set to fairly meaningless defaults, so the user should customize the POM.

3.3 OCL Integration

To support the generation of Java code that evaluates OCL expressions for the verification of pre- and postconditions as well as invariants, a new SML structure `Ocl2DresdenJava` was implemented. It takes an OCL term and returns Java code which makes use of the Dresden OCL Standard Library to evaluate the constraint.

In addition to the large number of possible OCL expressions which could be supported, two main problems were encountered during the development of this extension:

1. A straightforward way to evaluate such expressions would be to simply nest the corresponding Java expressions. Since the resulting code would be completely unreadable and long expressions might trigger limits of the Java compiler, this was not a feasible option. An alternative solution which was also implemented in the Dresden OCL code generator is the use of temporary variables to evaluate each subexpression. Since these variables require unique names, this cannot be implemented easily in a purely functional way.

2. Similarly, `@pre` expressions posed another problem: There must be a way to store the values from the pre-state of the method execution which makes them accessible in the post-state.

The solution to these problems involved a diversion from the pristine path of functional programming: Two SML structures were introduced which contain functions that are not side-effect free:

1. `varcounter` implements a simple counter which is incremented on every use. This allows the creation of unique variable names which can be used to generate the temporary variables.

2. `preMap` is a mapping from a string to a variable id. The evaluation of `@pre` expressions is done by looking for such expression in the postconditions of a method during the code generation for the preconditions. When such an expression is found, code for the evaluation of its argument is generated and the variable id of the variable with the result is stored in the map. The key in the map is a string representation of the expression. Now it is possible to look up the variable which corresponds to an `@pre` expression in the postcondition.

3.3.1 Supported OCL Subset

Since a complete implementation of a translation from OCL to Java was not feasible, only a subset of OCL expressions is currently supported:

- All operations on primitive types such as boolean and integer are supported.
- Some operations on collections like `size()` and `isEmpty()` have also been implemented.
- `result` and the value of method parameters may be used in pre- and postconditions of methods.
- Access to attributes and associations of classes works as well as method calls including parameters during the constraint evaluation.
- As mentioned before, `@pre` expressions in postconditions are supported.
- Some other operation like `if then else`, `oclIsDefined()` and `oclAsType()` were also implemented.

Most operations on collections such as `forAll()` and `exists()` are missing. Nevertheless, the existing structure seems fairly extensible such that an addition of support for a broader set of expressions is supported reasonably well.

3.3.2 JavaOCL Template

This template is very similar to the Java template, but it uses the string variables which contain the code for OCL expressions. Therefore the code which is generated using this template evaluates OCL constraints when calling a method of the class. This is done in a way that is similar to the JUnit tests: A wrapper function which evaluates the constraints calls the actual method which is a private method of the class. As such, this is comparable to what the Dresden OCL code injector does, except that is does not inject constraint checking code into existing classes but generates stubs for the classes from the model.

4 Users' Guide

4.1 Prerequisites

Because the outcome of this thesis is fairly multilingual, there are several external dependencies which must be fulfilled before the complete tool-chain can be used:

- The `jtestdataaccessor` library depends on JUnit $\geq$ 4.0, AntLR $\geq$ 4.7.6, Java $\geq$ 1.5 and Apache Maven 2. The latter is optional but highly recommended since it simplifies the compilation and installation process after it has been set up.
- For the compilation and execution of the generated unit tests, the Dresden OCL Standard Library from the Dresden OCL Toolkit is needed. The toolkit is currently also required to combine the OCL specifications with the XMI file that contains the UML model of the system and finally serves as input for the code generator. To compile the toolkit, Java $\geq$ 1.5 and Apache Ant [5] $\geq$ 1.6.2 are necessary.
- Since the code generator is implemented in SML, a SML runtime or compiler is required such as SML/NJ [16], MLton [19] or Poly/ML [10].

4.2 Compilation and Installation

In the following sections, it is assumed that Maven is used as the build tool and that everything works the way it should. If the latter is not the case, there usually should be a file called `README` which gives more information about the installation and is presumably kept up to date in contrast to this thesis.

4.2.1 `jtestdataaccessor`

Since JUnit and AntLR are defined as dependencies in the project descriptor file for Maven, `jtestdataaccessor` can be built, tested and installed into the local Maven repository by executing `mvn install`. Besides the `install` goal, the provided POM file supports some other goals which are useful:

`package`: This goal builds a JAR file of the `jtestdataaccessor` in the `target/` directory. The generated archive does neither contain JUnit nor AntLR, so they need to be present on the class path in order to use the library.

`assembly:assembly`: By using this goal, a JAR file containing all the dependencies of the library can be created in the `target/` directory.

`site`: The Java API documentation of the library can be constructed with this goal and will be placed under `target/site/apidocs/`. In addition to the API documentation, this goal also creates a project website which contains reports from the unit tests and the cross referenced source code of the project.

To use the `jtestdataaccessor` library in a project which is built by Maven, a dependency such as the one in Listing 4.1 can be used in the `<dependencies>` section of the projects' POM file after installing the library into the local repository.

```
<dependency>
    <groupId>ch.ethz.infsec.jtestdataaccessor</groupId>
    <artifactId>jtestdataaccessor</artifactId>
    <version>1.0-SNAPSHOT</version>
    <scope>test</scope>
</dependency>
```

Listing 4.1: `jtestdataaccessor` dependency

4.2.2 Dresden OCL Toolkit

The automatically generated unit tests depend on the Dresden OCL Standard Library since they also verify OCL constraints. It is recommended to check out the current version of the Dresden OCL Toolkit 2.0 from CVS and to build it from source. Since the build process is based on Apache Ant, it should suffice to execute `ant jar.ocl20stdlib` in the top level directory of the CVS checkout which compiles the library and builds the JAR archive `ocl20stdlib.jar` in the `lib/internal/` subdirectory. This library can then be installed into the local Maven repository by using the following command:

```
mvn install:install-file -Dfile=lib/internal/ocl20stdlib.jar \
    -DgroupId=tudresden.ocl20 -DartifactId=stdlib \
    -Dversion=1.0-SNAPSHOT -Dpackaging=jar -DgeneratePom=true
```

Analogous to the `jtestdataaccessor`, the OCL Standard Library may then be used in a project by specifying a dependency like the one in Listing 4.2. Because the `<scope>` tag tells Maven to only use the library during the testing phase of the build, this line must be removed if the JavaOCL template mentioned in section 3.3.2 was used to generate the skeleton of the system.

```
<dependency>
    <groupId>tudresden.ocl20</groupId>
    <artifactId>stdlib</artifactId>
    <version>1.0-SNAPSHOT</version>
    <scope>test</scope>
</dependency>
```

Listing 4.2: Dresden OCL Standard Library dependency

4.2.3 Code Generator

The code generator can be compiled in different ways: A call to `make codegen` in the `su4sml/` directory will compile the code generator using SML/NJ and output a heap image file. If the interactive `sml` shell is used instead (still in the same directory), the code generator may be run after executing `CM.make("src/codegen/codegen.cm");`.

Alternatively, the MLton compiler builds a binary which is directly executable when called in the `su4sml/` directory with `mlton src/codegen/codegen.mlb`.

4.3 Usage

4.3.1 `jtestdataaccessor`

When using the code generator extension to generate the test suite, the `jtestdataaccessor` does not need to be used directly since all such code is generated automatically. This generated code may therefore serve as an example on how to use the `jtestdataaccessor`, together with the `ch.ethz.infsec.jtestdataaccessor.TestHelper` class which does most of the work involving access to test data.

4.3.2 Code Generator

The code generator is called with two arguments, the first one being the input XMI file and the latter one being the target language. During this thesis, three new target languages were developed:

`junit`: This target language generates a test suite containing constraint checking code and follows the pattern described in section 2.2.1. In addition to the Java classes it also creates stubs for the test data files.

`javaocl`: When using this target, Java source code stubs for the system will be generated which contain constraint checking code as described in section 3.3.2. This is the only difference to the `java` target which existed already before (that is, besides the improvements of the `java` target which were implemented in this thesis as well).

`pom`: Basic Project Object Model Files for Maven as mentioned in section 3.2.3 are created by this target. As they contain several default values, these files must be customised by the user even though they should work out of the box.

When using Maven as the build tool and if a POM file has been created as well, the generated code may now be compiled using `mvn compile` which will put the class files in a directory tree unter `target/classes/`. `mvn test` can be used to compile and execute the unit tests.

5 Conclusion and Future Work

This thesis has shown the design and implementation of the automatic generation of JUnit test-harnesses. The presented solution consists of three main parts:

1. The `jtestdataaccessor` library, which simplifies access to test data from within JUnit tests. This test data is stored in simple files with a special format that is mostly independent of the implementation language of the system.
2. An extension of the Generic Code Generator which generates test suites following the devised pattern of a generic test strategy. The generated test suites make use of the `jtestdataaccessor` to access test data which can therefore be separated from the actual unit tests.
3. The verification of interface specifications during the execution of unit tests by another extension of the code generator. This extension is able to translate OCL constraints to Java code that uses the Dresden OCL Toolkit to evaluate those constraints at runtime.

During the work on this thesis, some other, mostly minor extensions of the code generator and the `su4sml` UML repository were implemented as needed. General experiences have shown that the existing architecture is relatively easy to extend and use.

5.1 Future Development

The generated code for the verification of interface specifications currently does not implement behavioural subtyping which basically consists of three properties:

1. Invariants of subtypes can only be stronger than those of its parents.
2. Preconditions of overriding methods of subtypes may be weaker than those of its parents.
3. Postconditions of overriding methods of subtypes can only be stronger than those of its parents.

An extension in the case of invariants and postconditions seems straightforward, but unfortunately more difficult for preconditions.

Since complete support for the translation of all OCL constraints to Java code was not feasible, this is an extension which is left for future work. It nevertheless seems that the implemented subset allows one to express many meaningful constraints.

Some outcomes of this thesis still have some minor dependencies on the chosen implementation language. If the architecture is ported to other languages, these should be removed such that the test data files can be used for systems implemented in different languages without the need for adaptations.

A Test Data File Grammar

```
TestData = { Section };

(* A section consists of a global scope and any number of test cases. *)
Section = '[' IdentString ']' { ( GlobalAssignment | TestCase ) };

(* Keys and allowed values in the global scope *)
GlobalAssignment    = ResultTypeDef | InputTypesDef
                    | SetupDef | TearDownDef;
ResultTypeDef       = 'resulttype =' Type ';';
InputTypesDef       = 'inputtypes =' Type { ',' Type } ';';
SetupDef            = 'setup =' Function ';';
TearDownDef         = 'teardown =' Function ';';

(* Keys and allowed values in the test case scope *)
TestCase            = '{' TestCaseAssignment { TestCaseAssignment } '}';
TestCaseAssignment  = InputDef | ResultDef | CheckerDef | CommentDef;
InputDef            = 'input =' Value { ',' Value } ';';
ResultDef           = 'result =' ( Value | ClassName ) ';';
CheckerDef          = 'checker =' ( Function | DefaultChecker ) ';';
CommentDef          = 'comment =' '"' String '"' ';';

(* Default checkers which map directly to JUnit assertions. *)
DefaultChecker = 'EQUALS' | 'NOTNULL' | 'NULL' | 'NOTSAME' | 'SAME'
               | 'FAIL' | 'NOTEQUAL';
Value          = Function | '"' String '"' | '\'' Character '\'' | Integer
               | Real | Boolean;
Function       = IdentString '()';
Type           = ClassName [ '[]' ];
IdentString    = AlphCharacter { ( AlphCharacter | Digit ) };

(* Terminals *)
String        = STRING;
Character     = CHARACTER;
Integer       = INTEGER;
Real          = REAL;
Boolean       = BOOLEAN;
ClassName     = CLASSNAME;
AlphCharacter = ALPHABETICCHARACTER;
Digit         = DIGIT;
```

Listing A.1: EBNF grammar for the test data file

The grammar in Listing A.1 does not include comments which are nevertheless supported by the actual implementation: Comments are usually discarded by the lexer and therefore they do neither need to be supported by the parser, nor are they part of the grammar. Some of the terminals are not common but the meaning should be clear from their name, whereas `CLASSNAME` could be an exception: Since the grammar for class names depends on the effective programming language, this is not specified in the grammar for test data files. In C# for example, name space names usually start with a capital letter whereas in Java package names are typically all lowercase.

List of Figures

List of Listings

Bibliography

[1] Software Engineering Group at Technische Universität Dresden.
Dresden OCL toolkit.
`http://dresden-ocl.sourceforge.net/`.

[2] Kent Beck et al.
SUnit — the mother of all unit testing frameworks.
`http://sunit.sourceforge.net/`.

[3] Yoonsik Cheon and Gary T. Leavens.
The JML and JUnit way of unit testing and its implementation.
Technical report, The University of Texas at El Paso Iowa State University, 2004.

[4] Raphael Eidenbenz.
A generic code generator for su4sml, 2006.
Semester Thesis.

[5] Apache Software Foundation.
Apache Ant.
`http://ant.apache.org/`.

[6] The Eclipse Foundation.
Eclipse.
`http://www.eclipse.org/`.

[7] Erich Gamma and Kent Beck.
JUnit.
`http://junit.sourceforge.net/`.

[8] Steffen Gemkow and Andreas Braig others.
JUnitDoclet.
`http://www.junitdoclet.org/`.

[9] Gary T. Leavens et al.
JML - the Java modeling language.
`http://www.jmlspecs.org/`.

[10] David Matthews.
Poly/ML.
`http://www.polyml.org/`.

[11] Bertrand Meyer.
Building bug-free o-o software: An introduction to Design by Contract.
`http://archive.eiffel.com/doc/manuals/technology/contract/`.

[12] Object Management Group (OMG).
Object constraint language (OCL).
`http://www.omg.org/technology/documents/modeling_spec_catalog.htm#OCL`.

[13] Terence Parr.
The ANTLR parser generator.
`http://www.antlr.org/`.

[14] Adam Skogman et al.
DDSteps.
`http://www.ddsteps.org/`.

[15] JTestCase Project Team.
JTestCase.
`http://jtestcase.sourceforge.net/`.

[16] The SML/NJ Team.
Standard ML of New Jersey.
`http://www.smlnj.org/`.

[17] Linus Tolke et al.
ArgoUML modeling tool.
`http://www.argouml.org/`.

[18] Jason van Zyl et al.
Apache Maven.
`http://maven.apache.org/`.

[19] Stephen Weeks, Matthew Fluet, Henry Cejtin, and Suresh Jagannathan.
MLton Standard ML compiler.
`http://mlton.org/`.

[20] Wikipedia.
List of unit testing frameworks, 2007.
`http://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=112287570`.

[21] Wikipedia.
xUnit, 2007.
`http://en.wikipedia.org/w/index.php?title=XUnit&oldid=106304974`.