Triops - Black Box Compiler Testing

Dominique Schneider

Semester Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

SS 06 / WS 07

Supervised by:
Dipl.-Ing. Werner M. Dietl
Prof. Dr. Peter Müller

Abstract

This paper describes the specification and implementation of a management tool for black box compiler testing called Triops. Triops itself is implemented in Java and intended to be used for Java compilers. Triops is a enhancement of the current used test procedure for the MultiJava compiler. This implementation is based on Makefiles which handle the testcase selection, the invocation of the compiler with the testcase and finally the comparison of the output data. Besides some redundant code in the Makefiles this implementation is coupled very closely to the MultiJava compiler and doesn't allow simple reuse of testcases for other compilers. Furthermore the editing of the Makefiles which are distributed over several folders is a cumbersome and error-prone task. This paper presents a solution which allows the reuse of testcases for multiple compilers and the administration of the corresponding output data. It clearly separates the testcases from the configuration data and the involved external applications (i.e. compilers) which improves maintenance and understandability. As a side-effect of this clear separation it gets very simple to extend Triops with new ways how generated testcase output could be compared against the expected outcome. As a case study the current existing Universe Type System testcases of MultiJava will be ported to Triops.

Contents

Chapter 1

Introduction

1.1 Overview

Chapter 1 presents a short analysis of the implemented testing procedure of MultiJava and points out some weaknesses. Afterwards an informal introduction into the new basic test procedure will be given. Based on this informal specification the next chapter 2 presents the detailed design and the technical implementation of the specified requirements. The closing chapter 3 evaluates Triops based on the migration of the Universe Type System from MultiJava to Triops and outlines possible future work.

1.2 Current state

One way to do functional testing on a compiler is to compile a code snippet and compare the generated binary with an expected binary. If the binaries are equal the test is declared successful and failed otherwise. This test method is called black box testing and can be described as:

> Black box testing takes an external perspective of the test object to derive testcases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure.[wik07a]

Whereas the 'valid input' is the code snippet and the 'correct output' corresponds to the expected binary or emitted warnings during compilation. In general the valid output doesn't have to be a binary but can be the runtime output of the compiled code snippet or maybe the decompiled class file.
In MultiJava the test administration is mainly done with Makefiles which contain targets for compiling, decompiling and running testcases among other things. These Makefiles are found in every folder with testcases and call each other recursively i.e. a Makefile calls other Makefiles in its subfolders. The Makefiles contain the invocation calls to the applications used for compilation, decompilation and so on. The main drawback of solution is the general restriction of flexibility because each small adaptation (like using a different application for compilation) requires adapting one or more Makefiles and maybe the introduction of new targets e.g. using `javac` compiler instead of the MultiJava compiler requires the updating of each call to the MultiJava compiler in every Makefile. Some Makefiles are quite large and modification can be time consuming hence it is desirable to have a greater flexibility in switching the involved external applications.
Certain recurring tasks are defined redundantly in multiple Makefiles which leads to duplicated code and complicates maintainability. Often used code should be extracted and stored in a single file to enforce code reuse.
Currently the testcases are stored together in the same folder with their generated and expected output. This mixture makes simple tasks like deleting all generated output to a painful job

and requires another Makefile target which must be maintained. Furthermore storing all data in the same folder is suboptimal if one wants to work with several applications, because sometimes it's crucial which application produced a certain output and one has to ensure that the output is distinguishable e.g. if different compilers produce different binaries of the same testcase they should still be distinguishable. The output data should be stored in different folders with the producing application somehow coded into the folder name.
As shown in the previous examples the administration of tests and their related data can be cumbersome and it's desirable to cut down the effort on writing testcases and automate the rest. An application is needed that handles the test execution and the comparison of the output, furthermore it should be possible to configure the program in a few configuration files. The following sections will define the requirements of a testing application which avoids the shown limitations and annoyances.

1.3 Requirements

Roughly summarised Triops works like this: Tester transform the testcase data from one representation into another (e.g. compile source into a binary, decompile a binary into a byte code file or execute a binary) and compare the produced output against the expected output. To perform the transformations external application will be invoked (e.g. compilers, decompilers, runtime environments) by the tester but which applications should be used and how they have to be called is declared in a component. A component will configure a tester and use it to perform a transformation. So basically the components represent an external application and can be applied to a testcase but in fact they delegate the testcase together with some configuration data to a tester which will do the real work.
If one wants to execute a testcase one has to apply some components in a specified order to the testcase (e.g. compile first, run later). Setups define such a sequence of components and guarantee the availability of the output data of a early executed component as input data for a later component. Normally one needs several setups to execute all kinds of testcases for a compiler. An environment represents a set of setups for one compiler and allows the sequential execution.
The testcases themselves are common `.java` files with some additional tags at the beginning. Tags are used to select a testcase for execution and otherwise to determine if a component is allowed to execute a testcase.
The basic concept of Triops is that the user first selects the testcases he wants to execute (the test set). This can be done with test suites, which build a test set by including a testcase based on some simple rules. After assembling the test set the user chooses the compiler (i.e. the environment) which should be executed on the test set. Now all setups of the environment will sequentially execute their components on each testcase, produce output data and compare it against the expected outcome. The output will be compared and lines which differ will be shown to the user.

Testcases A testcase is a file containing a code snippet e.g. a common `.java` file containing a class definition. These testcases produce data which can be used for comparison (e.g. compilation errors, binary, runtime output).

Tags It should be possible to place tags in a testcase. A tag states a property of a testcase and can have different meanings. A tag can be used to include a testcase in the test set (the set of all tests which will be executed) or it can be used to determine if a certain application can process this testcase e.g. if one works with the JML compiler one could define that only testcases with the tag `JML` will be processed.

Folder structure All testcases are organised in a folder structure called *source structure* whose top folder is called the *source structure root*. All output produced by testers should be stored in another folder structure whose root is on the same level as the *source structure root*. This structure is called *generated structure* and is quite similar to the *source structure* so that it is easy to locate the corresponding output of a certain testcase in the *generated structure*. A third structure (whose root is again a sibling of the *source structure root*) contains the expected output of the testcases and is therefore called *expected structure*.

Test set The test set contains all testcases nominated for execution. A test set should be buildable by selecting all testcases with a certain tag or accordingly to a test suite. A test suite is defined in a folder and allows the user to add testcase in this folder to the test set if it matches certain criteria. The test suite provides three methods for adding testcases:

- add all testcases with a certain tag combination i.e. a tag must have tag `X` and tag `Y` but not tag `Z`
- add testcases whose filenames match a pattern
- add all testcases in a subfolder based on a test suite defined in this subfolder

The first two methods can only add files in the same folder where the test suite is defined in whereas the third method can only refer to test suites defined in a direct subfolder.

Testers A tester processes a testcase and produces data for comparison. There should exist one tester for each kind of comparison i.e. one tester should perform the compilation and the comparison of the compile related warnings, another tester should run the test and compare the runtime output. The user should be able to create new testers and add them to the existing ones in a easy manner.

Components A tester only defines what and how output files should be generated but not by whom e.g. the binaries of a testcase could be produced by the standard Java compiler or by the JML compiler. A component is a linkage between a tester and an external application which actually performs the task.
A component should process a testcase in a special way upon detection of a certain tag. Let's assume a testcase should test if the standard Java compiler with 1.3 compliance emits an error if it's invoked with a testcase containing an `assert` statement (which was introduced in 1.4). But because the other testcases may require post 1.3 compliance it is necessary that only this testcase is treated exceptional. In this example an extra parameter could be passed on to the Java compiler to set the compliance mode to 1.3 if a predefined tag is found.

Environment and Setups Instead of talking about a component one should rather talk about test setups, because a setup contains a sequence of components which are applied to a testcase. A possible setup could be a first component which produces binaries and afterwards a second which executes them e.g. for Java one would use `javac` (compiler) and `java` (runtime) but for a setup which should only perform some static analysis a single `check` component could be sufficient. Because of the dependencies between the components the order of the sequence should be constructed accordingly i.e. a component which performs the compilation should precede a component which requires the binary.
Sometimes setups are rather similar e.g. two setups which use the same compiler but different decompilers. These coherent setups should be grouped together to an environment so that all setups of an environment can be easily executed at once.

Configuration Sometimes it doesn't make sense to execute all testers on a testcase e.g. a testcase that provokes a compilation error won't generate a binary that could be decompiled. Therefore a testcase should possess some additional tags to declare the testers that can handle this testcase. This way a testcase which would fail during compilation could be excluded from decompilation or runtime tests.

Comparison The whole output is stored in files and will be compared against the expected output. The comparison is done line-wise and by computing the longest common subsequence of lines[wik07c]. The difference of the two files is printed similar to the program `diff` [wik07b] i.e. only lines which are different will be shown to the user whereas equal sections will be suppressed.

1.4 Naming Triops

Triops or Triopas, a three-eyed character of the Greek mythology, is the son of Poseidon, the god of the sea. Three eyes seem to see more details than just two which was considered helpful in a context where one mainly compares output data. Furthermore the three eyes could represent the three built-in testers present in Triops. Apparently the descendants of Poseidon seemed to have some issues with eyes because the half brother of Triopas was the cyclops Polyphemus. Luckily both belong to the same family so at least the average number of eyes per person stays on a normal level.

Chapter 2

Detailed Design

2.1 Testcases

A testcase is a `.java` file containing a default Java class declaration. In fact Triops doesn't care about the validity of the testcase because it will forward the unmodified testcase to the processing external application. Triops will only parse the first two lines of each testcase, hence tags at other locations than the first line will be ignored (as we will see later the second line is reserved for something else). So if the testcase is not valid Java or it is not fully understood by the processing compiler then it will likely emit an error message as output.
Limitations: Inner classes will be compiled into separate `.class` files and Triops won't be able to anticipate the generated files without previously inspecting the testcase. Triops assumes that for each `.java` file a homonymous `.class` will be created. Although the `.class` files of the inner classes will be compiled into the same folder as the main class and hence are available for runtime tests. However it is not possible to do explicit operations with an inner class because Triops does not know about the existence of such a file. The functionality of testcases with inner classes is not guaranteed.
The use of the package declarations is problematic because the package name will be mapped into a folder structure upon compilation. Triops would have to parse the package name from a testcase to discover the location of the compiled file. Triops assumes that all testcases are declared in the default package and therefore package declarations mustn't be used.
A `.java` file can contain sequentially declared classes (we will call them secondary classes). Secondary classes do not have the same name as the `.java` files they are declared in and are compiled into separate `.class` files. The filename of a compiled inner class will contain the class name of the outer class in the filename of the `.class` which guarantees the uniqueness of the filename. This is not the case with secondary classes which may be a problem in a scenario where multiple `.java` files contain secondary classes with the same name. If these classes are compiled each homonymous secondary class will be compiled into a `.class` file with the same name (thereby overwriting previously generated classes). If executed some classes will misleadingly use the last generated version of this class which may lead to strange test behaviour. As a naming convention secondary classes should always contain the name of the main class in their own class name to prevent these name clashes.

2.2 Tags

Tags can be used to assign some properties to a testcase and can be freely chosen by the user. These properties are used to build a test set, to separate the output of different setups and to handle a testcase in a special way if it has a certain tag.
A tag is a case-insensitive alphanumeric string (i.e. contains numbers 0-9, letters a-z and A-Z) plus the special character "`_`". Tags must be declared in the first line of a testcase. The line must start

with the prefix `//Tags:` and all following tags must be separated by whitespace. If there are no tags declared after the prefix or the whole line is missing, the testcase has no tags.

2.3 Testers

As stated in section 1.3 a tester is responsible for executing testcases and compare their output against the expected outcome. A tester will perform the invocation of the external application, may rename already existing files with the same names and knows which files should be used for comparison. Three common ways of comparing in- and output data are:

- compile a testcase and use the console output generated by the compiler for comparison (the generated binary is not observed i.e. a static analysis tool could be used instead of a compiler)
- decompile a binary and use the generated source for comparison
- run a binary and use the console output for comparison

Each of these comparisons is implemented by one tester. A tester can reuse the functionality of another tester i.e. a tester that needs a binary as input (e.g. for decompilation) can be executed after a tester which produces a binary. The reuse of testers simplifies the introduction of new testers. If one wants to introduce a new comparison which needs a binary as input then the new tester only has to implement the new comparison and can presume the existence of a binary. Although it's clear that the user somehow has to assert that the binary producing tester is executed before his new tester. We will see later that testers can be chained together to assert an order of execution. According to the common ways of comparing data three testers are built-in and distributed with Triops (with their name in bold):

check: the test file is statically analysed and may produce a binary. The console output will be stored in a file with the suffix `-warning` whereas output on the error stream will be placed in a file with the ending `-error`. The emitted console output or warning streams will be used for comparison.

code: the binary is decompiled and compared. The console output will be stored in a file with the suffix `-code` whereas output on the error stream will be placed in a file with the ending `-error`. The prior execution of `check` is necessary to get a binary although there is no check which asserts this requirement.

runtime: the binary will be executed and its console output will be used for comparison. The console output will be stored in a file with the suffix `-out` whereas output on the error stream will be placed in a file with the ending `-error`. The prior execution of `check` is necessary.

In section 1.3 the requirements stated the necessity of marking testcases to be processable by a certain tester. For this reason a testcase has a special line together with the tag line introduced in 2.2. Whereas the tags of the tag line are used to select testcases for execution, separate output, etc. the new line is simply to tell a processing tester if it actually should process this testcase. The line has to start with the prefix `//Testers:` followed by the allowed tester's name separated by whitespace. The testers line has to be the second line of the file or the first if no tag line exists. If no testers line is present the testcase is not processable by any tester.

2.4 Components

2.4.1 Overview

A component contains the information about an external application which is needed by a tester to successfully run it on a testcase. It has an unique name and a type which is equal to the name of the underlying tester. Because not all external applications have the same invocation schema (e.g. order and names of parameters, etc.) the user can provide an executable (`command`) which is invoked by the tester with some parameters which then can be forwarded to the real external application's binary. This way the executable can reorder or omit the parameter to fit the invocation schema of the application. Some example parameters delivered by a tester are the classpath, the target file (or directory), the output path and some further additional parameters which vary depending on the tag that is currently processed.

As stated in the requirements one should be able to list all tags a component can process and even customise the external application's behavior if a certain tag has been found. A component has a list with known tags which it will compare against the tags a testcase possesses. It will call the specified `command` file once for each tag of the intersection of these two tag sets. Each tag has some corresponding flags which will be forwarded to the external application via the tester. A component skips a testcase if it doesn't know at least one of the testcase's tags (i.e. the intersection is the empty set).
The declaration of the components and the tags they know is done in one XML file called `config.xml`. There is no default `config.xml` distributed with Triops as the content is based on the tags and components chosen by the user. A config file should be seen as part of the user-defined testcases, because besides writing and tagging his own testcases a user furthermore creates this file to define the specific applications to use for his testcases. As we will see later the user has to specify the location of the testcases and the config file Triops should use at runtime. For the moment it is important to note that only one config file can be used at once and that referencing of components declared in other config files is not possible.

2.4.2 XML

As stated in the requirements one should be able to list all tags a component can process and even customise the external application's behavior if a certain tag has been found. Every `component` can have multiple `tag` child entities which define a processable tag with a `name` and some optional `flags` to pass to the external application. This `flags` attribute is the only optional attribute of all entities in this file.
The setups always have to be declared inside an environment. An environment has the single unique attribute `name`. A setup must have a name that is unique in the environment's scope and must contain a sequence of `items` whose content corresponds to a component's `name`. The components will be executed in the same order as the `items` are declared, hence a `runtime` component should be listed after a `check` component because it requires a binary as input. Every tester can only be used once per setup i.e. there can not be multiple `check` components in one setup.
The components attribute `command` can contain an absolute or relative path or may reference to an executable whose parent directory is in the environment variable `PATH`. All `name` attributes are alphanumeric strings plus the special chars "`-`" and "`.`". All existing components are defined at the beginning of `config.xml` followed by the available setups and environments (see section 1.3).

2.4.3 Example

```
<config>
  <!-- check components -->
  <component type="check" name="jml" command="jml.sh">
    <tag name="jml" flags="-p" />
  </component>
  <component type="check" name="esc" command="esc.sh" >
    <tag name="esc" flags="-w" />
  </component>
  <!-- code components -->
  <component type="code" name="javap" command="javap.sh">
    <tag name="jml" />
  </component>
  <!-- runtime components -->
  <component type="runtime" name="java1.4" command="java-1.4.sh">
    <tag name="jml" />
  </component>
  <component type="runtime" name="java1.5" command="java-1.5.sh">
    <tag name="jml" />
  </component>
  <!-- environments -->
  <environment name="JML">
    <setup name="java1.4">
      <item>jml</item>
      <item>javap</item>
      <item>java1.4</item>
    </setup>
    <setup name="java1.5">
      <item>jml</item>
      <item>javap</item>
      <item>java1.5</item>
    </setup>
  </environment>
  <environment name="ESC">
    <setup name="default">
      <item>esc</item>
    </setup>
  </environment>
</config>
```

2.5 Folder structure

2.5.1 Overview

This section explains the application's folder structure and the testcase organisation. At startup Triops will expect the testcase location as parameter. Consequently the user can place the testcases in a folder of his choice and does not have to put them at the same location as Triops itself.
The application's folder structure is rather simple. It contains `Triops.jar` with all Java related data (classes, packages) which is nothing special per se. Only the subfolder `testers` should be of some interest because it contains the built-in and user-defined testers. The distributed Triops structure looks like this (libraries are placed in subfolder `lib` but are omitted for simplicity):

```
triops
|   Triops.jar
|
+---lib
\---testers
        AbstractTester.class
        CheckTester.class
        CodeTester.class
        RuntimeTester.class
        testers.xml
```

According to section 1.3 a testcase structure consists of three parts, firstly the testcases themselves, secondly the output produced upon execution of the testcases and thirdly the output which is expected to be produced. The three parts are represented by three almost identical folder trees which all start in the root folder of the testcase structure. For clarity reasons the three prefixes *source* (if it has something to do with the testcase), *generated* and *expected* (if it's related to the testcase's generated or its expected output) will be used from now on. The user just needs to write the testcases and provide the expected output and Triops will execute the testcases, generate the

output and compare it against the expected output. For each testcase some corresponding files may exist in the expected tree and the generated files must match the expected ones in amount and content. If the user only has his source files and is sure that the output will be correct Triops can store the produced output as expected output. Iff a *source* folder is placed directly into the testcase root folder it will get cloned into a generated and expected structure. In general the user could create more folders in the testcase structure root and they would all get cloned i.e. each direct source subfolder in the testcase root will get cloned to an *expected* and *generated* structure, however all folder trees start in the testcase structure root. It is not instantly clear what would happen if a testcase is placed directly into the testcase root folder because there is no folder which could be cloned and be used for storing output. This issue is solved by prohibiting the placement of testcases directly into the testcase root folder. Testcases placed in the testcase structure root will be ignored.
The mapping of the source structure to the output structure (i.e. *generated* or *expected*) will be explained below. While reading the textual description one should have a look at the example (section 2.5.2) every now and then to reduce confusion.
Basically all folders of the *source* structure will receive the suffix `-generated` and some additional folders will be introduced. Because only output data will be stored the original testcases won't be copied into the new structure . The difference between the *generated* and *expected* structure is simply the usage of the string `-expected` instead of `-generated` in the folder names. If the user wants to locate the produced output of a certain testcase, he simply has to add the suffix `-expected` to each folder name below the testcase root. From there the output is split up in additional folders whose names are based on the selected environment, setup, component and tag and are created whenever the original *source* folder (i.e. the one without the suffix) contained some testcases. If the testcases were located in further subfolder of the *source* folder the algorithm will be applied recursively to the subfolders. So in the end each *source* folder which directly contained testcases will have a corresponding clone with newly introduced subfolders. The reason for introducing these further folder levels is to prevent mixture of output. But let's analyze each additional layer for itself:
If components rely on output generated by preceding components it must be assured that they don't access output data of other setups e.g. two setups use different `check` components which produce different binaries and the following `code` component must use the correct binary as input. To guarantee this separation the data flow of each environment-setup combination must be isolated in different subfolder whose name is the environment name and the setup name separated by a hyphen.
The next layer is based on the component's type and only helps to quickly distinguish the output generated by different testers. The separation of the testers output could be achieved by adding suffixes to the file but gets quite clearer with separate folders.
Section 2.4 mentioned that a component executes a testcase for each of the testcase's known tags and furthermore that upon detection of certain tags further tags could be passed to the external application. Due to this requirement the same testcase executed for two different tags doesn't need to produce the same output even with the same environment, setup and component.
The files found in the tag folder are based on the tester's specification (see section 2.3).

2.5.2 Example

A user writes two testcases `AbstractWithBody.java` and `AssertStmt.java` and for some reason places another testcase `AbsIncomplete.java` in a subfolder `myTestsSubfolder`. His `config.xml` defines an environment `JML` with only one setup `default` which chains a `check`, a `code` and a `runtime` component (in this order). All components can process the tags `jml_1` and `jml_2`. The user assumes all his testcases produce the correct output and lets Triops generate the expected folder structure. The file `suites.xml` is used to select the testcases which will be processed. However, these files are listed for completeness and can be ignored for this example.

```
testcases
|   config.xml
|   suites.xml
|
+---myTests
|   |   AbstractWithBody.java [Tags: jml_1]
|   |   AssertStmt.java [Tags: jml_1]
|   |   suites.xml
|   |
|   \---myTestsSubfolder
|           AbsIncomplete.java [Tags: jml_1,jml_2]
|           suites.xml
|
\---myTests-expected
    +---JML-default
    |   +---check
    |   |   \---jml_1
    |   |           AbstractWithBody.java-error
    |   |           AssertStmt.class
    |   |           AssertStmt.java-error
    |   |
    |   +---code
    |   |   \---jml_1
    |   |           AssertStmt.java-code
    |   |
    |   \---runtime
    |       \---jml_1
    |               AssertStmt.java-out
    |
    \---myTestsSubfolder-expected
        \---JML-default
            \---check
                +---jml_1
                |       AbsIncomplete.java-error
                |
                \---jml_2
                        AbsIncomplete.java-error
```

If we look at the folder's hierarchy we notice that the whole folder structure of `myTests` has been cloned and every previously existing folder received the suffix `-expected` and has been filled with the expected output of the testcases.
In this example the testcase `AbstractWithBody.java` obviously should produce an compilation error hence only an error file has been created. `AssertStmt.java` also produced some error output but nonetheless produced a binary (actually the emitted warning was due to the used keyword `assert` which won't fail compilation). Because compilation succeeded the `code` and `runtime` components were able to decompile and run the binary and produce the corresponding files `AssertStmt.java-code` and `AssertStmt.java-out`.
The subfolder `myTestsSubfolder` has been mapped to `myTestsSubfolder-expected` and received the same additional subfolder structure as his father. The testcase `AbsIncomplete.java` had two tags and therefore two folders were created in the tag layer and the testcase produced in both cases an error.
We now have two folder trees which start in the testcase structure root and if we would let Triops execute the testcases and compare their output data against the expected output a third tree would be created with the same structure as the expected tree (although the suffix `-generated` is used instead of `-expected`).

2.6 Extending testers

2.6.1 Overview

As presented in chapter 2.3, Triops has built-in support for the common testers but the user should be able to add new testers. Each new tester must extend the abstract class `AbstractTester` which provides helper functions to invoke an external process, perform a comparison, etc. A subclass has to implement the abstract method `build` which should generate all output and mark it for comparison. After calling `build` the `AbstractTester` will compare all marked files. The knowledge about available testers is stored in `testers.xml` which can be found in the subfolder `testers` of the application's directory (see the folder structure of Triops in section 2.5). To use the new tester, the class must be dropped in the folder `testers` and a new line must be added to `testers.xml`.

2.6.2 XML

Each `tester` possesses a unique `name` and is associated with a `class` which implements the new functionality. A component which is associated with a tester will have the tester's `name` as type.

2.6.3 Example

A possible `testers.xml` could look like this:

```
<testers>
  <tester name="runtime" class="RuntimeTester" />
  <tester name="code" class="CodeTester" />
  <tester name="check" class="CheckTester" />
  <tester name="myTest" class="MyTestTester" />
</testers>
```

2.7 Test suites

2.7.1 Overview

Sometimes one doesn't want to execute all existing testcases but only an assembled subset. Possible test suites could select all testcases with a combination of certain existent and absent tags or testcases that are thematically cohesive (e.g. all tests that have been introduced after fixing a bug). A test suite builds a test set by adding testcases which match certain criteria. These suites are defined in the `suites.xml` files which attracted attention in section 2.5 and can be found in every folder of the *source* part in the testcase structure.
Before introducing the several ways a test suite can add testcases to a test sets, one should know that each `suites.xml` only can include testcases which are in the same directory as itself. Besides including testcases directly a test suite of a subfolder can be invoked and the resulting test set can be merged in the current one.
As described in 2.5 testcases which are placed directly in the testcase root folder won't be processed. Consequently a test suite defined in the testcase root folder won't be able to add testcases from this folder to the test set and only calling other test suites makes sense. This top level suite can be used as a switch to call suites in lower folders.
The various ways to build a test set are explained together with the corresponding XML elements which can be used:

<file filter="*.java" /> this element adds all testcases from the current directory whose file name matches this filter. To construct a filter expression one can use the wildcard symbol `*` which stands for a sequence of chars with random length e.g. filename `nestedClass.java` would match the filter `nested*` but not `nested*2.java`.

<tags>tag1 !tag2</tags> this element adds all testcases from the current directory which contain each of the blank-separated tags in the current folder. One can model tags that must be absent by adding a preceding exclamation mark. The example would build a set with all testcases that own `tag1` but not `tag2`.

<subfolder name="dir2" suite="suite3" /> this element will add the result set of the suite `suite3` in the current directory's subfolder `dir2`.

If a suite has several enumerated elements the resulting test set will be the union of all partial sets. Multiple suites can be declared in one XML file and each has to be identified by an unique `name`.

2.7.2 Example

An example test suite would look like this:

```
<testsuites>
  <suite name="expensive">
    <file filter="Factorial.java" />
    <file filter="*-power.java" />
    <subfolder name="subfolder2" suiteName="expensive" />
    <tags>jml !multijava</tags>
  </suite>
</testsuites>
```

This test suite is named `expensive` and should group all testcases which use a lot of CPU cycles. The test suite contains the file `Factorial.java`, all files that end with `-power.java`, all files that contain the tag `jml` but not `multijava` and the set that is returned by the test suite `expensive` in `subfolder2`.

2.7.3 Implicit defined suites

A quite common test set will contain all testcases with a certain tag. A suite to collect all testcases with the tag `jml` would look like this (if there are N subfolders):

```
<suite name="jml">
  <tags>jml</tags>
  <subfolder name="subfolder1" suiteName="jml" />
  ...
  <subfolder name="subfolderN" suiteName="jml" />
</suite>
```

This suite must be declared in every folder and a recursive call for each subfolder has to be added which is quite an overhead for such a simple task. To reduce the amount of work one can just assume that suites which search for a certain tag are implicitly defined i.e. if the user provides a suite name which is not defined in a `suite.xml` it is interpreted as a tag and all testcases from the current folder and its subfolders are added without looking at the subfolder's suites. Instead of writing the example suite above the user can just invoke the imaginary suite `jml` in that subfolder and it will collect all desired testcases.

2.8 Usage

This section explains how to start Triops and explains the different parameters. It is necessary to tell Triops where the testcase structure root is and which `config.xml` should be used. Sometimes these two things do not change very often and it's also possible to set two environment

variables. `$TRIOPS_TESTROOT` points to the testcase structure root and `$TRIOPS_CONFIG` points to the `config.xml` file to use.
An example scenario could look like this: The user goes into the directory that contains the testcases he wants to execute and invokes Triops. It is not necessary that the user is in the testcase structure root folder. To successfully run Triops at least following parameters must be provided: the testcase structure root and the config file to use. Although Triops should know which environment and setup to execute it is not mandatory. If no environment is provided all existing environments with all their setups are executed. If the user only provides an environment without a setup all setups of the provided environment will be executed. The default mode of Triops is to generate testcase output and compare it against the expected output but the user can enable the "generating" mode to store the testcase output as expected output (see section 2.5) by providing the flag `-gX` which would overwrite already existing expected output or `-g` which renames the existing output. The renaming will add an integer number as suffix to all corresponding files. If already renamed files exist the integer will be incremented until a free number is found.

The program can be started as follows:

```
java ch.ethz.sct.Triops [options]
```

whereas the `options` are:

-verbose prints more details about what Triops does

-g this flag enables the generation of the expected output files and disables the comparison. This is helpful to generate the expected output after writing a new testcase. Existing output files are renamed. This flag is mutually exclusive with `-gX`.

-gX has the same functionality as `-g` but overwrites existing output files instead of renaming them. This flag is mutually exclusive with `-g`.

-config path the absolute or relative path to the `config.xml` that should be used. If this argument is omitted the environment variable `$TRIOPS_CONFIG` is used.

-testroot path the absolute or relative path to the root directory of the testcase structure. If this argument is omitted the environment variable `$TRIOPS_TESTROOT` is used.

-env environment:setup the environment and setup that should be used. All valid environments and setups are listed in the provided `config.xml`. Omitting the semi-colon and the name of a setup will execute all setups of this environment. If no parameter is passed the test is run for every environment and every setup defined in `config.xml`. A setup cannot be provided without environment.

-s suitename the name of a test suite to build the test set. This suite must be described in the `suites.xml` of the working directory. This flag is mutually exclusive with `-f`

-f filename the full name of a testcase to process. The testcase must be located in the working directory. This flag is mutually exclusive with `-s`

-help shows this info list about the options

Remember the fact that some suites are defined implicitly (see section 2.7). To select all testcases with the tag `X` one can use the parameter `-s X`.

2.9 Implementation

2.9.1 Overview

The main work is done by a few core classes which are used to iterate over all testcases and process them with the right tester. The core classes are visible in figure 2.1. Besides this main iteration a

few helper classes are necessary to prepare the main iteration and to perform the comparison.
The core of Triops is defined in the package `ch.ethz.sct.triops` whereas all testers are situated in the package `testers`. The reason for this separation is the folder structure of Triops which makes adding new testers as simple as possible. As we have seen in the section 2.6 the user can drop his new testers in the subfolder `testers` and they will be loaded by Triops. Accessing the folder `testers` from the core classes inside the `Triops.jar` is equal to accessing the package `testers` from the package `ch.ethz.sct.triops`.
Triops is structured in the following packages:

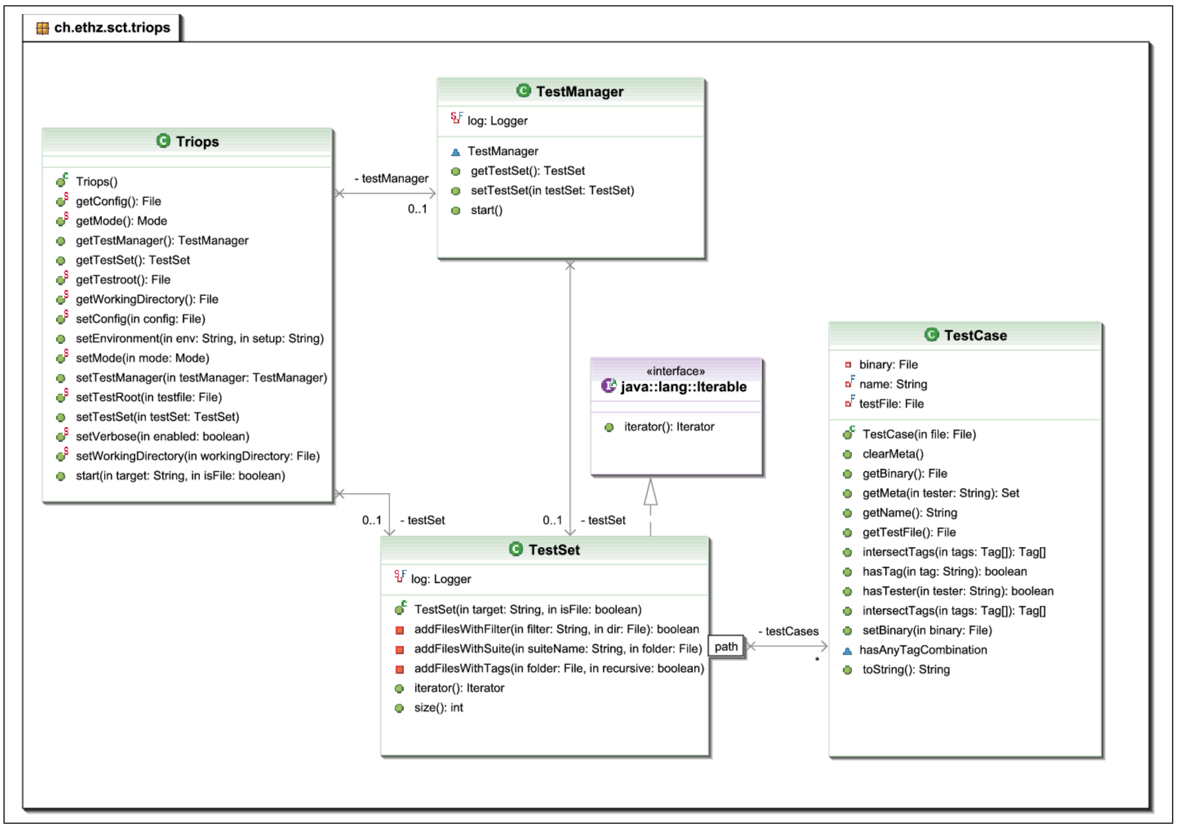
ch.ethz.sct.triops
Triops
Triops()
getConfig(): File
getMode(): Mode
getTestManager(): TestManager
getTestSet(): TestSet
getTestroot(): File
getWorkingDirectory(): File
setConfig(in config: File)
setEnvironment(in env: String, in setup: String)
setMode(in mode: Mode)
setTestManager(in testManager: TestManager)
setTestRoot(in testfile: File)
setTestSet(in testSet: TestSet)
setVerbose(in enabled: boolean)
setWorkingDirectory(in workingDirectory: File)
start(in target: String, in isFile: boolean)
- testManager
0..1
TestManager
log: Logger
TestManager
getTestSet(): TestSet
setTestSet(in testSet: TestSet)
start()
«interface»
java::lang::Iterable
iterator(): Iterator
TestCase
binary: File
name: String
testFile: File
TestCase(in file: File)
clearMeta()
getBinary(): File
getMeta(in tester: String): Set
getName(): String
getTestFile(): File
intersectTags(in tags: Tag[]): Tag[]
hasTag(in tag: String): boolean
hasTester(in tester: String): boolean
intersectTags(in tags: Tag[]): Tag[]
setBinary(in binary: File)
hasAnyTagCombination
toString(): String
0..1
- testSet
0..1
- testSet
TestSet
log: Logger
TestSet(in target: String, in isFile: boolean)
addFilesWithFilter(in filter: String, in dir: File): boolean
addFilesWithSuite(in suiteName: String, in folder: File)
addFilesWithTags(in folder: File, in recursive: boolean)
iterator(): Iterator
size(): int
path
- testCases
*

Figure 2.1: Overview of Triops core classes

ch.ethz.sct.triops is the main package of triops and contains the basic application used for building test sets, running tests.

ch.ethz.sct.triops.xml The processing of the XML files is done with XMLBeans which automatically maps XML entities to Java objects and attributes to fields and provides access to the XML data in an object-oriented way. XMLBeans offers the possibility to extend the interfaces of the generated objects by static methods to ease access to XML data[xbe07].

ch.ethz.sct.triops.utils classes used by the core classes and testers to produce a `diff`-Output (`DiffBuilder`) and to handle internal logging (`TriopsLogHandler`)

testers contains all testers and their abstract class `AbstractTester`.

2.9.2 Extending AbstractTester

Introducing a new tester requires inheriting from `AbstractTester` and implementing the `build` method. The `build` method should have the following skeleton:

- determine the set of produced files
- rename files if necessary (`renameFiles`)
- generate the parameters for external application
- invoke external application (`invokeProcess`)
- update testcase's metadata

To access data previously produced by other testers each tester should associate his produced output files with itself by adding it to `TestCase`'s metadata. For comfort a produced binary should be stored directly as a `TestCase`'s binary instead. All files that should be considered for comparison should be added to the compare set. `invokeProcess` only adds the captured output and error stream to the compare set but not potential produced files by the external application. The comparison will be done automatically after the execution of `build`.

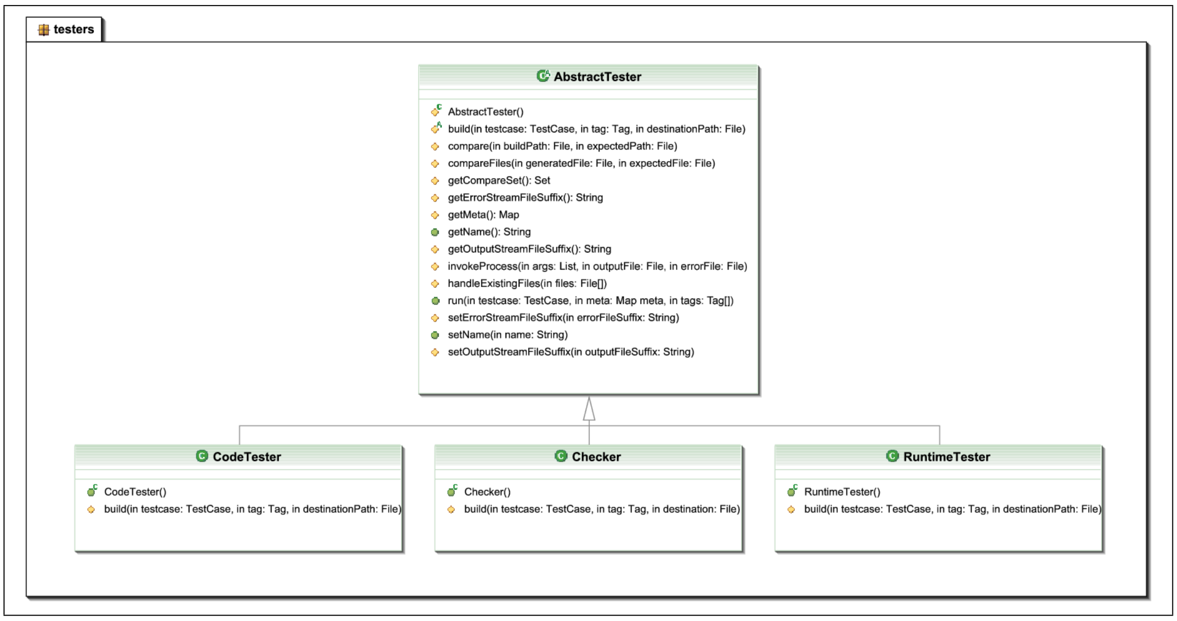
testers
AbstractTester
AbstractTester()
build(in testcase: TestCase, in tag: Tag, in destinationPath: File)
compare(in buildPath: File, in expectedPath: File)
compareFiles(in generatedFile: File, in expectedFile: File)
getCompareSet(): Set
getErrorStreamFileSuffix(): String
getMeta(): Map
getName(): String
getOutputStreamFileSuffix(): String
invokeProcess(in args: List, in outputFile: File, in errorFile: File)
handleExistingFiles(in files: File[])
run(in testcase: TestCase, in meta: Map meta, in tags: Tag[])
setErrorStreamFileSuffix(in errorFileSuffix: String)
setName(in name: String)
setOutputStreamFileSuffix(in outputFileSuffix: String)
CodeTester
CodeTester()
build(in testcase: TestCase, in tag: Tag, in destinationPath: File)
Checker
Checker()
build(in testcase: TestCase, in tag: Tag, in destination: File)
RuntimeTester
RuntimeTester()
build(in testcase: TestCase, in tag: Tag, in destinationPath: File)

Figure 2.2: AbstractTester with implementing testers

2.9.3 Control flow

Triops can be started from console (with the `Main` class) or directly through the `Triops` class. Invoking triops from the console will parse the provided parameters and will create and configure a `Triops` object accordingly (see figure 2.1). After starting the `Triops` object it will create a `TestSet` based on a provided target (which is a suite name or a testcase name). The TestSet will traverse the folder structure and process suites in a recursive manner until all testcases are collected. After creating a TestSet a TestManager is created which is responsible for coordinating the testcase execution operation and based on the user-provided input and the defined components and testers. It determines which setups must be executed and will invoke each of its components with all `TestCases` from the TestSet. In fact the component itself will not be invoked because it's only an object representing an XML entity but it will invoke the referred AbstractTester and passes the necessary metadata along (e.g. the environment, setup and the tags known). These will produce the output data and will either store it as expected output or as generated ouput and compare it against the expected output.

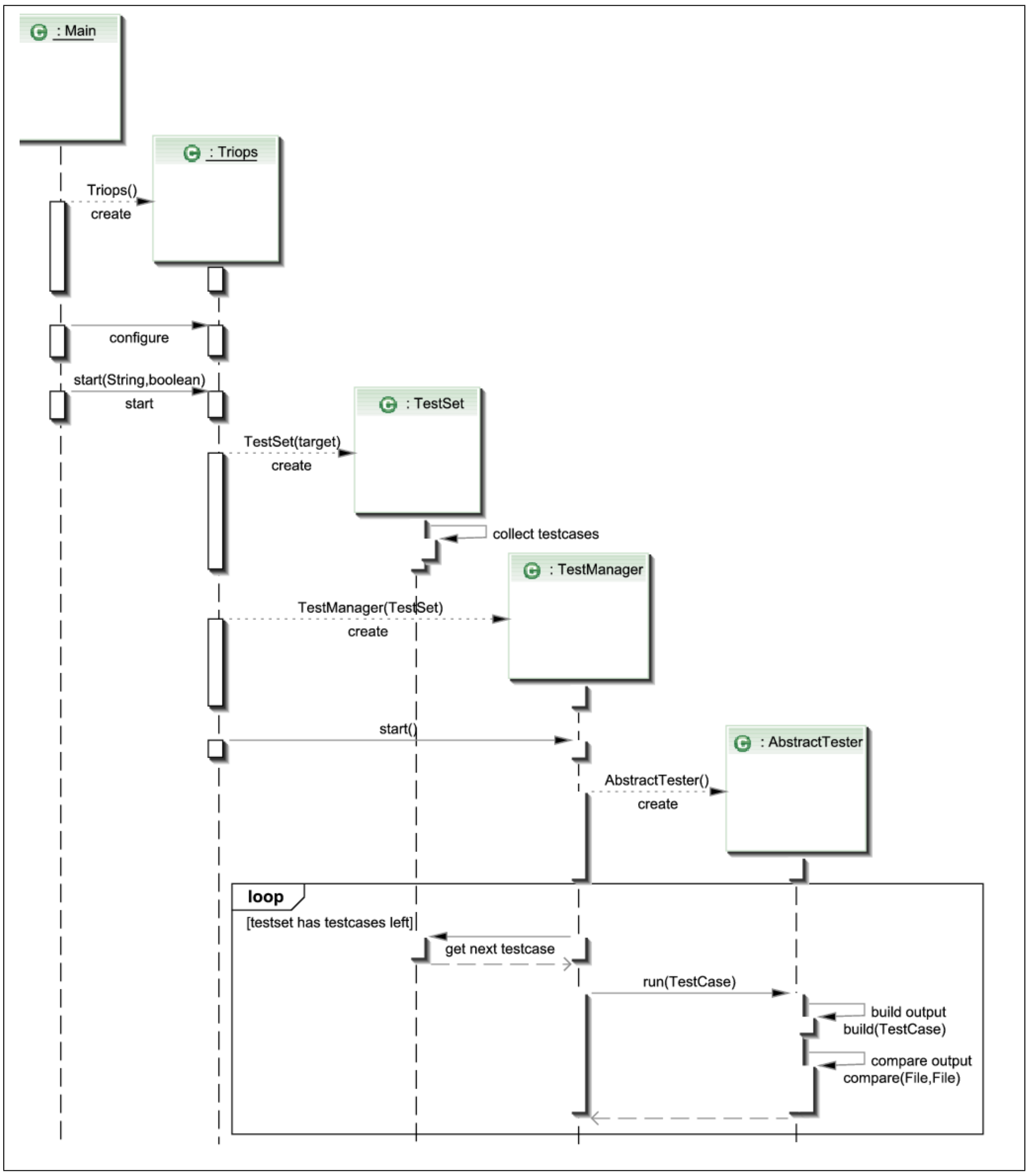

Figure 2.3: Simplified sequence diagram of Triops

Chapter 3

Evaluation and Conclusions

3.1 Case Study: Universe Type System

The Universe Type System (UTS) has been implemented for MultiJava and JML and for both compilers separate testcases have been written which are bound to the compiler. As a case study the testcases will be migrated from MultiJava to Triops with respect to a later portation of the JML testcases. This case study should show the basic structure of the ported solution and won't comment every detail, therefore certain details will be dropped for higher readability.

3.1.1 Tags

MultiJava and JML both are able to understand UTS annotations but they do not support the same syntax. JML supports annotations in JML comments whereas MultiJava supports special keywords. Due to the different syntax one compiler won't be able to understand every testcase. Tags will be used to mark testcases according to their syntax so that a component can either skip a unsupported syntax or pass the correct flags to the tester to enable the support for this syntax. Three tags have been defined based on the existing MultiJava testcases and other testcases which are not subject of this case study but may be ported in the future.

utskeywords testcases which use keywords for MultiJava

utsjmlkeywords testcases which use annotations in JML comments

utsjmlcompatible testcases which use annotations with backslash in JML comments

All testcases of this case study will receive the tag `utskeyword` because only MultiJava testcases are ported.

3.1.2 Testcase organisation

The testcases are divided in four groups, where each is collected in a own folder: `base` (compile tests), `codegen` (byte code tests), `runtime` (runtime tests) und `bugs` (compile tests with some other parameters). The `codegen` testcases are divided once more into `annotations` and `attributes` which gives a total of six folders. Each testcase in a folder will be processed by the same setup which results in five different setup (the `codegen` folder does not contain any testcases besides the two subfolder and doesn't need a own setup).

3.1.3 config.xml

Each group uses the MultiJava compiler with a different configuration, hence an adapted `check` component is needed for each setup. The two `codegen` subfolders use the same decompiler and can share one `code` component. Only one runtime is needed by the `runtime` folder so there is no

need for more than one `runtime` component.
One setup is needed per group and all test the MultiJava compiler which is mapped to an environment `multijava` which contains the five setups for the five groups.

```
<config xmlns="http://ethz.ch/sct/triops">

    <!-- check components -->
    <component type="check" name="mjc-bugs" command="mj-c.cmd">
        <tag name="utskeywords" flags="-W --universes" />
    </component>
    <component type="check" name="mjc-runtime" command="mj-c.cmd">
        <tag name="utskeywords" flags="-universes" />
    </component>
    <component type="check" name="mjc-base" command="mj-c.cmd">
        <tag name="utskeywords" flags="-W --universes -w2 --filter org.multijava.mjc.UniverseFilter" />
    </component>
    <component type="check" name="mjc-attr" command="mj-c.cmd">
        <tag name="utskeywords" flags="--universesx=parse,check,xbytecode" />
    </component>
    <component type="check" name="mjc-anno" command="mj-c.cmd">
        <tag name="utskeywords" flags="--universesx=parse,check,annotations" />
    </component>

    <!-- code components -->
    <component type="code" name="mj-dis" command="mj-dis.cmd">
        <tag name="utskeywords" flags="--universes --stdout --sorted" />
    </component>

    <!-- runtime components -->
    <component type="runtime" name="java" command="java-run.cmd">
        <tag name="utskeywords" />
    </component>

    <!-- environments -->
    <environment name="multijava">
        <setup name="bugs">
            <item>mjc-bugs</item>
        </setup>
        <setup name="base">
            <item>mjc-base</item>
        </setup>
        <setup name="codegen-attr">
            <item>mjc-attr</item>
            <item>mj-dis</item>
        </setup>
        <setup name="codegen-anno">
            <item>mjc-anno</item>
            <item>mj-dis</item>
        </setup>
        <setup name="runtime">
            <item>mjc-runtime</item>
            <item>java</item>
        </setup>
    </environment>
</config>
```

3.1.4 Suites

Only one `suites.xml` was created in the testcase root with a suite for each base group. The subfolders do not have any further `suites.xml` i.e. the referred suites are implicitly declared (section 2.7.3). All testcases can be selected by using the implicit suite `utskeywords`.

```
<testsuites xmlns="http://ethz.ch/sct/triops">
    <suite name="bugs">
        <subfolder name="bugs" suiteName="utskeywords" />
    </suite>
    <suite name="codegen">
        <subfolder name="codegen" suiteName="utskeywords" />
    </suite>
    <suite name="runtime">
        <subfolder name="runtime" suiteName="utskeywords" />
    </suite>
    <suite name="base">
        <subfolder name="base" suiteName="utskeywords" />
    </suite>
</testsuites>
```

3.1.5 Issues

The package delcaration had to be deleted from all testcases and almost every `runtime` testcase used secondary classes which had to be renamed to prevent name clashes (see section 2.1).
Three issues have been encounter which needed some workarounds:

Compilation dependencies Some `runtime` testcases use a helper class `external.java`, which should be compiled without UTS support to model a interface between an UTS class and a class without UTS support. The exernal class must be compiled before the accessing UTS testcase because otherwise the compiler would detect the absence of the external file during compilation of the testcase and will compile it with UTS support, which is wrong. A mechanism is needed to model an order in which a tester will process the testcases.

Extending CheckTester One could try to solve the previous example by introducing a new tester `external` which would be executed before the `check` tester and would compile all external files (which one would have to mark with a new tag). This solution guarantees that all external classes would be compiled first but due to the new tag the classes would be compiled into a different output folder (as we remember, the tag is part of the output path). The testcase which needs the external doesn't know that destination path. One would need to adapt the existing `CheckTester` to include the folder with the compiled external files into the classpath of the compilation.

Cyclic compilation dependencies A similar issue as compilation dependencies occured with some other `runtime` tests. A scenario appeared where a file had to be compiled with UTS support, then one without and again one file with enabled support and the order was fixed because they depended on each other. Adapting an existing tester or introducing a new one can't solve this problem because at least one tester would have to executed twice in a setup which violates the restriction that a tester is allowed only once per setup.

Workaround all three presented issues were related with modeling interaction of UTS classes with non-UTS classes. As a workaround all external files have been compiled by hand and have been placed in a separate folder which is included in the classpath of compilation and runtime execution. Compiling the classes by hand is somehow reasonable because the external classes are part of the testcase and there is no need to recompile them every time.

3.2 Conclusions

The case study confirmed how simple and small the configuration can be realised in Triops compared to the Makefile system of MultiJava. The new organisation can be used for porting further testcases from JML and to include other compilers which can be tested with the same testcases. Keeping the configuration data in XML files makes them readable and easy maintainable. The introduction of abstraction allows the reuse of functionality by sharing a tester amongst components and using the same component in multiple setups.
Future work may concentrate on solving the problems discovered during the case study and on building a user interface optimise the user interaction with Triops in a more powerful graphical way.

3.3 Future Work

Allowing package declarations The problem is described in section 2.1. Future version of Triops could retrieve `package` information from testcases with reflection.

Allowing inner and secondary classes The problem is described in section 2.1. Future version of Triops could analyse testcases with reflection and solve name clashes.

Eclipse integration An Eclipse plugin could be designed which allows the easy execution and managing of testcases through an user interface.

(Cyclic) compilation dependencies The problem is described in section 3.1.5. Further versions could try to solve this compilation dependencies.

Porting JML testcases The JML testcases could be ported to Triops as another case study.

Using Jakarta Commons Digester instead of XML Beans XML Beans maps XML entities to objects and allows the user to extend its interface with static methods. Sometimes more advanced features would have been appreciated to make the design more elegant e.g. instead of creating an XML object, reading and forwarding all of its data to another object is not that elegant. It would be nicer to be able to inherit from the XML object or to have a framework which will directly create and customise the 'real' object instead of making a detour over a dummy XML object. Jakarta Commons Digester could be a framework which fits the described needs.

Bibliography

[wik07a] Black box testing. http://en.wikipedia.org/wiki/Black_box_testing, Mar 2007.

[wik07b] Diff. http://en.wikipedia.org/wiki/Diff, Mar 2007.

[wik07c] Longest common subsequence. http://en.wikipedia.org/wiki/Longest_common_subsequence_problem, Mar 2007.

[xbe07] XML Beans - Interface Extension. http://wiki.apache.org/xmlbeans/ExtensionInterfacesFeature, Mar 2007.

List of Figures

Appendix A

tester.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://ethz.ch/sct/triops"
    targetNamespace="http://ethz.ch/sct/triops" elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:element name="testers">
        <xs:complexType mixed="false">
            <xs:sequence minOccurs="0">
                <xs:element name="tester" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType mixed="false">
                        <xs:attribute name="name" type="xs:Name" use="required"/>
                        <xs:attribute name="class" use="required">
                            <xs:annotation>
                                <xs:documentation>valid java class name</xs:documentation>
                            </xs:annotation>
                            <xs:simpleType>
                                <xs:restriction base="xs:string">
                                    <xs:pattern value="[a-zA-Z_$][a-zA-Z0-9_$]*"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:attribute>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
        <xs:unique name="name">
            <xs:selector xpath="./tester"/>
            <xs:field xpath="@name"/>
        </xs:unique>
    </xs:element>
</xs:schema>
```

Appendix B

suite.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://ethz.ch/sct/triops"
    targetNamespace="http://ethz.ch/sct/triops" elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:element name="testsuites">
        <xs:complexType mixed="false">
            <xs:sequence minOccurs="0">
                <xs:element name="suite" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType mixed="false">
                        <xs:choice maxOccurs="unbounded">
                            <xs:element name="file" maxOccurs="unbounded">
                                <xs:complexType mixed="false">
                                    <xs:attribute name="filter" use="required">
                                        <xs:annotation>
                                            <xs:documentation>only valid class names or filter
                                                expressions</xs:documentation>
                                        </xs:annotation>
                                        <xs:simpleType>
                                            <xs:restriction base="xs:string">
                                                <xs:pattern value="[a-zA-Z_$][a-zA-Z_$.0-9]*"/>
                                                <xs:pattern value="([a-zA-Z_$][a-zA-Z_$.0-9]*)?
                                                    \*[a-zA-Z_$.0-9]*"/>
                                                <xs:pattern value=""/>
                                            </xs:restriction>
                                        </xs:simpleType>
                                    </xs:attribute>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="subfolder" maxOccurs="unbounded">
                                <xs:complexType mixed="false">
                                    <xs:attribute name="name" type="xs:Name" use="required"/>
                                    <xs:attribute name="suiteName" type="xs:Name" use="required"/>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="tags" maxOccurs="unbounded">
                                <xs:annotation>
                                    <xs:documentation>tags are alphanumeric strings with _ and
                                        maybe a ! in front</xs:documentation>
                                </xs:annotation>
                                <xs:simpleType>
                                    <xs:list>
                                        <xs:simpleType>
                                            <xs:restriction base="xs:string">
                                                <xs:whiteSpace value="replace"/>
                                                <xs:pattern value="!?[a-zA-Z0-9_]+"/>
                                            </xs:restriction>
                                        </xs:simpleType>
                                    </xs:list>
                                </xs:simpleType>
                            </xs:element>
                        </xs:choice>
                        <xs:attribute name="name" type="xs:Name" use="required"/>
                    </xs:complexType>
                    <xs:unique name="fileName">
                        <xs:selector xpath="./file"/>
                        <xs:field xpath="@name"/>
                    </xs:unique>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
        <xs:unique name="suitename">
            <xs:selector xpath="./suite"/>
            <xs:field xpath="@name"/>
        </xs:unique>
    </xs:element>
</xs:schema>
```

Appendix C

config.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ns1="http://ethz.ch/sct/triops"
    targetNamespace="http://ethz.ch/sct/triops" elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:element name="config">
        <xs:complexType mixed="false">
            <xs:sequence minOccurs="0">
                <xs:element name="component" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence minOccurs="0">
                            <xs:element name="tag" minOccurs="0" maxOccurs="unbounded">
                                <xs:complexType>
                                    <xs:attribute name="name" use="required">
                                        <xs:simpleType>
                                            <xs:restriction base="xs:string">
                                                <xs:pattern value="[a-zA-Z0-9_]+"/>
                                                <xs:pattern value="\*"/>
                                            </xs:restriction>
                                        </xs:simpleType>
                                    </xs:attribute>
                                    <xs:attribute name="flags" type="xs:string"/>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                        <xs:attribute name="type" use="required">
                            <xs:simpleType>
                                <xs:restriction base="xs:string">
                                    <xs:pattern value="[a-zA-Z0-9_]+"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:attribute>
                        <xs:attribute name="name" type="xs:Name" use="required"/>
                        <xs:attribute name="command" type="xs:normalizedString" use="required"/>
                    </xs:complexType>
                    <xs:unique name="tagname">
                        <xs:selector xpath="./tag"/>
                        <xs:field xpath="@name"/>
                    </xs:unique>
                </xs:element>
                <xs:element name="environment" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="setup" minOccurs="0" maxOccurs="unbounded">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="item" type="xs:Name" maxOccurs="unbounded"/>
                                    </xs:sequence>
                                    <xs:attribute name="name" type="xs:Name" use="required"/>
                                </xs:complexType>
                                <xs:unique name="itemContent">
                                    <xs:selector xpath="./item"/>
                                    <xs:field xpath="item"/>
                                </xs:unique>
                            </xs:element>
                        </xs:sequence>
                        <xs:attribute name="name" use="required">
                            <xs:simpleType>
                                <xs:restriction base="xs:Name"/>
                            </xs:simpleType>
                        </xs:attribute>
                    </xs:complexType>
                    <xs:unique name="setupName">
                        <xs:selector xpath="./setup"/>
                        <xs:field xpath="@name"/>
                    </xs:unique>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
        <xs:unique name="environmentName">
            <xs:selector xpath="./environment"/>
            <xs:field xpath="@name"/>
        </xs:unique>
        <xs:unique name="compilerName">
            <xs:selector xpath="./component"/>
            <xs:field xpath="@name"/>
        </xs:unique>
    </xs:element>
</xs:schema>
```