College of Engineering and Computer Science

Code Coverage Tool for X10

COMP4560 Advanced Computing Project

Uni ID: U4788232
Name: Zhan Yuan Hiu
Supervisor: Joshua Milthorpe
Acknowledgement

I want to thank my supervisor, Joshua Milthorpe for his guidance and knowledge all the way throughout the whole project. His help has been invaluable in this project.

I also want to thank Dr. Weifa Liang for all the knowledge he shared during the weekly meeting. It has been most helpful in managing project schedule, giving presentation and writing this report.

Lastly, I would also thank JaCoCo core team as well as X10 core team for communicating with me during this project.
## Contents

Acknowledgement .................................................................................................................. 2  
Abstract .................................................................................................................................. 4  
Introduction ............................................................................................................................. 5  
Background ................................................................................................................................ 6  
  Code Coverage ......................................................................................................................... 6  
  Why use code coverage? ............................................................................................................. 6  
  Categories of code coverage ................................................................................................. 6  
X10 ........................................................................................................................................... 7  
  X10 Features .......................................................................................................................... 7  
  X10 Practical Uses ................................................................................................................ 8  
Project & Requirements ........................................................................................................... 9  
Development Environment ...................................................................................................... 9  
Extending on Existing Code Coverage Tool ............................................................................. 10  
  JaCoCo ................................................................................................................................... 10  
Challenges in Measuring Coverage of X10 Code ...................................................................... 13  
Example output report of unmodified JaCoCo ......................................................................... 15  
Work Done .............................................................................................................................. 17  
Improvement to X10 Language ................................................................................................. 18  
Analysis of the software .......................................................................................................... 20  
Analysis of Test Suite .............................................................................................................. 24  
Flaws of the software and Future Work .................................................................................... 25  
Conclusion ............................................................................................................................... 28  
Bibliography ............................................................................................................................ 29  
Appendix I ............................................................................................................................... 30  
Appendix 2 ............................................................................................................................... 33  
Appendix 3 ............................................................................................................................... 35  
Appendix 4 ............................................................................................................................... 37
Abstract
Software testing is one of the most important aspects in the software development cycle. A well written test suite can ensure a smooth launch of software and increase confidence for the software itself. One way of ensuring we have a good test suite is by using code coverage.

In this project, a Java code coverage tool, JaCoCo is modified to support the X10 programming language and improve readability when using it on X10 code. This project is important to the X10 community because there is currently no code coverage tool for x10 at the moment.

X10 already has an extensive test suite, but coverage information has not previously been available. As X10 is a fairly new language, it would not be surprising if there are still a lot of undiscovered bugs. Applying the X10 code coverage tool to the X10 test suite identified important areas of code for which tests are missing, and could be improved.

The benefit of this approach compared to output of native JaCoCo is that we get to see the code coverage information on X10 code itself, rather than the generated Java code. Coverage information on X10 source code is meaningful to an X10 developer without intimate knowledge of code generation in the X10 compiler. Generated ‘boilerplate’ code that may be assumed to be reliable is excluded from the output so as not to obscure coverage information for more important hand-written code. The additional generated methods and classes to optimize the performance of X10 on JVM are combined with their respective parent/original methods and classes to map to the original X10 methods and classes.
**Introduction**

X10 is a new parallel programming language developed by IBM for high performance computing. X10 is a managed, object-oriented language like Java, with support for distributed parallelism. X10 utilizes dual-path source-to-source compilation, which is to either C++ or Java.

Before this project, there is no code coverage tool available for the language. The workaround this is to use the native code coverage tool available on C++ or Java on the generated code. This method is not productive, as the output report is hard to read with methods and classes name mangled and additions of internally generated classes to guarantee the interoperability between X10 and Java.

Therefore, this project is undertaken in order to develop a code coverage tool for X10. Also, the developed code coverage tool will be tested against the X10 test suite.
Background

Code Coverage
Code coverage is a measure of degree to which the source code is tested by test suites. It is a measure of the quality of the test cases for software. A high quality test suite may increase confidence in the quality of the software.

Why use code coverage?
Code coverage allows the developers to find areas of program that is not tested, or junk code that has not been removed. By finding these untested codes, developers can write additional tests for them and discover additional bugs and problems.

Although it is always possible to achieve 100% code coverage in software, it is not necessary to do so. This is because the last few percentage of code coverage is usually hard to be achieved, due to hard to reach exceptions.

Categories of code coverage

Line/Statement Coverage
Line or statement coverage is the simplest and the most well known coverage type. It is also the most useful type of code coverage. A line is considered covered when an instruction associated with the line is covered. [Coverage Counters, 2013]

Branch Coverage
Branch coverage allows the developers to detect which path is executed when there is a branch. It is useful as line coverage cannot always tell users if all the branches are being executed. This usually happens when there is an uneven amount of codes in different branches.

Example:

```java
if (x == 1)
    System.out.println("decision1");
else if (x != 1) {
    System.out.println("decision2");
    //99 more lines of the same print statement
}
```

When x is not 2, line coverage would gives 101/102 lines covered. It would not be obvious to the users that the first branch is not being covered. It is easily detected with the usage of branch coverage, as it gives the result of 50% of the branch path is covered.

Method Coverage
A method is considered covered if any instruction associated with the method is covered.

Class Coverage
A Class is considered covered if at least one instruction associated to the method is executed.
Cyclomatic complexity
Cyclomatic complexity is the minimum number of paths that can, in (linear) combination, generate all possible paths through a method.” The complexity number can be used as an indication of the number of unit tests to fully cover the whole method. [Watson, 1996]

X10
X10 is a modern object-oriented programming language in the APGAS family. X10 is Java-like, and specialize in locality and parallel activities. X10 is a fairly new language, with its development started in 2004.

X10 utilizes dual path compilations. Developers can either choose to compile to C++ (native x10) or compile to Java (managed x10). X10 also has an Eclipse-based IDE, which is a plus to any programmers using Eclipse.

X10 Features
The most important X10 feature is its support for concurrency. The following concepts are fundamental to X10.

Activity
An activity is a statement that can be executed and has its own variables. It is similar to threads and is more light weight compared to it. An X10 program consists of one or more activities executing in parallel.

Place
“X10 place is a repository for data and activities, corresponding loosely to a process or a processor.” (Sarawat & Bloom & Peshansky, 2013) This introduces the concept of locality. When an activity accessing some data on that place, it will be very efficient and fast compared to accessing some data on remote place.

Async
The “Async” keyword creates a new activity. The activity will be located “here” (the current place of the activity calling async) if no place is specified. By calling “async s”, s is a statement that will be executed asynchronously while the current activity continues.

Finish
By calling “Finish”, the parent activity will waits for its child activities to terminate before continuing. Finish can be used to synchronize with other activities.

At
Using “At” allows developers to specify the place they want to run certain activities.

```java
for (var i:Int=0; i<Place.MAX_PLACES; i++) {
    val iVal = i;
    async at (Place.places(iVal)) {
        Console.OUT.println("Hello World from place "+here.id);
    }
```
The code segment above asynchronously prints “Hello World from place x” at multiple places, where x is the place’s id.

**Atomic**

“Atomic” block allows developers to run statements as if other activities are suspended. However, there are certain restrictions when using atomic blocks. Atomic activity may not spawn another activity, use at expressions, and run any blocking statements (when, finish etc).

**X10 Practical Uses**

X10 is already been used in the practice. Some of the applications written in X10 include ANUChem (a collection of computational chemistry codes) [ANUChem, 2013], Megaffic traffic flow simulator, ScaleGraph graph library [ScaleGraph], [Osogami & et al 2012], Global Matrix Library and also X10 runtime itself that is used in X10.

There is an extensive test suite for X10, but there is no code coverage tool for X10. Therefore, there is no reliable measurement of the quality of the X10 test suite.
Project & Requirements
For this project, it is required to develop a prototype code coverage tool for current release of X10 (2.4), running code coverage tool together with X10 test suite as well as finding flaws on current X10 release and attempt to improve on X10 language.

These are the requirements for the code coverage tool:

1. Code coverage tool output at least one type of coverage information, line/ statement coverage in X10 source code. Code coverage tool has to be modular to ease future updates.
2. Code coverage tool has to have reasonable runtime overhead.
4. Code coverage tool has to work on the full X10 test suite.

Development Environment
In this project, Eclipse 4.3 (Kepler) is used as Java IDE in modifying JaCoCo.

Maven build tool is used to compile JaCoCo.

Ant build tool is used to build X10, as well as generating code coverage report.

Shell Script is used to allow the usage of JaCoCo along with X10 test suite.

The X10 2.4 is used in this project.

Java 1.7.0_25 (Java 7 Update 25) is used in this project.

Jacoco 6.3 is used in this project.
Extending on Existing Code Coverage Tool

Due to the fact that X10 program runs as Java bytecode in JVM or as C++ program, code coverage information can only be obtained by running code coverage tools in C++/Java. Therefore, it is a better choice to extend existing tool than writing a new code coverage tool from scratch.

On Java platform, there are a few code coverage tools available (Emma, JaCoCo, Cobertura and Clover).

On C++ platform, there are Gcov and Lcov. Due to my expertise in Java compared to C++, I chose to work on Java backend and therefore, these tools are excluded as options.

There are a few requirements to be met when choosing the code coverage tool:

1. The code coverage tool has to be open-sourced.
2. The code coverage tool has to have an active development community and not dead.
3. The code coverage tool has to support the newest Java version (Java 1.7).

Clover is excluded as an option as it is not open-sourced.

Emma has not been updated for a few years. Its last official release in sourceforge was in 2005 and therefore, the project is considered “dead”. Also, according to the forums, Emma has problems with Java v7. Therefore, Emma is not a suitable candidate. [EMMA, 2006]

Cobertura meets the requirements. However, from March 2010 until May 2013, there is not a single update for the software. The project is assumed to be dead and resurrected recently in 2013. The current version of Cobertura could not be built from source using the instructions provided. [Cobertura, 2013]

JaCoCo

JaCoCo is another code coverage tool that met all the requirements and is therefore chosen. JaCoCo offers simple integration and is known to support multiple JVM languages. JaCoCo also boasts minimal runtime overhead and comprehensive documentations that is available. [JaCoCo, 2013].

JaCoCo's Ways of Instrumenting Class

On-The-Fly Instrumentation

On-The-Fly class instrumentation requires the usage of Java agent. As X10 supports the usage of Java agent, this option is chosen as the way to instrument class as it is the easiest option. The user only need to include addition argument "-Javaagent:lib/JaCoCoAgent.jar" when running Java programs.

Offline-Instrumentation

The Java classes which is going to be analyzed has to be instrumented separately before executing them. This can be done using either Ant or Maven build script using Jacocoagent.jar. In the offline mode, JaCoCo runtime has to be configured. Also, instrumented classes get a direct dependency on JaCoCo runtime, so jacocoagent.jar has to be on the classpath and accessible by the instrumented classes.
**JaCoCo’s Output**

The colour in JaCoCo’s output represents the state of code coverage information.

Figure 1 shows an output Java source code from JaCoCo. For line coverage, green coloured source line means the source line is covered and red coloured source line means the line is not covered. This is the same for branch coverage. Additionally, yellow coloured source line means the branch is partially covered (not all the branches are traversed). Branches are tagged with a diamond as shown on the figure below.

```java
IAgentOutput createAgentOutput() {
    final OutputMode controllerType = options.getOutput();
    switch (controllerType) {
    case file:
        return new FileOutput();
    case tcpserver:
        return new TcpServerOutput(logger);
    case tcpclient:
        return new TcpClientOutput(logger);
    case none:
        return new NoneOutput();
    default:
        throw new AssertionFatalError(controllerType);
    }
}
```

**Figure 1. Example JaCoCo source code output**

Figure 2 shows an example class page. In the class page, there is a row for each method. For every row, from the left, JaCoCo displays missed instructions of the method, coverage percentage of the instructions, missed branches in the method, branch coverage percentage of the method, cyclomatic complexity and missed cyclomatic complexity, total lines in a method and missed lines, total methods and missed methods.
<table>
<thead>
<tr>
<th>Element</th>
<th>Missed Instructions</th>
<th>Cov.</th>
<th>Missed Branches</th>
<th>Cov.</th>
<th>Missed</th>
<th>Cty</th>
<th>Missed</th>
<th>Lines</th>
<th>Missed</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>getInstance(AgentOptions)</td>
<td>0%</td>
<td></td>
<td>0%</td>
<td></td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>getInstance()</td>
<td></td>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getExecutionData(boolean)</td>
<td>76%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>createAgentOutput()</td>
<td>85%</td>
<td></td>
<td>80%</td>
<td></td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>createSessionId()</td>
<td>83%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>startup()</td>
<td>100%</td>
<td></td>
<td>100%</td>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>shutdown()</td>
<td>100%</td>
<td></td>
<td>100%</td>
<td></td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Agent(AgentOptions, IExceptionLogger)</td>
<td>100%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>setSessionId(String)</td>
<td>100%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>dump(boolean)</td>
<td>100%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>setSessionId()</td>
<td>100%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>reset()</td>
<td>100%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>getData()</td>
<td>100%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>getVersion()</td>
<td>100%</td>
<td></td>
<td></td>
<td>n/a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>43 of 209</td>
<td>79%</td>
<td>5 of 17</td>
<td>71%</td>
<td>5</td>
<td>24</td>
<td>14</td>
<td>63</td>
<td>2</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 2. Example JaCoCo Class Page
Challenges in Measuring Coverage of X10 Code

Prior to this project, to check code coverage in X10, one is required to use the existing code coverage tool on C++/Java platform, the programming language that X10 compiles to. Even with the usage of existing tools on Java/C++, it is not efficient and productive to do so, due to the following issues.

Expansion of code generated

Based on the code samples in official X10 2.4 release (located in x10.dist/samples), each line of X10 results in an average of around 10 lines of generated Java code. This result even includes the comments on X10 codes. Besides being much longer than the X10 source code, the generated Java code is usually hard to read with the generation of new internal variables and methods. Some of the X10 classes and methods are also mangled with additional information when compiling X10 to Java. Examples of HelloWorld in X10 and its generated Java counterpart can be founded in the Code section at the end of the report.

Table 1. Expansion in generated Java source code

<table>
<thead>
<tr>
<th>Sample Code</th>
<th>X10 lines of code</th>
<th>Generated Java line of code</th>
<th>Magnitude of expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArraySum</td>
<td>72</td>
<td>771</td>
<td>10.71</td>
</tr>
<tr>
<td>GCSphere</td>
<td>120</td>
<td>1045</td>
<td>8.71</td>
</tr>
<tr>
<td>HelloWorld</td>
<td>43</td>
<td>276</td>
<td>6.42</td>
</tr>
<tr>
<td>HelloWholeWorld</td>
<td>23</td>
<td>104</td>
<td>4.52</td>
</tr>
<tr>
<td>Histogram</td>
<td>45</td>
<td>586</td>
<td>13.02</td>
</tr>
<tr>
<td>KMeans</td>
<td>151</td>
<td>2010</td>
<td>13.31</td>
</tr>
<tr>
<td>KMeansDist</td>
<td>157</td>
<td>2818</td>
<td>17.95</td>
</tr>
<tr>
<td>KMeansSPMD</td>
<td>210</td>
<td>2324</td>
<td>11.07</td>
</tr>
<tr>
<td>MontyPi</td>
<td>42</td>
<td>412</td>
<td>9.81</td>
</tr>
<tr>
<td>NQueensDist</td>
<td>119</td>
<td>1229</td>
<td>10.33</td>
</tr>
<tr>
<td>NQueensPar</td>
<td>121</td>
<td>1380</td>
<td>11.40</td>
</tr>
<tr>
<td>StructSpheres</td>
<td>123</td>
<td>1746</td>
<td>14.20</td>
</tr>
</tbody>
</table>

X10 Closure

Closure functions are featured by X10.

The syntax for closure expression for X10 is

\[
\text{ClosureExp ::= Formals Guards? HasReturnType? => ClosureBody}
\]
An example of closure function code in X10 is the calculation to find pi in MontyPi.X10 in x10.dist/samples.

```java
Val pi = 4*result.reduce((x:Double, y:Double) => x+y, 0.0)/N*Place.MAX_PLACES);
```

Whenever a closure function is used, an additional Java class and its internal methods will be generated. These functions are evaluated by calling their $apply method. If a X10 source code contains many closure functions, there will as many additional closure class. This reduces readability of the code coverage tool’s report.

**X10 Constructors**

When X10 is compiled to Java, every X10 constructor is converted into a 2 phase constructor (a pair of Java constructors and an instance field initializer) [Takeuchi al et, 2011]. The purpose of doing so is for interoperability between X10 and Java. Every X10 class requires a Java constructor. However, the Java constructor is not called at all and only the other constructor and the field initializer is used. This results in one of the constructor code not being covered and reduction in code coverage percentage.

An example of constructor for HelloWorld would be used. The full code of HelloWorld.x10 and HelloWorld.Java will be available in Appendix I.

```java
//#line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
// creation method for Java code (1-phase Java constructor)
public HelloWorld(){this((Java.lang.System[]) null);
HelloWorld$$init$S();}
```

The above code segment (the first constructor) will not called or used.

```java
// constructor for non-virtual call
final public HelloWorld HelloWorld$$init$S() { {
```

This second constructor is used instead.

**X10 Class and Method name mangling**

X10 compiler mangles the X10 methods’ name with additional information. The reasoning for this is because X10 allows methods with same name but different type parameters [Takeuchi al et, 2011]. On the other hand, Java does not allow this. To support this, the method name is mangled with the
parameters when compiling X10 to Java. The method name mangling has reduced readability in the code coverage output.

An example of name mangling would be Method `parse(Rail[String], Rail[Option])` in `x10.util/OptionsParser`. The mangled method name is `parse__0$1x10$lang$String$2__1$1x10$util$Option$2(Rail, Rail)`. In this example, the type of Rail is mangled onto the method’s name.

**X10 Static Fields**

Whenever a non-trivial static field (cell, custom class) is declared in X10, the X10 compiler will generate additional code with many “if and else” clauses and “try and catch” clauses. Most of the code is unreachable and meant for debugging and catching rarely occurring exceptions. This results in lower code coverage, especially when there are many non-trivial static fields in the X10 source code.

An example of code generated for non-trivial static field is available in Appendix 2.

**Example output report of unmodified JaCoCo**

The image above shows methods within a class. It is hard to read useful information due to name mangling. The mangled method names are so long that all the information will not fit into a normal page. Users have to scroll horizontally to read the coverage data. The figure above has been shrunk in order to show the whole impression of the page. A bigger report output by stock JaCoCo is available in Appendix 2.

![Figure 3. Example class page output from JaCoCo](image)

The image above shows classes within a package page. The name mangling and lots of generated internal classes are similar to the problems ClassPage is facing.
<table>
<thead>
<tr>
<th>Figure 4. Example package page output from JaCoCo</th>
<th></th>
</tr>
</thead>
</table>
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
| JaCoCo | {}
Work Done
Modifications to JaCoCo to Support X10 Code Coverage

All of the modifications done to JaCoCo except methods filtering are done in the report package of JaCoCo to change the output report to support X10 source code, X10 classes and methods. Methods filtering are done in the core package in JaCoCo.

Code coverage information on X10 code
To implement this, each generated Java source line is mapped back to their corresponding X10 code. Each generated Java code has the corresponding X10 line number commented above them when X10 compiles to Java. X10 source code corresponding line number on generated Java source code is assumed to be correct. It is also assumed that one generated Java class maps to one X10 source code. For example, HelloWorld.Java maps to HelloWorld.X10. The code coverage tool will read the comment and store the corresponding X10 line number in an Array.

After getting the corresponding X10 line number for each generated Java code, a for loop is used against the mapping in order to get all the code coverage information for the X10 counterpart.

The newly developed tool is able to output correct line coverage and branch coverage in X10 source code.

The code coverage tool modified to generate 2 reports, one in Java source code and one in X10 source code. The reason for 2 reports is to allow people using the tool to compare and debug. The report on generated Java source code does not combine methods and class, and its main usage is to be used for debugging. The report on x10 source code combines all the internal methods and classes. Methods filtering is enabled for both reports.

Combine generated classes into original X10 class
As mentioned above, there are cases that additional classes are generated. These classes are usually named with the parent class name concatenated by two “$” sign followed by other information depending on the generated classes’ purposes.

First, the algorithm add the classes without “$” into an arraylist. For the classes that contains “$” sign, the algorithm will split the class name at the point of the “$” sign. Then, it compare part of the class name before the “$” sign with the other classes which their name contains no “$” sign. If they match exactly, all the coverage information in the class with “$” sign will be added into the class without the “$” sign. It is assumed that no programmers will include “$” sign in naming classes.

Combine the generated methods together
Contrary to combining classes, methods are combined based on the corresponding first line and last line of the method in x10 code. Methods with bigger first line number and smaller last line number are combined into the methods with smaller first line number and larger first line number.

Method Filtering
To implement method filtering, an “if” condition is added to the analyzer package of JaCoCo. If the
methods name are prefixed with “$” sign, the methods is ignored and not added to be analyzed. The reasoning for this is because the internal generated methods are marked prefixed with “$” sign. These internal generated methods are not relevant to the source code that is analyzed for code coverage.

However, this does not apply to “$apply” method. Basically, X10 functions are being translated to Java classes that implement Java interface `x10.core.fun{Fun, VoidFun}_0_n`. “$apply” methods are called to evaluate these X10 functions. [Takeuchi at el, 2011]

An example of these internal generated methods is $_serialize and $_deserializer. These methods are used for serialization of static fields. To prevent the static fields from having variable values when initializing them, static fields are only initialized in PLACE.First.Place. Then, these static fields are serialized and broadcasted to the places that need them. Finally, these fields will be deserialized. These details and their coverage information will not be useful to X10 programmers. Therefore, they are filtered out.

**Method name demangling and renaming**

As mentioned earlier, when generating Java source code, compiler mangled some of the method name with additional information depends on the parameters and the purposes of the methods. The implemented algorithm is able to remove most of the mangled part. X10 compiler mangles name with both single “$” sign and double “$$” sign.

In the case of methods with “$” sign, when there are multiple methods with same name but different type, the compiler will concatenate the method’s name with “__” followed by a number. Then, it will follow by “$” sign and additional information. The string after “$” sign is removed by the algorithm.

Most of the methods mangled with “$$” sign are followed by init or this. The methods mangled with “$$init” are the constructor that is actually used (The second constructor with field initializer). An example of such constructor can be found in HelloWorld.Java in Appendix 1.

The methods mangled with “$$this” is used to return the whole class. An example code can be found in HelloWorld.Java in Appendix 1. Therefore, the name is not changed to class_this in order to not lose its original purpose.

As mentioned earlier, every X10 closure function generate a Java class with $apply method. In order to improve readability, these $apply methods are renamed to “Some_Closure_Function_At_Lx”, where x is the X10 line number for X10 closure function.

**Improvement to X10 Language**

In addition to developing the X10 code coverage tool, suggestions was also made to the X10 core team in order to improve code coverage information generation.

**Improvement to the X10 compiler**

A few problems were found in existing x10 compiler. Suggestions are made in order to improve both the performance of the X10 compiler as well as to increase code covered percentage.
The current way of X10 core team implementing static initializer, it included the exceptions which are rarely being executed. This reduced the code covered percentage in general.

To better support code coverage without sacrificing performance, the static initializer method could be split into 3 methods, a cold path initialization, a hot path for the case where the field has already been initialized and exception handling. Then, we can filter out the exception handling method since it rarely execute. The suggestion has been accepted by the core X10 team as changes in next release.

We identified some junk code generated when compiling X10 to Java has been identified. Generating these codes takes up additional time during compiling and also cause false code coverage information. These junk codes have been removed by the X10 core team and is fixed for X10 2.4.1. This segment of code was part of the serialization code that was obsolete, but was not taken out until it is reported recently.

private void writeObject(java.io.ObjectOutputStream oos) throws java.io.IOException {
    if (x10.runtime.impl.java.Runtime.TRACE_SER) {
        java.lang.System.out.println("Serializer: writeObject(ObjectOutputStream) of " + this + " calling");
    }
    oos.defaultWriteObject();
}

**Running code coverage tool along with X10 test suite**

A new shell script (codeCoverage) has been written to instrument the Java classes while the test suite is running and generates a code coverage report afterwards. This shell script is placed in x10.tests/bin.

Modifications to the existing build script are committed by Josh Milthorpe for X10 2.4.1. The modifications for includes adding codes in x10.runtime/build.xml that zips every Java and x10 source code, and compiled class files together into x10.jar. The set INCLUDE_JAVA_SOURCE=true option has to be enabled in the build.xml by users. This x10.jar will be used when generating the code coverage report for the X10 test suite.
Analysis of the software

Time Overhead

In order to find out the impact of the modified JaCoCo on the systems, a series of test are carried out. These tests are run on Intel Core i3-2350M running on Linux Ubuntu 13.04. The whole test suite is run on 1 place (X10_NPLACE=1).

Running the full X10 test suite without instrumenting classes takes 3 hours, 46 minutes and 58.7 seconds. Running the full X10 test suite with instrumenting classes takes 4 hours 8 minutes and 55.4 seconds. There is an increase of 21 minutes and 56.7 seconds. There are total of 1980 test cases in the X10 test suite. Therefore, there is an average of 0.665 seconds increase in time overhead per test case, which is reasonable.

Generating a code coverage report after the X10 program is executed takes less than 10 seconds.

Another test is run to compare the time overhead from generating report using stock JaCoCo and the modified version. The code coverage data used data is 22.4 MB in size and obtained by running every test using 2 X10 places in the X10 test suite.

The result shows an average of 1.6 seconds increase in time overhead comparing the modified JaCoCo and stock JaCoCo. The overhead is minimal compared to running the whole test suite itself, which take hours.

Improvements in Readability

The output report for the modified version of JaCoCo is shown below. The output report by stock JaCoCo can be found in Appendix 2.

Figure 5 shows the coloured X10 source code output by modified JaCoCo. The colouring convention remains the same with stock JaCoCo. When pointing to each X10 line of code, it shows how many lines of generated Java code of the corresponding X10 line are covered. If there is branch in the line, it will show the corresponding information as well along with a diamond at the leftside of the line.
```java
public def equals(thatObj: Any): boolean {
    if (this == thatObj) return true;
    if (!(that instanceof Region)) return false;
    val thatRegion = thatObj as Region;

    // we only handle rect==rect
    if (!(that instanceof RectRegion))
        return super.equals(that);

    // ranks must match
    if (this.rank != that.rank)
        return false;

    // fetch bounds
    val thisMin = this.min();
    val thisMax = this.max();
    val thatMin = (that as RectRegion).min();
    val thatMax = (that as RectRegion).max();

    // compare 'em
    for (i in 0..(rank-1)) {
        if (thisMin(i)!-thatMin(i) || thisMax(i)!-thatMax(i))
            return false;
    }
    return true;
}
```

Figure 5. Code Segment that showing part of X10 source code page

Figure 6 shows part of X10 class page with all the X10 methods in it. There is still some part of method name mangling remaining, but majority of the name mangling have been removed. This is definitely readable compared to the output of JaCoCo. The same class page output by stock JaCoCo can be found in Appendix 4. A larger version of Figure 6 is also available in Appendix 4.
Table 2 shows the reduction in methods in class page output by stock JaCoCo and modified JaCoCo. All of the classes have a reduction of over 50%.

### Table 2: Reduction in methods in ClassPage

<table>
<thead>
<tr>
<th>Class</th>
<th>Amount of Java methods</th>
<th>Amount of combined methods</th>
<th>Reduction Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X10.lang/Runtime</td>
<td>162</td>
<td>74</td>
<td>54.3</td>
</tr>
<tr>
<td>X10.lang/Point</td>
<td>131</td>
<td>42</td>
<td>67.9</td>
</tr>
<tr>
<td>X10.regionarray/RectRegion</td>
<td>147</td>
<td>39</td>
<td>73.5</td>
</tr>
<tr>
<td>X10.regionarray/DistArray</td>
<td>107</td>
<td>41</td>
<td>61.7</td>
</tr>
<tr>
<td>X10.util/OptionsParser</td>
<td>120</td>
<td>29</td>
<td>75.8</td>
</tr>
</tbody>
</table>

Figure 7 shows the PackagePage output by modified JaCoCo. Again, majority of the classes are being combined together. The classes with “new” methods mangled to them are core issues in JaCoCo. As seen in figure 7, one of these classes is “Array.Anonymous.14242.Anonymous.newIterator() {...}”. The classPage output by stock JaCoCo is hard to read. The mangled class name are very long that the class page output by stock JaCoCo cannot fit into a page. Users have to scroll horizontally to read the coverage data. A larger version of this figure is available in Appendix 3.
Table 3 shows the magnitude of reduction in classes in packages. The classes reduction is at least 48% from the entries in the table.

### Table 3: Reduction in classes in PackagePage

<table>
<thead>
<tr>
<th>Package</th>
<th>Amount of Java classes</th>
<th>Amount of X10 classes after combining</th>
<th>Reduction Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X10.utils</td>
<td>125</td>
<td>60</td>
<td>52.0</td>
</tr>
<tr>
<td>X10.lang</td>
<td>295</td>
<td>105</td>
<td>64.4</td>
</tr>
<tr>
<td>X10.regionarray</td>
<td>183</td>
<td>50</td>
<td>72.7</td>
</tr>
<tr>
<td>X10.array</td>
<td>52</td>
<td>19</td>
<td>63.5</td>
</tr>
<tr>
<td>X10.compiler.ws</td>
<td>25</td>
<td>13</td>
<td>48.0</td>
</tr>
</tbody>
</table>

### Improvement in code coverage

A series of tests are carried out to determine the magnitude of improvement in code coverage from stock JaCoCo. NQueensDist from x10.dist/samples is being used as benchmark.

Figure 8 above shows the code coverage percentage obtained when using stock JaCoCo.
Figure 9. code coverage percentage with methods filtering enabled

Figure 10 shows another 7% improvement in code coverage. Code coverage is improved by 9% for the case of NQueensDist.

Note that the static fields changes has not committed by X10 core team. It will show greater improvement over the current code coverage percentage.

**Analysis of Test Suite**

Initial analysis shows that 37% of the instructions in X10.jar are covered. Branch coverage for the whole test suite is at 29%. Surprisingly, some of the fundamentals constructs in X10 has coverage of less than 50%. “Place” and “FinishState” class in x10.lang is only 37% covered. Serialization is also important and widely used in X10, but the package is only 42% covered.

By crosschecking with coverage data obtained by running the samples in x10.dist/samples, some of the classes used are not covered at all in the X10 test suite. These classes include Option and OptionsParser in x10.util. Team is also used in the samples, but only 13% covered by X10 test suite.
Flaws of the software and Future Work

Method Filtering
The algorithm to filter out methods at the moment is not modular and hard-coded to filter out methods that start with "$" sign. JaCoCo core team is currently putting together ideas for filtering classes and certain source code. They have been contacted regarding the idea to filter out methods. If this proposal gets implemented, it would support the x10 code coverage tool. For now, the code coverage tool will have to use the current implementation of filtering methods.

Bad Assumption
1 Java source file points to 1 X10 source file. This is not the case in real life. Need to be fixed. At the moment, the codes not corresponds to x10 source file with the same name is ignored.

To fix this, mapping of generated Java code to X10 code will need to be done before generating the report. This would require more memory usage to store a much larger mapping array, compared to single Java source to X10 source mapping. This was not done due to time constraint and the effort required to redesign the algorithm.

Methods Combining
The algorithm to combine methods is not perfect. Works on most cases, but in some case, method in generated Java source code corresponds to a few lines very far apart. This is caused by method inlining. When X10 compiler optimization is enabled, inlining happens. For example, in

```x10.lan runge/Probe() , Line 669 in Runtime.X10 is inlined to line 1284 in Runtime.X10.
```

```java
//#line 1283 "/home/josh/x10-trunk/x10.runtime/src-
x10/x10/lang/Runtime.x10"
public static void
probe(
    
    //#line 669 . "/home/josh/x10-trunk/x10.runtime/src-
x10/x10/lang/Runtime.x10"
    final x10.core.Thread t82865 =
        x10.core.Thread.currentThread();

    //#line 669 . "/home/josh/x10-trunk/x10.runtime/src-
x10/x10/lang/Runtime.x10"
    final x10.lang.Runtime.Worker t82866 =
        x10.rtt.Types.<x10.lang.Runtime.Worker>
        cast(t82865,x10.lang.Runtime.Worker.$RTT);

    //#line 1284 "/home/josh/x10-trunk/x10.runtime/src-
x10/x10/lang/Runtime.x10"
    t82866.probe();
    ```
To fix this issue, one has to have in depth knowledge of how X10 compiler works. This is out of the scope for me. A workaround for this issue is to not enable optimization. In `x10.runtime/build.xml`, Compiler optimization is disabled by default.

**Core JaCoCo Issues**
JaCoCo ignore `try{}` statements in calculating code coverage. As a result, this causes inconsistency in mapping of Java source code to X10 source code. Last line’s number is less than first line’s number for a X10 method. In this case, the first line’s number will be the smallest of the two line number and the last line’s number with be the largest number of the two. This will cause a problem in combining methods. The original X10 method name might not be shown, but some other internal’s method might be shown due to this after combining the methods.

Temporary workaround for this is if lastline number is smaller than firstline number, replace the firstline number with lastline number for the method.

One way to fix this is that X10 compiler mark each `try{}` with certain comments. My algorithm can scan for these comment and ignore the `try` clause. Again, these solution requires knowledge of how X10 compiler works and is therefore shelved for future work.

Also, JaCoCo displays additional classes generated. They have signature of class$new <stuff> (exception()/object(), thread() etc ) and is irrelevant and contains no code. If the filtering proposal above is implemented, we can filter these classes out.

**Missed Methods do not match**
Methods missed and cyclomatic complexity missed do not add up to the total count. This is caused by the way JaCoCo determine a missed method. A method is considered missed when no instructions that correspond to that particular method are executed. When combining methods together, the instructions coverage information are combined together. Therefore, unless both methods are missed methods, the combined method becomes not missed. Cyclomatic complexity is related to method covered and missed count. Therefore, this bug applies to cyclomatic complexity for now. The difficulty of this is beyond the scope, and this bug is shelved for future work.

**Class Combining**
Total class count does not match. Again, if classes combining algorithm is done in the highest layer, it will be fixed. This is a minor issue, but requires a lot of effort to it, not a high priority, so not fixed.

**X10 Constructors**
X10 constructor generate a pair of Java constructors and field initializer structure. The Java constructor (1 phase constructor)is not called but is needed for interoperability between X10 and Java. The constructor with field initializer is used instead. The unused constructor means reduction in code coverage percentage. This will be a limitation of the code coverage tool. In the future, these unused constructors could be filtered out if the filtering proposal above is implemented.
**Broken core JaCoCo tests**
JaCoCo maintains integrity by running a set of tests on compilation. By modifying JaCoCo, the tests are broken as the output report format has been modified. At the moment, these tests are skipped when compiling. This can be easily fixed by modifying the tests for the checking the report format. Fixing the tests would require extensive understanding of the written tests in order to modify them. It is deemed too much of an effort to fix the tests in addition to that this is not a priority issue, thus this is shelved for future work.

**Modified JaCoCo Requires Patch**
JaCoCo core team will be contacted for a request for this modification to be added to official JaCoCo. X10 code coverage can be enabled by using additional argument.
Conclusion
This project developed a code coverage tool for the X10 programming language based on the existing JaCoCo tool for Java. The modified JaCoCo is able to output X10 source code rather than the generated Java code. The new improved code coverage tool improves on code coverage over stock JaCoCo when used on X10 source code. Readability is greatly improved and will be useful to all X10 developers. This would also help the X10 community as users are now able to generate a code coverage report for the X10 test suite. Last but not least, the runtime overhead of running the tool is minimal.

Although there is still room for improvements for this software, this code coverage tool would be useful for the X10 development team and the broader X10 community.
Bibliography


Appendix I
Hello World in X10

/*
 * This file is part of the X10 project (http://x10-lang.org).
 * This file is licensed to You under the Eclipse Public License (EPL);
 * You may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 * http://www.opensource.org/licenses/eclipse-1.0.php
 * (C) Copyright IBM Corporation 2006-2010.
 */

import x10.io.Console;

/**
 * The classic hello world program, shows how to output to the console.
 */
class HelloWorld {
    public static def main(Rail[String]) {
        Console.OUT.println("Hello World!");
    }
}

Generated Java source code for HelloWorld

Compiling the above code with “x10c” command will generate the following code. The indentation of the generated Java source code is not changed in any way.

@x10.runtime.impl.Java.X10Generated public class HelloWorld extends x10.core.Ref implements x10.serialization.X10JavaSerializable {
    public static final x10.rtt.RuntimeType<HelloWorld> $RTT =
        x10.rtt.NamedType.<HelloWorld> make("HelloWorld", HelloWorld.class);
    public x10.rtt.RuntimeType<?> $getAsObject() {return $RTT;}
    public static x10.serialization.X10JavaSerializable
     $deserialize_body(HelloWorld $_obj,
    x10.serialization.X10JavaDeserializer $deserializer) throws
    Java.io.IOException {
        if (x10.runtime.impl.Java.Runtime.TRACE_SER) {
            x10.runtime.impl.Java.Runtime.printTraceMessage("X10JavaSerializable:
             $deserialize_body() of " + HelloWorld.class + " calling");
        }
        return $_obj;
    }
}
public static x10.serialization.X10JavaSerializable
$_deserializer(x10.serialization.X10JavaDeserializer $deserializer) throws Java.io.IOException {
    HelloWorld $_obj = new HelloWorld((Java.lang.System[]) null);
    $deserializer.record_reference($_obj);
    return $_deserialize_body($_obj, $deserializer);
}

public void $_serialize(x10.serialization.X10JavaSerializer $serializer) throws Java.io.IOException {
}

// constructor just for allocation
public HelloWorld(final Java.lang.System[] $dummy) {
}

//#line 18 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
public static class $Main extends x10.runtime.impl.Java.Runtime {
    private static final long serialVersionUID = 1L;
    public static void main(java.lang.String[] args) {
        // start native runtime
        new $Main().start(args);
    }

    // called by native runtime inside main x10 thread
    public void runtimeCallback(final x10.core.Rail<java.lang.String> args) {
        // call the original app-main method
        HelloWorld.main(args);
    }

    // the original app-main method
    public static void main(final x10.core.Rail<java.lang.String> id$0) {

        //#line 19 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
        final x10.io.Printer t11 = 
            ((x10.io.Printer)(x10.io.Console.get$OUT()));

        //#line 19 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
        t11.println(((java.lang.Object)("Hello World!")));
    }

    //#line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
    final public HelloWorld
HelloWorld$$this$$HelloWorld(
}

// line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
return HelloWorld.this;
}

// line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
// creation method for Java code (1-phase Java constructor)
public HelloWorld(){this((Java.lang.System[]) null);
HelloWorld$$init$$S();}

// constructor for non-virtual call
final public HelloWorld HelloWorld$$init$$S() { }

// line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
;

// line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"

// line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
this.__fieldInitializers_HelloWorld();
    }
return this;
}

// line 17 "/home/u4788232/x10-trunk/x10.dist/samples/HelloWorld.x10"
final public void
__fieldInitializers_HelloWorld(
{}
)}
Appendix 2
This segment of code are part of NQueensDist.X10 and NQueensDist.Java.

Declaring public static val EXPECTED_SOLUTIONS = [0, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, 365596, 2279184, 14772512]; in NQueensDist.X10 will generate this segment of code in NQueensDist.Java.

```
public static x10.core.Rail
get$EXPECTED_SOLUTIONS()
{
    if (((int) NQueensDist.initStatus$EXPECTED_SOLUTIONS.get()) ==
        ((int) x10.runtime.impl.Java.InitDispatcher.INITIALIZED)) {
        return NQueensDist.EXPECTED_SOLUTIONS;
    }
    if (((int) NQueensDist.initStatus$EXPECTED_SOLUTIONS.get()) ==
        ((int) x10.runtime.impl.Java.InitDispatcher.EXCEPTION_RAISED)) {
        if (((boolean) x10.runtime.impl.Java.InitDispatcher.TRACE_STATIC_INIT) ==
            ((boolean) true)) {
            x10.runtime.impl.Java.InitDispatcher.printStaticInitMessage(
                ((Java.lang.String)("Rethrowing ExceptionInInitializer for field:
                        NQueensDist.EXPECTED_SOLUTIONS")));
            throw NQueensDist.exception$EXPECTED_SOLUTIONS;
        }
    }
    if (NQueensDist.initStatus$EXPECTED_SOLUTIONS.compareAndSet((int)(x10.runtime.impl.Java.InitDispatcher.UNINITIALIZED),
        (int)(x10.runtime.impl.Java.InitDispatcher.INITIALIZING))) {
        try {
            NQueensDist.EXPECTED_SOLUTIONS =
                (x10.core.Rail) (x10.runtime.impl.Java.ArrayUtils.<x10.core.Long>
                    makeRailFromJavaArray(x10.rtt.Types.LONG, new long[]{0L, 1L, 0L, 0L, 2L,
                        10L, 4L, 40L, 92L, 352L, 724L, 2680L, 14200L, 73712L,
                        365596L, 2279184L, 14772512L}));
        } catch (Java.lang.Throwable exc$12024) {
            NQueensDist.exception$EXPECTED_SOLUTIONS = new
                x10.lang.ExceptionInInitializer(exc$12024);
            NQueensDist.initStatus$EXPECTED_SOLUTIONS.set((int)(x10.runtime.impl.Java.
                InitDispatcher.EXCEPTION_RAISED));
        }
    x10.runtime.impl.Java.InitDispatcher.lockInitialized();
x10.runtime.impl.Java.InitDispatcher.notifyInitialized();
    throw NQueensDist.exception$EXPECTED_SOLUTIONS;
```
if (((boolean) x10.runtime.impl.Java.InitDispatcher.TRACE_STATIC_INIT) == ((boolean) true)) {
    x10.runtime.impl.Java.InitDispatcher.printStaticInitMessage(((Java.lang.String)("Doing static initialization for field: NQueensDist.EXPECTED_SOLUTIONS"));
}
NQueensDist.initStatus$EXPECTED_SOLUTIONS.set((int)(x10.runtime.impl.Java.InitDispatcher.INITIALIZED));
x10.runtime.impl.Java.InitDispatcher.lockInitialized();
x10.runtime.impl.Java.InitDispatcher.notifyInitialized();
} else {
    if (NQueensDist.initStatus$EXPECTED_SOLUTIONS.get() < x10.runtime.impl.Java.InitDispatcher.INITIALIZED) {
        x10.runtime.impl.Java.InitDispatcher.lockInitialized();
        while (NQueensDist.initStatus$EXPECTED_SOLUTIONS.get() < x10.runtime.impl.Java.InitDispatcher.INITIALIZED) {
            x10.runtime.impl.Java.InitDispatcher.awaitInitialized();
        }
        x10.runtime.impl.Java.InitDispatcher.unlockInitialized();
        if (((int) NQueensDist.initStatus$EXPECTED_SOLUTIONS.get()) == ((int) x10.runtime.impl.Java.InitDispatcher.EXCEPTION_RAISED)) {
            if (((boolean) x10.runtime.impl.Java.InitDispatcher.TRACE_STATIC_INIT) == ((boolean) true)) {
                x10.runtime.impl.Java.InitDispatcher.printStaticInitMessage(((Java.lang.String)("Rethrowing ExceptionInInitializer for field: NQueensDist.EXPECTED_SOLUTIONS"));
            }
            throw NQueensDist.exception$EXPECTED_SOLUTIONS;
        }
    }
    return NQueensDist.EXPECTED_SOLUTIONS;
<table>
<thead>
<tr>
<th>Class</th>
<th>Page output by stock Jacoco</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The table appears to be incomplete and lacks clear column headers. The content seems to be related to class page output by stock Jacoco, which typically includes metrics and coverage data from software testing tools.
<table>
<thead>
<tr>
<th>Class</th>
<th>Test #1</th>
<th>Test #2</th>
<th>Test #3</th>
<th>Test #4</th>
<th>Test #5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassPage output by modified JaCoCo</td>
<td>0.33</td>
<td>1.35</td>
<td>2.06</td>
<td>1.1</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.95</td>
<td>1</td>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1.45</td>
<td>2</td>
<td>12</td>
<td>0.06</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1.6</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0.45</td>
<td>1</td>
<td>2</td>
<td>0.08</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0.7</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0.7</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0.7</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Revealed
Appendix 4
PackagePage output by stock Jacoco. The unmangled names are too long to fix into one page.