Stochastic Performance Logic User Manual

Haas František
frantisek.haas@gmail.com

Lacina Martin
lacina.martin@gmail.com

Kotrč Jaroslav
kotrcj@gmail.com

March 20, 2013

Contents

Introduction

Stochastic Performance Logic project is a set of tools for measuring and comparison of performance of Java code. This project is based on **"Capturing Performance Assumptions using Stochastic Performance Logic"** [1].

The idea behind SPL paper is to reason about functions performance using *performance relations* between them. These assumptions are declared in source files in form of Java annotations. This makes it easier to keep these relations up to date.

It's possible to compare for example different implementations of an interface on performance with specific sizes of input data. Examples follows

```
for i in (1000, 5000, 9000) holds: encrypt(i) < 3*memcpy(i)
for i in (1000, 2000, 3000) holds: bubblesort(i) > quicksort(i)
```

The interpretation of the examples shows the power of **SPL**. The first example expresses that encrypt function is very fast. And the seconds example states that quick sort outperforms bubble sort.

Performance regression testing is an important but usually overlooked part of most projects' testing suites. Most notably, in long term projects where functions or modules are regularly updated, improved, more secured or fixed a performance penalty may gradually rise unnoticed. What more, some updates that should improve performance do actually worsen the running time. For example changing implementations of heavily used data structures without thorough testing may slow the code. There are examples of such phenomena in our case study. See [Dark side of updates on page 53]).

Nevertheless expressing performance relations between functions is not only possible in a current version of a single project but also across revision history of Git and Subversion systems or even across history of various projects with different repositories. This allows comparing performance of libraries with similar interface but much more importantly it allows regression testing. For example

```
for i in (..) holds: encrypt_v1(i) >= encrypt_v2(i) >= ... >= encrypt_HEAD(i)
```

In this case the encrypt function is highly utilized in the core of an application (found out probably by profiler) and therefore its performance must be supervised and kept in the same range or improved only.

SPL tools are designed for this task and it's therefore much smarter and easier to use them for performance and regression testing than a single purpose code. Tools simplify measuring of running times, evaluation of measured data and presentation of results. And can therefore save much time for improving the code rather than testing if anything has been improved or worsen.

We consider **SPL tools** as a **performance equivalent** to **JUnit** [2]. What more, using it does not bring almost any dependencies to the project. Only a single class file declaring

[1] The article "Capturing Performance Assumptions using Stochastic Performance Logic" can be found on each of the following URLs:
`http://dx.doi.org/10.1145/2188286.2188345`
`http://dl.acm.org/citation.cfm?id=2188345`
`http://d3s.mff.cuni.cz/publications/`

[2] JUnit web page `http://junit.org/`

structure of SPL annotation must be present.

SPL tools are composed of a **command line** utility the offers complete functionality for processing annotated projects.

Then there's an **Eclipse plugin** that has various features and guides user from creation of the basic SPL project files onto syntax verifying, configuration guidance, measurement control and ends with presentation of performance results.

And to complete the functionality there's a **Hudson plugin** to evaluate SPL formulas. The results may be viewed directly from published HTML output or accessed via Eclipse plugin remotely.

License agreements

1. Annotations

Annotations are used to describe what methods are measured and what is the input for them. The annotations definition is:

```
package cz.cuni.mff.spl;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

/** Main SPL annotation. */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SPL {
   /** SPL formula declarations. */
   String[] formula() default {};
   /**  Local method aliases declarations. */
   String[] methods() default {};
   /** Local generator aliases declarations. */
   String[] generators() default {};
}
```

Textbooks often use sorting as examples and this one is not an exception. On the next example is shown simple annotation:

```
@SPL(
    generators = {
        "genInt=SPL:spl.IntegerUniform('10;50')#generate()"
    },
    methods = {
        "libSort=java.util.Arrays#sort(int[])"
    },
    formula = {
        "SELF[genInt](5,10) <= (1, 1.5) libSort[genInt](5,10)"
    }
)
public void mySort(int[] arr){
    // some logic
}
```

The meaning of the annotation is that the method **mySort** run takes at most 1.5 longer than run of the library method **java.util.Arrays#sort(int[])**.

This annotation defines one generator alias in field **generators**, one method alias in field **methods** and one formula in field **formula**. There can be any number of generator and method aliases and formulas in the fields of the annotation.

Generator alias **genInt** refers to integrated generator of random integer numbers configured to return values between **10** and **50**. Method alias **libSort** refers to Java method for sorting

integer array. The formula compares method **mySort** referred as **THIS** with generator **genInt** as input with the library sorting method referred as **libSort** with the same generator as input. In one measurement cycle the method will run **5**-times (that's the number of inputs created by generator) on array with **10** items.

Before comparison the run time of the left side is multiplied by **1** and the run time of the right side is multiplied by **1.5** so the result should be that left side runs at most **1.5** slower than right side.

1.1 Grammar

The grammar used for writing annotations is described in this section. It has three keywords:

THIS
: Is alias for the project where annotations are written. Also it is the project for which is the XML configuration file written.

HEAD
: Is alias for the most current revision of the repository where the revision belongs.

SELF
: Is alias which refers to the method near which is the annotation written.

THIS at revision **HEAD** is the only location where framework searches for annotations. Annotations written elsewhere are ignored.

1.1.1 Formulas

In formulas can be used full method or generator definition or aliases defined in the same annotation or global aliases defined in XML configuration file which is described further in section [XML configuration file on page 23]. Formula consists of optional variables declaration followed by one or more comparisons joined by logical operators.

Variables

Variables are used as input for generators or as a part of lambda expression. It is optional part or the formula and can have only non-negative integer values. For better comprehension formula part after variable declaration is simplified in this section. There are two kinds of variables.

Simple variables

```
for( i{1, 2} j{3, 4} ) A[G](i) < B[G](j)
```

There can be any number of variables and their values. Variable is declared by writing its name followed by its values in curly brackets **{1,2}** for variable **i** and **{3,4}** for variable **j**. The values are separated by comma. The result is that formula is expanded and for every value is created new formula part linked by logical AND. The variables are combined as a Cartesian product.

This example is expanded to formula:

```
A[G](1) < B[G](3) &
A[G](1) < B[G](4) &
A[G](2) < B[G](3) &
A[G](2) < B[G](4)
```

Sequence variables

```
for( (a, b){ (1, 2), (3, 4) }) A[G](a) < B[G](b)
```

It consist of variables names in parenthesis **(a,b)** followed by their values in curly brackets **{(1,2), (3,4)}**. Each value is written in parenthesis and represents one set of values used for one measurement. There is defined one sequence variable **(a,b)** with two sub variables **a** and **b** followed by two values of the variable. The value contains one sub value for every sub variable.

The example is expanded to formula:

```
A[G](1) < B[G](2) &
A[G](3) < B[G](4)
```

Its difference from simple variables is that this variable doesn't combine its values as a Cartesian product.

The number of sub variables and number of the sub values must be the same. For example

```
for( (a, b){(1, 2, 3)} )
```

is invalid because there are only two sub variables but three sub values in the definition.

Both variable types can be combined and the result is a Cartesian product of the values.

```
for( (a, b) (1, 2), (3, 4)  c5,6) A[G](a, c) < B[G](b, c)
```

This is equivalent to four set of values:

- a=1, b=2, c=5
- a=1, b=2, c=6
- a=3, b=4, c=5
- a=3, b=4, c=6

And the example is expanded to formula:

```
A[G](1, 5) < B[G](2, 5) &
A[G](1, 6) < B[G](2, 6) &
A[G](3, 5) < B[G](4, 5) &
A[G](3, 6) < B[G](4, 6)
```

Comparison

After variables declaration there is at least one comparison. Multiple comparisons are combined by conjunction (written as &), disjunction (|) or implication (==> ordered by descending priority. Parenthesis () can be used to enforce different priority.

Single comparison consist of two methods each with one generator compared by comparison operator and optional lambda expression.

```
SELF[pkg.GenClass1] (i, 3) <= pkg.ClassName#method[pkg.GenClass2]
```

Method **SELF** with generator **pkg.GenClass1** as its input is compared with method **pkg.ClassName#method** with the generator **pkg.GenClass2** as its input. The variable **i** placed in parenthesis **()** after method on the left side is used as input for generator **pkg.GenClass1** and it must be declared as a variable before formula (in the part **for(...)**) or as a parameter in the XML configuration file. Also non-negative integer constants (**3** in this example) can be used. There can be any number of variables and constants separated by comma. On the right side no variables are used so the parenthesis have to be omitted.

Lambda expression

Lambda expression can be used after comparing sign in the comparison. It is written in parenthesis **()** and has two parts separated by comma. Each part contains one or more real number constants, variables defined in **for(...)** part or parameters defined in XML configuration file separated by asterisk *****. The result value of the lambda part is computed as a product of all constants, variables and parameters used in the part.

The meaning is that measured times of left side of the comparison are multiplied by the result value of the left part of the lambda expression and measured times of the right side of the comparison are multiplied by the result value of the right part of the lambda expression before the times are compared. For example:

```
pkg.Sort#quicksort[pkg.IntGen](1000) <=
(1, 0.8*THREADS) pkg.Sort#parallelQui[pkg.IntGen](1000)
```

It means that before comparison is made measured times of **pkg.Sort#quicksort** method with **pkg.IntGen** generator is multiplied by **1** and measured times of **pkg.Sort#parallelQui** with **pkg.IntGen** is multiplied by product of constant **0.8** and the value of **THREADS** which has to be defined as a variable or parameter.

Comparison operators

Comparison operators defines relation which should be met between measured times of both sides of the comparison. There can be used operators lesser ($<$), lesser or equal ($<=$), exact equal ($==$), greater or equal ($>=$) and greater ($>$).

There is also short cut $=$ for equality comparing with default 5% tolerance interval. It means that execution times of the comparison operands satisfy following relation in the statistical t-test check for both combinations of comparison operand measured values assigned to $time_1$ and $time_2$ variables:

$0.95 * time_1 < 1.05 * time_2$

This equality short cut simply means, that the comparison operands are almost same and very close to each other.

Two special lambda expressions can be used for this operator to specify how big the tolerance interval is.

First expression type has two parts like the normal lambda expression followed by real number separated by comma which defines the interval. It looks like this:

```
pkg.Sort#quicksort[pkg.IntGen](1000)(i, 3) =
(1, 0.8*THREADS, 0.1) pkg.Sort#parallelQui[pkg.IntGen](1000)
```

There is defined lambda expression which compares left side multiplied by **1** and right side multiplied by **0.8*THREADS** for equality with 10 % tolerance interval.

Second expression type omits parts of the normal lambda expression and consists only from real number defining the interval. It looks like this:

```
pkg.Sort#quicksort[pkg.IntGen](1000)(i, 3) =
(0.1) pkg.Sort#parallelQui[pkg.IntGen](1000)
```

There is defined lambda expression which compares left and right side without any multiplication for equality with 10 % tolerance interval.

1.1.2 Method aliases

Method alias is a single string short cut used to refer to whole method declaration in a formula. Its definition is:

```
alias =
    project@revision: full.class.Name('parameter')
    # method ( type1 name1, type2 name2, ... )
```

It means:

alias
: Is the name of defined alias. It is used to refer to the method in formulas.

project
: Is the name of the project from which came the method. If omitted then **THIS** project is used instead.

revision
: Is the name of the revision of specified project where the method came from. If omitted (@ is omitted too) then **HEAD** revision is used instead. If project and revision name is omitted then colon `:` is omitted too and it is the same as

  `THIS@HEAD:`

full.class.Name('parameter')
: Is the full name of the method class. There are two kinds of declaration:

  Name()
  : Can be used for static or member type and if it is member type then the class must have constructor without parameters. Parenthesis () are optional and have no meaning.

  Name('parameter')
  : Is used for member method which class have constructor with one string parameter for which **parameter** value is used.

method
: Is the name of measured method after which method parameter types can be optionally

specified. If types are not specified and the method is overloaded then an error is issued as it is not possible to determine which prarticular method to use.. If types (**type1** and **type2** in the example) are specified they are written after method name in parenthesis separated by commas and they determine the method. After parameter type there can be optional parameter name (**name1** and **name2** in the example) but it is not used by the framework.

This is simple example:

```
staticMethod =
    cz.cuni.mff.spl.IntegerSumCalculator#calculateSumOfIntegers
```

There is defined alias **staticMethod** separated by equals sign = from method declaration. It may refer to method:

```
package cz.cuni.mff.spl.IntegerSumCalculator;

class IntegerSumCalculator{

    // method can have any return type
    public static int calculateSumOfIntegers( /* input parameters...*/){
        // logic
    }

}
```

The method doesn't need to be static but then the class must have constructor without parameters:

```
package cz.cuni.mff.spl.IntegerSumCalculator;

class IntegerSumCalculator{

    public IntegerSumCalculator(){
        //logic
    }

    // method can have any return type
    public int calculateSumOfIntegers( /* input parameters...*/){
        // logic
    }
}
```

Method name is separated by hash sign # from the class name. If the method is overloaded then the correct method is chosen by the return type of the generator used as method input in a formula.

More complex example is:

```
staticMethod =
    THIS@HEAD:
    cz.cuni.mff.spl.IntegerSumCalculator()#calculateSumOfIntegers(int i)
```

Although it looks different its meaning is the same as the definition in previous example. Project and revision **THIS@HEAD:** can be omitted, parenthesis after class name **Inte-**

gerSumCalculator() are ignored because no parameter is present and parameter type specified after method name **calculateSumOfIntegers(int i)** can be omitted too.

To use method from other project at specific revision it is first needed to configure that project in XML configuration file. Lets assume that project named **examples** has revision named **init** with the same method as in previous example. Alias for this method is:

```
staticMethod =
    examples@init:
    cz.cuni.mff.spl.IntegerSumCalculator#calculateSumOfIntegers
```

Another examples are described in section [Detailed look at the measurements on page 43].

1.1.3 Generator aliases

Generator alias is a single string short cut used to refer to whole generator declaration in a formula. Its definition is:

```
alias =
    project@revision: full.class.Name('parameter1')
    # method ('parameter2')
```

The first part of the definition is the same as for method alias. The difference is in parameters and method specification:

full.class.Name('parameter1')
: Is the full name of the generator class. There are two kinds of declaration:

  Name()
  : It is used when no parameters are needed for constructor of generator class. For class type of generator it means that the constructor has no parameters. For method type of generator it means that the method is static or the class instance is created by constructor without parameters for member method.

  Name('parameter1')
  : In this case generator class is created by constructor with one string parameter. For class type of generator is the created instance used. For method type generator is used member method of the instance.

method ('parameter2')
: This method specification is used only for method type of generator. Method **method** is called to obtain values from generator. **parameter2** is optional and is used as input for **method**. For class type of generator the last part of alias declaration after hash # is omitted.

This is simple example:

```
intGen = cz.cuni.mff.spl.test.IntegerGenerator('10')#generate()
```

There is defined alias **intGen** separated by equals sign = from method generator declaration. It refers to generator which returns arrays of random integer values. The generator code is:

```
package cz.cuni.mff.spl.test;

import java.util.ArrayList;
import java.util.Random;

public class IntegerGenerator {

    private int max;

    /**
     * Create instance and parse argument to get maximum
     * of generated integers.
     *
     * @param arg
     *         maximum of generated integers
     */
    public IntegerGenerator(String arg){
        max = Integer.parseInt(arg);
    }

    /**
     * Returns list with one array of Objects.
     * In the array is stored one array of variable count
     * of random integers.
     * The array of Objects is used as input for single method call
     * so the method must have one integer array as argument.
     *
     * @param count
     *         Size of the array of integers.
     * @return
     *         List with one array of Objects with one array of integers.
     */
    public ArrayList<Object[]>  generate(int count){
        Random rnd = new Random();
        ArrayList<Object[]>  result = new ArrayList<Object[]>();
        int[] arr = new int[count];
        for(int idx = 0; idx < count; ++idx){
            arr[idx] = rnd.nextInt(max);
        }
        Object[] obj = new Object[1];
        obj[0] = arr;
        result.add(obj);
```

```
        return result;
    }
}
```

In the generator constructor is parsed maximum value of generated integers so the measurement using alias **intGen** will always get values lesser than **10**.

The alias usage is:

```
pkg.Sort#quicksort[intGen](1000) > pkg.Sort#quicksort[intGen](900)
```

In the comparison is used the same generator but with different integer parameters. The left side generator will return array of **1000** items and the right side generator will return array of **900** items but both generators return values lesser than **10**.

The generator in the example returns one array of arguments so the method will run only once in single measurement cycle. To extend it to return variable count of arrays of arguments this method is added:

```
/**
 * Returns list with variable count of arrays of Objects.
 * In the array is stored one array of variable count
 * of random integers.
 * Every array of Objects is used as input for single method
 * call so the method must have one integer array as an argument
 * and the method will be called once for every array.
 *
 * @param arrays
 *         Number of arrays of Objects generated
 * @param count
 *         Size of the array of integers.
 * @return
 *         List with variable count of arrays of Objects with one
 *         array of integers.
 */
public ArrayList<Object[]>  generate(int arrays, int count){
    Random rnd = new Random();
    ArrayList<Object[]>  result = new ArrayList<Object[]>();
    for(int arrIdx = 0; arrIdx < arrays; ++arrIdx){
        int[] arr = new int[count];
        for(int idx = 0; idx < count; ++idx){
            arr[idx] = rnd.nextInt(max);
        }
        Object[] obj = new Object[1];
        obj[0] = arr;
        result.add(obj);
    }
    return result;
}
```

This method has two integer parameters as its input so the usage is:

```
pkg.Sort#quicksort[intGen](5, 1000) > pkg.Sort#quicksort[intGen](5, 900)
```

In the comparison is used the same alias but with different number of parameters so the new method is used as the generator. It will run **5**-times for every measurement cycle because **5** arrays of arguments are generated.

Another examples are described in section [Generators on page 41].

1.2 Generators

Generator creates input for methods during measurements.

1.2.1 Generator description

The generator generates data of the following type:

```
Iterable<Object[]> data;
```

That said a single call of the generator creates data for multiple calls (`Iterable<>`) of a function with possibly multiple arguments (`Object[]`). The items of the array of arguments must be of correct types just the way like when the method is called in code. The number of method calls during single measurement cycle is determined by the number of items in the **data** container and number of measurement cycles is determined in the INI configuration file.

The data created by generator for one method run corresponds to the input of method **java.lang.reflect.Method.invoke(Object obj, Object... args)**.

Generator input can be one of this:

- without parameters
- one string parameter
- any number of integer parameters
- one string parameter followed by any number of integer parameters

String parameters are written in generator declaration as a string value. If generator alias is declared then the parameter is part of the alias declaration and is the same in every formula where the alias is used.

Variable count of integer parameters are written in formula next to the method where generator is used as input. It is not part of generator declaration so one alias can be used in formulas with different values of the parameters or even with different count of integer parameters. As integer parameters can be used integer constants, variables defined in **for(...)** part of the formula or parameters defined in XML configuration file.

There are two types of generators:

Class generator
: Generator class must implement `Iterable<Object[]>` itself so it can be used as the data container. Which constructor is used for class instantiating depends which parameters are used in generator declaration and its usage in formulas.

Method generator
: Generator is method which must return `Iterable<Object[]>`. It can be static or

member method. If member method is used then the instance of method class is created by constructor without parameters or with single string parameter depending if parameter is used next to class name in generator declaration. The parameter for constructor is different from the one in generator input. Whole generator input is passed to the method defined as generator.

For example if generator alias

```
gen = cz.cuni.mff.spl.MethodGenerator('paramConstr')
    # generate('paramMethod')
```

is used in formula

```
SELF[gen] = SELF[gen](1,2)
```

then for the left side will be created instance of **cz.cuni.mff.spl.MethodGenerator** with constructor with one string parameter for which **paramConstr** will be used as its input. On this instance will be called method **generate(String)** with **paramMethod** as its input. For the right side will be created the class instance the same way but method **generate(String, int, int)** will be called on it with **(paramMethod, 1, 2)** as its input.

1.2.2 Integrated generators

There are integrated generators that can be used directly within annotations without writing any code. They generate arrays, arrays of primitives and containers implementing iterable, list and collection interface.

There's a class for each value type and distribution. Container type and distribution parameters are declared as semicolon separated string values in the constructor. Possible container values are *array*, *arrayPrimitive*, *list* and default value is primitive array. Number of calls to generate and number of elements in the container per call is specified in the generator method.

To declare a generator specify following.

```
SPL:spl.IntegerUniform('lowerBound;upperBound;argumentType')#generate()
SPL:spl.IntegerPermutation('argumentType')#generate()

SPL:spl.DoubleExponential('lambda;argumentType')#generate()
SPL:spl.DoubleGaussian('mu;sigma;argumentType')#generate()
SPL:spl.DoubleUniform('lowerBound;upperBound;argumentType')#generate()

SPL:spl.LongUniform('lowerBound;upperBound;argumentType')#generate()
```

To actually use the generator number of calls and element count must be specified. Constants on each side may differ.

```
sort1[generator](calls,elements) = sort2[generator](calls,elements)
```

1.3 Practical advices

Writing annotations can be a little tricky on the first sight so there are few advices how to write and maintain them.

1.3.1 Writing annotations

It is a good idea to use aliases. Global aliases written in the XML configuration file can be used in every formula of the project. Global generator aliases are useful if there are only few generators used for measurement. Global method aliases are useful for a reference method.

Aliases are useful even if they are used only locally. They make the formula shorter and better readable.

Annotations can be written next to method used for measurement. It gives possibility to use **SELF** to refer this method. However that is not necessary and in some cases it might be better to have the annotation in a separate source.

Annotations placed in a separate file may add some readability because there is not mixed Java and SPL grammar. It is also possible to have different set of annotations in separate files and configure the framework to measure just selected part which can be single package or even single class.

It is useful during debugging new annotations. They can be written in a separate class and if framework is configured to measure just this class it is faster than if all annotations are measured.

It is also possible to have different groups of annotations. For example one group is measured every night and other is measured just once in a week because the measurement takes much time. To differ this cases it is needed only to change scan patterns in XML configuration file.

Annotations placed in out of Java source can be placed to different source tree and it gives possibility to compile business logic without annotations.

1.3.2 Writing generators

Maybe the simplest way of writing own generator is to extend some iterable container for example `ArrayList<Object[]>` and in its constructor fill it with arrays of objects that represents parameters for measured method single call by using `this.add(objectArray)`.

There is simple generator example that can be used for every method that has single integer parameter:

```
public class IntGenerator extends java.util.ArrayList<Object[]>{
    public IntGenerator(){
        java.util.Random rnd = new java.util.Random();
        Object[] result = new Object[1];
        result[0] = rnd.nextInt();
        this.add(result);
    }
```

```
}
```

This generator is of class type without any parameters. In the constructor it creates just one array of arguments so the measured method will be called just once in every measurement cycle and single random integer value will be used as the method input.

For more configurable generator string parameters can be used. It can be parsed in the generator so almost any information can be passed to the generator. For method type generator there can be actually used two string parameters. First for constructor of the method class and second for the method itself.

Also integer parameters are useful. They can be easily used in formula with many values and the framework will automatically expand them. This way it is easy to write formula for measuring the same method with many generator configurations.

It is a good practise to store generators in some repository and to use fixed revision for their declaration. If they are stored in local folder they cannot have revisions thus for every framework run the formulas that use them are measured again. If generators from project with repository are used then measurements can be cached and if the same revision of method and generator is used again in other framework run the cached data can be used. The measurement won't run again until some change in revision of method or generator is made.

However if generators are stored along with the annotations in some kind of Git repository where hash code of the commit is used to identify revision it is needed to make change in generators in two commits. First commit changes the generator sources and second commit have to change the identification of the revision from which generators are used.

2. Command Line Tool

2.1 Overview

The *SPL Tools Framework* can be used as a command line tool, an Eclipse plugin or a Hudson plugin. The command line interface is the basic way to use the *Framework*. It allows complete processing of annotated projects. Starting from obtaining annotated project trough scanning annotations, code generation, execution to the evaluation and result generation.

2.1.1 Requirements

Requires **Java Development Kit 1.7** (JDK7)[1][2] or higher.

Compilation from sources requires **Apache Ant 1.8**[3] or higher and **Git**.

Remote execution machine must have installed **Java 7** and **SSH server**.

2.1.2 Installation

The *Framework* is distributed as a prepared binary package.

The distribution package can be downloaded from the following URL:

`http://sourceforge.net/projects/spl-tools/files/release/spl-latest.zip`

However in some cases in the following text we will suppose that the *Framework* source repository has been checked out and that some additional files (like simple examples) are present and accessible.

2.1.3 Compilation from source

To compile the *Framework* from source code follow these steps:

1. Obtain the source from the repository on following URL

   ```
   git://git.code.sf.net/p/spl-tools/code
   http://git.code.sf.net/p/spl-tools/code
   ```

2. Open shell in the directory where the code has been checked out and run[4]

   ```
   ant bin-dist
   ```

   This produces the binaries in the **build/dist/** directory.

3. Optional: To check the distribution one may run unit tests with command:

   ```
   ant test-junit
   ```

[1] Oracle JDK web page `http://www.oracle.com/technetwork/java/javase/downloads/index.html`
[2] OpenJDK web page `http://openjdk.java.net/`
[3] Apache Ant web page `http://ant.apache.org/`
[4] Your main need to specify full path to Apache Ant executable

Some tests[5] will not be performed because they require sensitive credentials which are not distributed in the public repository.

2.2 Running the framework

The *Framework* is packaged in the ***SPL.jar***[6] file located in the **build/dist** directory or in the directory where the prepared package has been extracted.

To run it open shell where the *SPL.jar* is located and execute

`java -jar SPL.jar`

There are also prepared shell wrappers for running

`spl.bat, spl.sh`

If no arguments are specified the usage help description is printed.

2.2.1 Arguments

XML configuration file
: This is the most essential argument. It is a path to the XML file specifying the project configuration. This file describes location of the code which will be scanned for annotations and how it will be obtained and how it will be build if necessary. It's a required argument. See [XML configuration file on page 23] for details.

Working directory
: This argument specifies the working directory where persistent and temporary data (including results itself) are stored. This and all following arguments are optional. If the argument is omitted the default value is **.spl** in the current working directory. See [Working directory on page 21] for details.

INI configuration file
: The optional configuration may be stored in this file. For example it may contain access information to remote machines or repositories or configuration of sampling code. See [INI configuration file on page 30] for details.

Execution machine ID
: The identification of the machine where to perform measuring. If this argument is omitted the current machine is used for measuring. Current machine may be also specified by **"local"** value of the argument. If remote machine is to be used the access details must be specified in the INI configuration file. See [INI configuration file on page 30] for details.

Choose interactivity
: This argument specifies whether the user want to interact with the application. This might be for example needed if remote execution is specified and the remote machine requires password or interactive authentication. Or when credentials are required for repository access. By default interactivity is disabled. To enable interactivity specify

[5] Those tests cover Git, SVN and SSH connection to remote machines. Tests are in cz.cuni.mff.spl.deploy-.build.vcs and cz.cuni.mff.spl.utils.ssh packages.

[6] *SPL.jar* file name may include version number

value **“console”** then prompt is processed in the console where the program is running or **“gui”** then prompt is handled by a pop up window.

If any optional argument is set all previous arguments must be set also. The order of arguments is mandatory.

2.2.2 Evaluation results

The results are created inside the *Framework* working directory in the **evaluation** directory. Each run of the program creates its own sub-directory in the **evaluation** directory. To view the results simply find and open **index.html** of the right evaluation. See [Working directory on page 21] for details about evaluation results location and [Specifying types of output to generate on page 32] for configuration of the evaluation output.

3. Examples

Now that the process of obtaining the *Framework* binaries and the mean of arguments has been briefly described let's see some examples.

3.1 Bundled tests

The main repository contains few examples of projects which are used within unit tests. Some of these projects may serve well as simple examples and show some basic functionality of the *Framework*.

To run following examples open shell and execute:

```
java -jar <path to SPL.jar> <path to spl.xml of corresponding project>
```

3.1.1 Test Basic

This is the simplest example bundled. It is located in *Framework* folder on path **/test-/projects/test-basic/** and contains three classes. **Method.java** implements two sorting functions. **Generator.java** implements a generator which can be used as an input to these functions. **Measurement.java** contains an annotation which defines a formula made of comparison of these sort functions using the generator.

Note following things in the **spl.xml**:

- The **repository** tag contains information where to obtain the source code of the project. In this case the type of the repository is **sourceRelative**. It means the location of the source is specified relative to the **spl.xml** itself in the **this** directory.
- The **build** tag contains information how to build the code when it's obtained. The specified command **ant** uses the **build.xml** file in project folder automatically as build command is run in project directory.
- The **classpaths** tag contains information where to locate the classes of built project.
- The **scanPatterns** tag contains information which packages located on the classpath should be scanned for annotations.

3.1.2 Test Multiple Projects

This example demonstrates how code from multiple projects (different repositories) can be combined.

The first project **THIS** contains the annotations.

It's important to note that every **spl.xml** file has to have **THIS** project specified and that project has to have revision with alias **HEAD**. This is the first code (revision) obtained and is scanned for the annotations which are processed afterwards and the other specified projects and revisions are accessed as needed.

The second project **METHOD** contains implementations of sorting functions.

The third project **GENERATOR** contains implementation of a generator which can be used as an input to these functions.

Note how the annotations have changed in the **Measurement.java** in **THIS** project. The declarations of generators and methods do not reference **THIS** project anymore but do reference corresponding projects.

3.1.3 Test Integrated Generators

This example demonstrates the usage of integrated generators. They might make measuring functions easier because there's no need to implement own generators. To use the integrated generators one must reference them from a special built-in project which contains the implementation. For further description of integrated generators see [Integrated generators on page 13].

3.2 Testing repositories

There are also testing repositories available on the Internet. They can be accessed using following commands:

```
git clone git://git.code.sf.net/p/spl-tools/testingrepository1
git clone git://git.code.sf.net/p/spl-tools/testingrepository2
svn checkout svn://svn.code.sf.net/p/spl-tools/testingrepository3/
```

The third one contain project with SPL annotations and configuration that demonstrates cross referencing and dependencies across multiple projects.

To test the *Framework* on those repositories, than just download the XML configuration file from the following URL:

```
http://sourceforge.net/p/spl-tools/testingrepository3/20/tree/spl-config.xml?
format=raw
```

And run the *Framework* as described in previous section with path to the downloaded XML file. The framework will obtain all three repositories on its own and will perform the measurement and evaluation for all SPL annotations inside the third testing repository.

4. Working directory

This chapter describes structure of a **directory** which serves for storage of measured data, processed evaluations and also temporary files during the execution of the *Framework*.

We will refer to this directory as a *the working directory*.

4.1 Evaluation

All created evaluations are stored in the **evaluation** directory inside *the working directory*. Every evaluation is placed in it's own directory. Evaluations are not deleted or overwritten between *Framework* runs.

This is the directory to look for evaluation results after the *Framework* has finished the job.

4.1.1 Overall structure

```
working_directory/evaluation/run-evaluate-0
working_directory/evaluation/run-evaluate-1
working_directory/evaluation/run-evaluate-2
...
```

4.1.2 Single evaluation directory

This section describes the content of a single evaluation result directory.

Results output files
: HTML, XML and graph PNG files are generated only when their generation is allowed by *Framework* configuration.

  index.html
  : The HTML index file of the evaluation with all found annotation locations and their result summaries.

  a-*.html
  : HTML files with details for annotations.

  f-*.html
  : HTML files with details for formulas.

  c-*.html
  : HTML files with details for comparisons.

  m-*.html
  : HTML files with details for measurements.

  spl-result.xml
  : Evaluation result in its bare form. Can be loaded by *SPL Tools Eclipse Plug-in* for interactive examination.

g-*.png
Graphs of comparisons and measurements.

spl.log
Output of *Framework* execution run on Debug level.

Framework support files
Dot prefixed files. These files are generated by the *Framework* and should not be modified or removed.

4.2 Measurement

The **measurement** directory in *the working directory* is used as a cache for measured data. When there is a project with a lot of measurements to perform it might be useful to cache these results. For example if new formulas are added that have references to new measurements and also to old measurements only the new measurements will be performed. The old measurements can be loaded from cache.

However this is possible only in case a version control system is used in the project (such as Git or Subversion). Otherwise it's not possible to determine whether the cache data are valid or not and the measurement is performed again.

Files in this cache directory might be deleted with a few consequences:

- If *Framework* is performing execution and has skipped measurement because its measured values were present in cache and it is deleted prior to evaluation then evaluation will not have data and not computed comparison results will be present.
- If you use *SPL Tools Eclipse Plug-in* to inspect evaluation results for evaluation with already removed measured data then you will no longer be able to generate user defined graphs.

To sum up, measurement cache should be never accessed while any instance is running. And the best practice is to purge the cache completely or not at all. Unless some measurements are precious.

4.3 Temporary

The **temporary** directory should not be modified while the program is running. It contains directories where remote repositories are checked out, sampling code is generated and where execution takes place.

4.4 Recovery

In case of *Framework* crash the temporary and the evaluation result directory might be (but does not have to be) removed. Measurement cache should, under normal conditions, stay consistent.

5. XML configuration file

XML configuration file contains configuration of projects, global generators, methods and parameters. It is part of every project that uses framework.

This file can be easily edited using SPL configuration editor in Eclipse plug-in described in section [SPL Configuration Editor on page 65].

5.1 File overview

The main element of the file is named **info** and contains these elements:

projects
: Contains **project** elements with information about project configuration and repository including all repository revisions. The **project** with alias **THIS** must be always present. It represents the project which this file belongs and is the only project where annotations are searched. It doesn't need to have this file inside project folder. It can be configured that for measurement one can have only this configuration file on local system and the rest of the project is stored in repository somewhere else.

generators
: Contains any number of **generator** elements which represents definition of generators used as input for measured methods. They are objects on which points global generator alias objects. One **generator** can be used by any number of aliases.

methods
: Contains any number of **method** elements which represents measured methods. They are objects on which points global method alias objects. One **method** can be used by any number of aliases.

global-generators
: Contains any number of **generator-declaration** objects. They represent globally defined generator aliases which uses **generator** element as its generator definition. These generator aliases can be used in any formula of annotated project.

global-methods
: Contains any number of **method-declaration** objects. They represent globally defined method aliases which uses **method** element as its method definition. These method aliases can be used in any formula of annotated project.

parameters
: Contains any number of **parameter** objects which are named real number constants that can be used as part of lambda expression or as parameter for generator in any formula of annotated project.

In the **info** element only **projects** element must be always present. Other elements are optional.

5.2 Main element details

Elements mentioned in configuration file overview are described here.

5.2.1 Element <project>

Example:

```
<project pid="id-generators">
    <alias>generators</alias>
    <build>ant</build>
    <classpaths>
        <classpath>lib/*.jar</classpath>
        <classpath>build/core</classpath>
    </classpaths>
    <scanPatterns>
        <scanPattern>cz.cuni.mff.spl.casestudy.annotations.**</scanPattern>
    </scanPatterns>
   <repository type="Git" url="git://git.code.sf.net/p/spl-tools/casestudy">
        <revisions>
            <revision rid="id-head">
                <alias>HEAD</alias>
                <value>master</value>
            </revision>
            <revision rid="id-current">
                <alias>current</alias>
                <value>bcfe6d43107430b2604e2e34b9151742941652e0</value>
                <comment>Fixed revision to allow results caching.</comment>
            </revision>

            <!-- further revisions -->

        </revisions>
    </repository>
</project>
```

The **project** attribute **pid** identifies the element in the document and has to be unique across all element identifiers of the document. **project** contains this elements:

alias
: Is the name of the project. Project is referred by it in formulas and it has to be unique for every project.

classpaths
: The element contains any number of **classpath** elements. They represents the locations where framework searches classes during measurement building. Wild card * can be used instead part of file name in the path. For example **directory/*.jar** means all files with **jar** extension in **directory** folder. Or **directory/**.jar** means all files with **jar** extension in **directory** folder and all its sub folders.

scanPatterns

The element contains any number of **scanPattern** elements. They represents the locations where framework searches annotations. It is used only for project **THIS**, for other projects it is ignored. Wild card * can be used in the pattern. If pattern is only * then it scans everything on the class path. Scanning can be restricted to a single class (pattern is full class name), single package (**package.***) or package with all sub packages (**package.****).

repository

Configures project repository. Attribute **type** defines which repository type is used. Currently are supported this types:

Git

Git versioning system.

Subversion

Subversion versioning system.

Source

Local source folder.

SourceRelative

Local source folder located relative to XML configuration file

Attribute **url** defines location of the repository.

Element **revisions** contains any number of **revision** but there have to be one named **HEAD** which represents the most current revision. That is the revision where annotations are searched if it belongs to the project **THIS**. If the repository type is source folder the revisions are ignored.

Each **revision** has attribute **rid** which identifies the element in the document and has to be unique across all element identifiers of the document. **revision** has these elements:

alias

The name of the revision. It is used in formulas to refer to the revision. It has to be unique for every revision of the project and cannot be empty.

value

Identifies the revision in its repository. The format of the value depends on the repository type. For example it can be label or hash for Git or a revision number for Subversion.

comment

Is an optional element where revision description can be placed. It has no influence on the measurement. Its only purpose is to be shown in SPL Configuration editor when revision is edited.

5.2.2 Element <generator>

Example:

```
<generator gid="id-gen1">
    <path>cz.cuni.mff.spl.generators.IntegerGenerator</path>
    <parameter>p1</parameter>
    <genMethod>
        <name>factory</name>
        <parameter>p2</parameter>
    </genMethod>
    <revision rref="id-current"/>
</generator>
```

The **generator** attribute **gid** identifies the element in the document and has to be unique across all element identifiers of the document. **generator** contains this elements:

path
: Is the full class name of the generator class.

parameter
: Is an optional string parameter for constructor of generator class.

genMethod
: Is present only for method generator. For class generator this element is omitted. It is the method used to obtain data for measured method. **name** is the name of the method and cannot be empty. **parameter** is optional string parameter of the method.

revision
: Refers to the revision from which come the generator. The attribute **rref** refers to the attribute **project/repository/revisions/revision/@rid**.

5.2.3 Element <method>

Example:

```
<method mid="id-method1">
    <path>cz.cuni.mff.spl.basictests.IntegerSumCalculator</path>
    <parameter>foo</parameter>
    <name>calculateSumOfIntegers</name>
    <parameterTypes>char</parameterTypes>
    <parameterTypes>int</parameterTypes>
    <revision rref="id-head"/>
    <declarated type="WITH_PARAMETERS"/>
</method>
```

The **method** attribute **mid** identifies the element in the document and has to be unique across all element identifiers of the document. **method** contains this elements:

path
: Is the full class name of the method class.

parameter
: Is an optional string parameter for the class constructor.

name
Is the name of the method.

parameterTypes
Is the type of the method parameter. There can be any number of them and they are sorted according to their order in method header.

revision
Element refers to revision from which come the method. The attribute **rref** refers to the attribute **project/repository/revisions/revision/@rid**.

declared
Determines if method parameter types was declared. It has **type** attribute which can be:

WITH_PARAMETERS
Marks methods declared with parameter types. Method is fully determined by this declaration. If there is no element **parameterType** then it is assumed that method has no parameters.

WITHOUT_PARAMETERS
Marks methods declared without parameters. In the case of method overloading the correct method is chosen by the type of values returned by generator. If element **parameterType** is present then it is ignored.

5.2.4 Element <generator-declaration>

Example:

```
<generator-declaration pdid="id-globalGen" gref="id-gen1">
    <image>
        intGen=generators@current:
        cz.cuni.mff.spl.generators.IntegerGenerator('p1')#factory('p2')
    </image>
    <alias>globalGen</alias>
</generator-declaration>
```

The **generator-declaration** attribute **pdid** identifies the element in the document and has to be unique across all element identifiers of the document. The attribute **gref** refers to the attribute **generators/generator/@gid** of the generator used as generator declaration for this global alias. **generator-declaration** contains this elements:

image
Is the string that was parsed to create this alias.

alias
Is the non empty name of this generator alias which has to be unique across all generator aliases. It can be used for referring to this declaration in any formula of the annotated project.

5.2.5 Element <method-declaration>

Example:

```
<method-declaration pdid="id-foo" mref="id-method1">
    <image>
        integerSum=generators:
        cz.cuni.mff.spl.basictests.IntegerSumCalculator('foo')
        #calculateSumOfIntegers(char, int)
    </image>
    <alias>foo</alias>
</method-declaration>
```

The **method-declaration** attribute **pdid** identifies the element in the document and has to be unique across all element identifiers of the document. The attribute **gref** refers to the attribute **methods/method/@mid** of the method used as method declaration for this global alias. **method-declaration** contains this elements:

image
: Is the string that was parsed to create this alias.

alias
: Is the non empty name of this method alias which has to be unique across all method aliases. It can be used for referring to this declaration in any formula of the annotated project.

5.2.6 Element <parameter>

Example:

```
<parameter>
    <name>pi</name>
    <value>3.14</value>
</parameter>
```

parameter contains this elements:

name
: Is a non empty name of this parameter which has to be unique across all parameters. It can be used for referring to this parameter in any formula of the annotated project.

value
: Is a non empty real number parameter value. Parameter is substituted by the value during formula parsing.

5.3 Example XML file

The example below shows XML file content with one project named **THIS** with Git repository with two revisions and one example of global generator alias, global method alias and parameter.

```
<?xml version="1.0" encoding="UTF-8"?>
<info>
    <projects>
        <project pid="id-this">
            <alias>THIS</alias>
            <build>ant</build>
            <classpaths>
                <classpath>build/classes</classpath>
            </classpaths>
            <scanPatterns>
                <scanPattern>cz.cuni.mff.spl.**</scanPattern>
            </scanPatterns>
            <repository type="Git" url="git://repository.url">
                <revisions>
                    <revision rid="id-thisHead">
                        <alias>HEAD</alias>
                        <value>master</value>
                    </revision>
                </revisions>
            </repository>
        </project>
    </projects>

    <generators>
        <generator gid="id-generatorExample">
            <path>cz.cuni.mff.spl.Example</path>
            <revision rref="id-thisHead" />
        </generator>
    </generators>

    <methods>
        <method mid="id-methodFoo">
            <path>cz.cuni.mff.spl.Example</path>
            <name>foo</name>
            <revision rref="id-thisHead" />
            <declarated type="WITHOUT_PARAMETERS" />
        </method>
    </methods>

    <global-generators>
        <generator-declaration pdid="id-globalGen1" gref="id-generatorExample">
            <image>gen1=cz.cuni.mff.spl.Example</image>
            <alias>gen1</alias>
        </generator-declaration>
    </global-generators>

    <global-methods>
        <method-declaration pdid="id-globalMethod1" mref="id-methodFoo">
            <image>method1=cz.cuni.mff.spl.Example#foo</image>
            <alias>method1</alias>
        </method-declaration>
    </global-methods>

    <parameters>
        <parameter>
            <name>THREADS</name>
            <value>4.0</value>
        </parameter>
    </parameters>
</info>
```

6. INI configuration file

Project can be more finely configured using INI configuration file. The difference between INI configuration and XML configuration is that the XML configuration is supposed to be a public configuration common to all users and contrary to that INI configuration is a private configuration different for every user.

It contains access details to private repositories and machine. There's evaluator configuration with graph configuration and statistical settings. It's also possible to configure sampling process with Java arguments and time or call limits.

There are different sections in the INI file. Each section has its own set of rules where key-value pairs are defined. Section is specified using `[section-name]`. All following key-value pairs belong to that section.

This file can be easily edited using INI Configuration Editor in Eclipse plug-in described in section [INI Configuration Editor on page 67].

6.1 Access

This section contains connection details to repositories and remote execution machines.

6.1.1 Private repositories

In the XML project configuration the repository is configured specifying its type (Git, Subversion, source folder or other) and URL.

This is suitable for public or local repositories but in case the project's source is placed in a private repository that requires authentication more details can be specified in the INI configuration.

To create a section for Git or Subversion authentication create following declaration **`[access-.git.repo1]`** or **`[access.subversion.repo1]`** The last part of the dot separated name must match the name of the project this configuration belongs to in XML configuration.

For Git and Subversion supported values are the same. The distinction is purely for future compatibility if another version control system with different needs is added.

`username = user1`
: Username to login with.

`keyPath = /home/user1/.ssh/key.rsa`
: Path to private key for SSH authentication.

`trustAll = false`
: Whether to trust all host keys.

`fingerprint = 01:02:03:04...`
: Fingerprint of remote host public key to trust to.

`knownHostsPath = /home/user1/.ssh/knownhosts`
: Path to file with known hosts. Is used read only.

When specifying path to files **"~"** is not applicable as a home directory alias. It's is handled as a standard character.

If password or passphrase is required interactivity arguments must be set as in the remote execution example.

6.1.2 Secure shell details

To declare an execution machine details with name **id1** for later use create following section `[access.machine.id1]`. The last part of dot separated name is used for referencing the machine later. Possible keys for this section are (with example values)

`url = u-pl0.ms.mff.cuni.cz`
: Host name or IP address of the remote machine.

`path = spl/execution`
: The directory where the execution will be performed.

`username = user1`
: Username to login with.

`keyPath = /home/user1/.ssh/key.rsa`
: Path to private key for ssh authentication.

`trustAll = false`
: Whether to trust all host keys.

`fingerprint = 01:02:03:04...`
: Fingerprint of remote host public key to trust to.

`knownHostsPath = /home/user1/.ssh/knownhosts`
: Path to file with known hosts. Is used read only.

To use remote execution following arguments must be passed to *Framework*.

```
java -jar SPL.jar spl.xml spl-wd spl.ini id1
```

Where **spl.xml** is path to the project configuration, **spl-wd** is the working directory, **spl.ini** is the file with previously described configuration and **id1** is the name of the machine to use for execution. If further details such as password or passphrase is required for authentication interactivity must be enabled using following argument

```
java -jar SPL.jar spl.xml spl-wd spl.ini id1 console
```

Preferred way is to use private keys without passphrase for authentication.

6.2 Evaluation

The evaluation configuration includes configuration options for statistical evaluation, generating HTML output files, XML evaluation results description file and generated graph types for comparisons and measurements.

All declaration key-value examples in this section are listed with their default values.

6.2.1 Specifying types of output to generate

The INI section `[evaluator.output]` contains declarations that specify which output types should be generated. All values are of type **boolean** where *true* value can be written as one of *1*, *true* or *yes* and *false* value as one of *0*, *false* or *no*.

`generate-html-output = 1`
: Enables generation of the HTML result report

`generate-xml-output = 1`
: Enables generation of the XML result file.

`generate-graph-output = 1`
: Enables generation of the graph files for comparisons and measurements. Those files are added to the HTML report automatically if they are created.

6.2.2 Statistical evaluation parameters

The INI section `[evaluator.statistics]` contains following values for the statistical evaluation:

`minimum-sample-count-warning-limit = 10`
: Defines minimum number of measurement samples. Otherwise, warning would be displayed.

`t-test-limit-p-value = 0.05`
: Defines limit value for evaluating result of statistical t-test for comparison. Comparison is satisfied when t-test result p-value is greater than the limit value set here. The value should be floating point value from interval $[0, 1]$.

`default-equality-interval = 0.05`
: Defines default parameter for the **relaxed equality** ($=$) comparison operator.

`maximum-standard-deviation-vs-mean-difference-warning-limit = 75.0`
: Defines limit value for ratio of measurement standard deviation and mean in percent when warning should be shown in result report.

  The warning will be issued, when $\frac{standard\_deviation}{mean} \times 100 > limit$.

`maximum-median-vs-mean-difference-warning-limit = 20.0`
: Defines limit value for ratio of measurement median and mean in percent when warning should be shown in result report.

  The warning will be issued, when $\frac{median}{mean} \times 100 \notin [100 - limit, 100 + limit]$.

The warnings issued feature when *limit* is exceeded is not based on the statistical theory, but it is based on the observations of the measurement behaviour when big difference in the mean and standard deviation or median usually indicated problems during the measurement.

6.2.3 Graph generation configuration

Basic graph configuration

The INI section `[evaluator.graphs]` contains the basic configuration options used to generate graphs. Graphs are generated as *Portable Network Graphics* (PNG) images.

`graph-image-width = 800`
: The width for generated graph image in pixels.

`graph-image-height = 600`
: The height for generated graph image in pixels.

`rscript-command = Rscript`
: The command to run the **Rscript** executable for calculating probability density function of measurement data. Default value `Rscript` should work on both Windows and Linux systems with **R Project** installed.

`histogram-minimum-bin-count = 100`
: Defines minimum number of bins for plotting values in histogram graph types.

`histogram-maximum-bin-count = 10000`
: Defines maximum number of bins for plotting values in histogram graph types.

`graph-maximum-normal-density-y-axis-limit = 1.0E-4`
: This option allows to set maximum Y axis value for probability density graph for measurement (when normal distribution is plotted together with normal density with the same mean and standard deviation). The default value is 10^{-4} (0.0001). Set the value to `1` to disable this feature.

Specifying generated graph types

The types of graphs which should be generated for measurements are defined in the INI section `[evaluator.graphs.measurement]` and graph types for comparisons in the section `[evaluator.graphs.comparison]`.

The entries in both sections have the same format:

graph<order number> = <graph definition>

The **order number**s define the order of graphs during generation and also their order in the generated HTML report pages. There can be only one graph key with one **ordering number**, otherwise there is no guarantee which declaration will be effectively used. The ordering numbers have to be integers and they may not follow-up (i. e. the sequence 1, 2, 3, 4 has the same effect as the sequence 2, 4, 6, 7). The actual order of declarations in INI file has no influence on the **ordering number**. Note that there is no space between *graph* and **order number**.

The **graph definition** has following syntax:

<graph type>(*<sample data clip type>* [, *<numeric argument>*]*)

The graph type is one of following values:

Histogram
: Histogram of measured samples.

DensityComparison
: Probability density comparison graph.

TimeDiagram
: Graph describing measured times during the execution.

The measured sample data can be clipped to provide various views on them. The sample data clip type is one of following values:

None
: No data clipping, plot all measured values.

Sigma
: Measured samples that are farther from the mean than the sigma (standard deviation) multiplied with *specified multiplier* are removed. Number of this removal operation iterations can be specified as second parameter – specifying higher value can lead to more stable graphs as the mean and the standard deviation are computed from current measurement dataset.

  The first parameter defines the multiplier for standard deviation of measured data. The default value is `3` and it is used when the no parameter is specified.

  The second parameter defines number of iterations to run the clipping process. The default value is `1` and it is used when the second parameter is not specified.

Quantile
: Measurement samples that are lower than the lower percentile specified as the first numeric argument or higher than upper percentile specified as the second argument of measurement sample dataset are removed. Lower and upper percentile arguments have to be specified as floating point numbers in the interval $[0, 1]$.

Default graph definitions for each measurement:

```
graph1 = Histogram(None)
graph2 = Histogram(Sigma, 3.0, 1.0)
graph3 = DensityComparison(Sigma, 3.0, 1.0)
graph4 = Histogram(Quantile, 0.1, 99.1)
graph5 = Histogram(Quantile, 0.0, 99.0)
graph6 = Histogram(Quantile, 0.0, 95.0)
graph7 = TimeDiagram(Sigma, 3.0, 1.0)
graph8 = TimeDiagram(Quantile, 1.0, 99.0)
```

Default graph definitions for each comparison:

```
graph1 = DensityComparison(Sigma, 3.0, 1.0)
graph2 = Histogram(Quantile, 0.1, 99.9)
```

Setting graphs colors

The INI section `[evaluator.graphs.colors]` contains specification of colors used for generating graphs.

`text = black`
: The color for labels.

`background = lightgray`
: The color for background.

`background-transparent = 0`
: The flag indicating if the background should be transparent or not. This flag has no effect now and is reserved for further usage.

`sample<order number> = <color declaration>`
: Colours for the plotted sample series.

The **order number** has same meaning as the **order number** in the section [**evaluator.graphs.measurement**] described before.

The **color declaration** is either one of *red*, *orange*, *yellow*, *green*, *cyan*, *blue*, *violet*, *magenta*, *white*, *gray*, *black* or RGB declaration:

`rgb(<red component>, <blue component>, <blue component>)`

The **color component** is integer number from interval $[0, 255]$.

For example, the declaration for black color is `rgb(0,0,0)`.

Default colors for sample series

```
sample1 = red
sample2 = blue
sample3 = green
sample4 = yellow
sample5 = orange
sample6 = cyan
sample7 = magenta
```

6.3 Deployment

There are more values that might be set that do affect either the measuring process or clean up process.

These properties are all set with default values listed below. To override them specify following keys in a section named `[deployment]`

`useSystemShell = true`
: Sets whether to execute project's build command in a system shell or not. By default, on Windows build command is executed in `cmd /c` environment and on Linux and all other platforms in `/bin/sh -c` environment. If different shell is to be used set this option to `false` and specify complete command in the project's configuration.

`clearTmpBefore = true`
: If set true temporary directory is completely cleared when *Framework* is started.

clearTmpAfter = false
: If set true temporary directory is completely cleared when *Framework* has finished the job.

javaPath = "java"
: Using this property a preferred Java binary may be set for usage on execution machine.

samplerArguments = ""
: Arguments passed to Java at start of each sampling binary. For example `"-server"`.

warmupCycles = 1000
: How many calls to the measured function should be maximally performed to warm up.

warmupTime = 5
: How much time should be maximally spend calling measured function to warm up in seconds.

measurementCycles = 2000
: How many calls to the measured function should be maximally sampled.

measurementTime = 10
: How much time should be maximally spend measuring in seconds.

timeout = 300
: How much time can the sampling binary run at most in seconds. This option is to make sure sampling code does not hang somehow. If exceeded sampler gets killed. Set 2-3 times more then warming and measuring time.

6.4 Example INI file

The example below shows INI file contents with the default configuration without any access details for remote machines or private repositories.

```
[deployment]
useSystemShell = 1
clearTmpBefore = 1
clearTmpAfter = 0
samplerArguments =
warmupCycles = 1000
warmupTime = 5
measurementCycles = 2000
measurementTime = 10
timeout = 60

[evaluator.output]
generate-html-output = 1
generate-xml-output = 1
generate-graph-output = 1

[evaluator.statistics]
minimum-sample-count-warning-limit = 10
t-test-limit-p-value = 0.05
default-equality-interval = 0.05
maximum-standard-deviation-vs-mean-difference-warning-limit = 75.0
maximum-median-vs-mean-difference-warning-limit = 20.0
```

```
[evaluator.graphs]
graph-image-width = 800
graph-image-height = 600
rscript-command = Rscript
histogram-minimum-bin-count = 100
histogram-maximum-bin-count = 10000
graph-maximum-normal-density-y-axis-limit = 1.0E-4

[evaluator.graphs.measurement]
graph1 = Histogram(None)
graph2 = Histogram(Sigma, 3.0, 1.0)
graph3 = DensityComparison(Sigma, 3.0, 1.0)
graph4 = Histogram(Quantile, 0.1, 99.1)
graph5 = Histogram(Quantile, 0.0, 99.0)
graph6 = Histogram(Quantile, 0.0, 95.0)
graph7 = TimeDiagram(Sigma, 3.0, 1.0)
graph8 = TimeDiagram(Quantile, 1.0, 99.0)

[evaluator.graphs.comparison]
graph1 = DensityComparison(Sigma, 3.0, 1.0)
graph2 = Histogram(Quantile, 0.1, 99.9)

[evaluator.graphs.colors]
text = black
background = lightgray
background-transparent = 0
sample1 = red
sample2 = blue
sample3 = green
sample4 = yellow
sample5 = orange
sample6 = cyan
sample7 = magenta
```

7. Case Study

7.1 Introduction

This chapter is sum of experiences of usage *SPL Tools Framework* gained during case study creation. In the study will be demonstrated usage of the framework on a real world project. The intention of the study is not only to show usability of the framework but also the experience of using it from the side of a common user. That will help to develop more user-friendly application as well as to remove unexpected behaviour of the result software.

XML parsing was chosen to present usefulness of SPL. XML data are widely used so the parsers have to be fast. Developers should improve its performance and parsers are not based on some exact algorithm so every line of code can significantly influence the performance. Otherwise in a project that implements some well known algorithm there may not be much performance improvements.

It is required that the real world application on which the framework is tested is an open source project with source code stored in some supported revision control system.

For this reasons was chosen JDOM library[1] which purpose is to provide a Java-based solution for accessing, manipulating, and outputting XML data from Java code.

Its code is stored in Git repository. Initial commit was on 28–5–2000, latest commit (to the 18–2–2013) was on 9–11–2012 so it is more than 12 years old and it is still active.

Project has well commented source code and many revisions with apposite description. Another benefit is that the developers made their own performance testing[2] on which the study can be based.

Group of possibly interesting revisions was chosen through the whole life of the project. Revisions that may affect the performance was chosen and the decision was based on revision description and brief code exploration.

7.2 Study structure

The study is stored in Git repository[3]. Source files with annotations and generators are there along with configuration files needed to run measurement. File paths in the next text are relative to the directory where the study is checked out.

7.2.1 Configuration

Configuration files are stored in **src/spl/** directory. There is **spl.xml** file where global generators, projects and its revisions are configured and **spl.ini** file where measurement details like warm up and measurement time are specified.

[1] JDOM web page `http://jdom.org/`
[2] JDOM performance testing `https://github.com/hunterhacker/jdom/wiki/Verifier-Performance`
[3] Study Git repository `git://git.code.sf.net/p/spl-tools/casestudy`

In this chapter revision aliases which are defined in **spl.xml** are used for revision identification instead of their hash code.

Configured projects are:

THIS
: Contains configuration of the case study project itself.

jdom
: This is the annotated JDOM project used for measuring revisions after **start** revision which was committed on 2–8–2011. JDOM project changed its classpath and build command in it so it is needed to configure the same repository as two separate projects.

jdom_init
: JDOM project used from its initial commit to the commit from 23–7–2009 before its configuration changed.

generators
: This is the same project as **THIS** and configured with the same online Git repository but with other purpose. While **THIS** project source can be for debugging reason changed to be local, **generators** project should stay as Git repository because the measurements that use generators from the same project and the same revision will not be measured again until the measured method revision changes. If local source generators are used then the method will be always measured even when the method revision is the same.

7.2.2 Source files

Java source codes in **src/java/** directory contains four packages:

cz.cuni.mff.spl
: It is the package where SPL annotation is defined.

cz.cuni.mff.spl.casestudy.annotations
: Classes with annotations are there. Main class is intended for adding new annotations and running the method for testing generators from **TestMethods** class before measurement. Annotations that are already measured and runs without errors are separated to **MeasuredAnnotations*** classes where * is number from 1 to 4.

cz.cuni.mff.spl.casestudy.generators
: Contains generator classes used in measurements. All generators give the same predefined values for each run, no random data are used. They read their values from resource files and store it in memory so during measurement the program will not access the hard disk. This is important because if generator reads some data from disk during measurement then it greatly affects the result time. The time needed to read data from disk is inconsiderable and may be much greater than time of running the measured method itself.

cz.cuni.mff.spl.casestudy.generators.resources
: There are placed resource files from which generators read their values. Resource files were taken from JDOM developers performance testing.

7.2.3 Annotation structure

Annotations are written next to blank methods. Every annotation uses just methods from JDOM projects for measurement so the annotations are separated from measured methods. It allows writing annotations without changes in the JDOM source code.

In annotations global generator and local method aliases are used.

Next to method where annotation is placed is written Javadoc comment describing what code part annotation measures, what changes was made and what is the result. Because this comments don't propagate into measurement results every method have name combined from three parts for better orientation in result presentation:

`resultDescriptionMeasured`

The part meanings are:

result
: describes the result of all annotation formulas, can be one of:

  satisfy
  : formula in most cases holds

  fail
  : formula in most cases fails

  unstable
  : formula in many cases holds but even not very big measurement deviation can cause that formula fails

Description
: short description what code change is measured

Measured
: what part is measured, mostly it is name of a class

7.2.4 Measured methods

Before annotations were written a couple of methods were chosen to be measured through the time and their performance is compared to themselves in a number of revisions. The measured classes are:

Verifier
: *A utility class to handle well-formedness checks on names, data, and other verification tasks for JDOM (JavaDoc documentation).*

  It is used many times during document parsing so there the authors made the greatest effort to make it as fast as possible. Due to the developers performance testing there are three static methods that are the entry point for almost all of the **Verifier**'s activity:

  checkAttributeName(String name)
  : checks if supplied name is legal attribute name

  checkElementName(String name)
  : checks if supplied name is legal element name

checkCharacterData(String text)
check if supplied text contains only characters allowed by the XML 1.0 specification

ParseFileGenerator generator is used as an input for these methods with different source file for each method.

SAXBuilder
Builds a JDOM Document using a SAX parser (JavaDoc documentation).

Have only one measured method - **build(java.io.InputStream)** which is called on **SAXBuilder** instance and builds a JDOM Document using a SAX parser. During the building is used **Verifier** many times so its measurement usually measures incidence of **Verifier** performance change to whole parsing process. **XMLReadFileGenerator** with few XML source files is used as an input for this method. The difference between this and the **Verifier**'s methods is that the **SAXBuilder** class instance is needed for this measurement. Framework itself can recognize that and generates correct code for measurement.

DOMBuilder
Builds a JDOM Document from a pre-existing DOM org.w3c.dom.Document (JavaDoc documentation).

Have only one measured method - **build(org.w3c.dom.Document domDocument)** which is called on **DOMBuilder** instance and builds a JDOM Document from a pre-existing DOM. Accordingly **SAXBuilder** it also uses **Verifier** and can be used for measuring incidence of **Verifier** performance change to building process. **DOM-generator** with few XML source files is used as an input for this method.

7.2.5 Generators

In measurements three generators are used. Every generator use one or more input files. The set of files paths generator can use is stored in array in generator class. Which file will be read is specified in the generator constructor. Generators are:

ParseFileGenerator
reads one file of strings separated by space and stores it in array each string as one array item. It is used for measuring all measured methods in **Verifier** but for each method it use different input file. Its alias definition is:

```
gattributes =
generators@current:
cz.cuni.mff.spl.casestudy.generators.ParseFileGenerator('0')
```

Where **gattributes** is alias for the generator followed by generator declaration that means from project **generators** at revision **current** use class **cz.cuni.mff.spl.case-study.generators.ParseFileGenerator** which have constructor with one String parameter and its value is **0**. This generator is used for reading file with data for **checkAttributeName(String name)** method. To read file for **checkCharacter-Data(String text)** method the only change in generator declaration is that used parameter value is **1** and for **checkElementName(String name)** method it is **2**.

XMLReadFileGenerator
reads one or more XML files to the string, converts it to stream and every stream stores as single item in array. Which file will be read is specified in constructor by String parameter of the file number (0 to 2) or if constructor without parameters is used then it reads files from all stored paths. It is used as **SAXBuilder** input. Its alias definition is:

```
gxmlread =
 generators@current:
  cz.cuni.mff.spl.casestudy.generators.XMLReadFileGenerator()
```

Where **gxmlread** is alias for generator that from project **generators** at revision **current** uses class **cz.cuni.mff.spl.casestudy.generators.XMLReadFileGenerator** with constructor with no parameters.

DOMgenerator
Generator that reads one or more XML files, constructs **org.w3c.dom.Document** from it and every **Document** stores as single item in array. Which file will be read is specified in constructor by String parameter of the file number (0 to 7) or if constructor without parameters is used then it reads files from all stored paths. It is used as **DOMBuilder** input. Its alias definition is almost the same as **XMLReadFileGenerator** the only difference is in used class:

```
gdomgenerator =
 generators@current:
  cz.cuni.mff.spl.casestudy.generators.DOMgenerator()
```

Where **gdomgenerator** is alias for generator that from project **generators** at revision **current** uses class **cz.cuni.mff.spl.casestudy.generators.DOMgenerator** with constructor with no parameters.

7.2.6 Running the measurement

Because whole case study project is accessible through the Internet the simplest way of running measurement is to obtain *Framework* repository and in the directory where the code has been checked out run

```
ant case-study
```

It will download case study configuration files to the **build/examples/case-study/** directory and run the framework with it which will download all necessary repositories itself. As a working directory will be used **build/examples/case-study/spl-wd/** directory.

If framework is present just as JAR library then the measurement can be run by executing

```
java -jar <path to SPL.jar> <spl.xml> <path to spl-wd> <spl.ini>
```

where **spl.xml** and **spl.ini** are configuration files which are described in section [Configuration on page 38] and **spl-wd** is working directory where the measurement will be run.

On some computers running of the study needs command line Git to be present because library used by framework fails as is described in section [Git conflicts on page 85].

The study is configured that each measurement takes five minutes so the overall running time is about nine hours.

Running the study on Windows operating system isn't suitable because deviations are quite big thus the results are inferior. The best results were given when used dedicated Unix machine.

7.3 Example results

There is number of measurements to inspect in the case study. Annotations for them were written for interesting revisions which were chosen based on JDOM repository log. If the revision description look like the change can affect performance then brief code examination was made. This revision was chosen for measurement if even the change in the code look like it can influence the performance.

Most of the changes hardly have any outward impact to the performance according to the measurement results. However there are some updates that really have some performance gain or on the other hand they should have but the reality was unwilling. Or there are measurements that just look curious. This kinds of measurements is described in this section.

Generator aliases which definition was described in section [Generators on page 41] are used as input for all following measurements.

7.3.1 Detailed look at the measurements

First lets have detailed look on the measurements examples, how annotations for them are written and what is their meaning.

On the beginning in 2000 year there was a big change in the **Verifier** where order of character checking was changed.

Verifier.checkElementName

This update sped up mainly **checkElementName** one of three measured **Verifier**'s methods which runs about 10 % faster which look like good improvement.

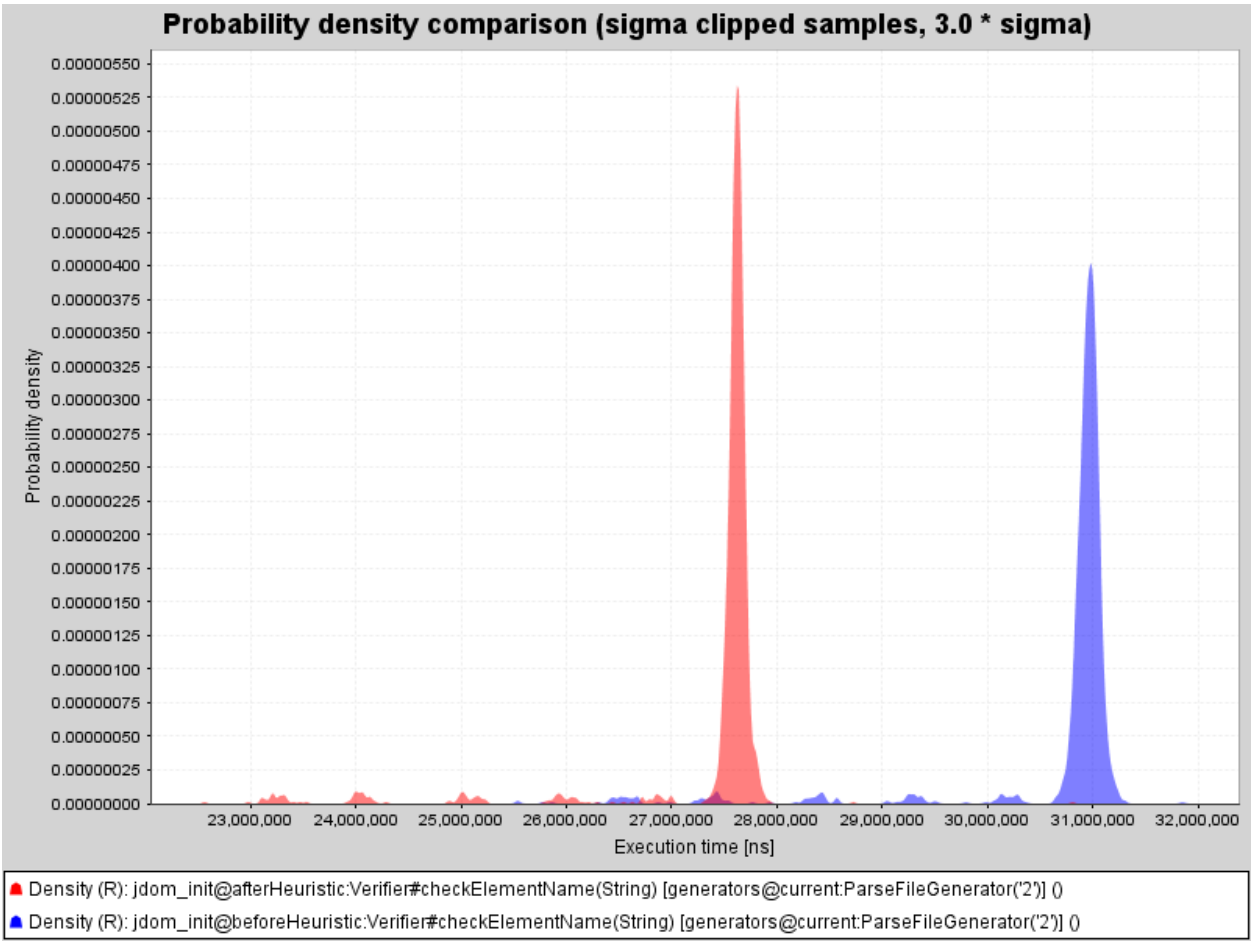


Figure 7.1: Verifier.checkElementName improvement, red graph is after update.

For definition of this comparison are used method aliases:

```
beforeHeuristicCheckElements =
jdom_init@beforeHeuristic:
org.jdom.Verifier#checkElementName(String name)

afterHeuristicCheckElements =
jdom_init@afterHeuristic:
org.jdom.Verifier#checkElementName(String name)
```

Where first one is alias named **beforeHeuristicCheckElements** and after equal sign is method definition. First part of definition is specification of project with alias **jdom_init** and its revision with alias **beforeHeuristic** both are defined in **spl.xml** configuration file. In this revision is class **org.jdom.Verifier** with method **checkElementName** with one String parameter. That is the method for which the alias is defined.

The second alias **afterHeuristicCheckElements** is defined for the same method but in revision **afterHeuristic**.

When the method aliases are defined the comparison of them is simple:

```
afterHeuristicCheckElements[gelements] <=
beforeHeuristicCheckElements[gelements]
```

On each part of comparison is method with its generator. There are used method and generator aliases but whole method definition can be used instead of alias. Aliases just makes the comparison more readable. There it means that method for which is defined alias **afterHeuristicCheckElements** with generator **gelements** as its input is compared with method for which is defined alias **beforeHeuristicCheckElements** with the same generator as its input. Both sides of comparison are separated by comparison operator which defines relation between measured time of left and right side of the comparison. There it means that the first one measured time is lesser or equals than the second one measured time. In other words first one should run faster than or equal as the second one.

Verifier.checkCharacterData

On the other hand the **checkCharacterData** method runs after the change about 10 % slower.

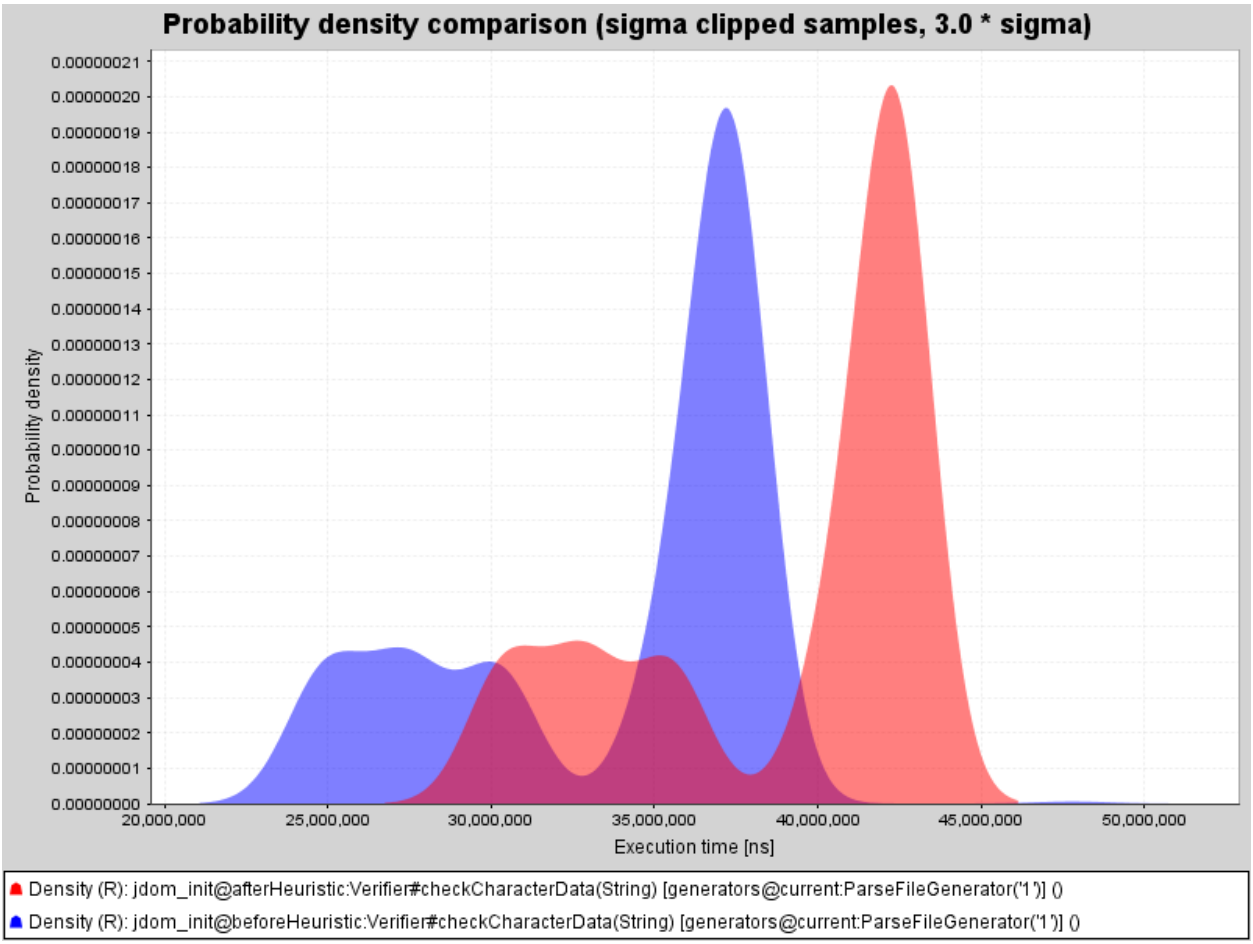


Figure 7.2: Verifier.checkCharacterData downgrade, red graph is after update.

For this comparison are used aliases which are similar to the previous comparison but there are different names of aliases and the measured method:

```
beforeHeuristicCharacter =
jdom_init@beforeHeuristic:
org.jdom.Verifier#checkCharacterData(String name)

afterHeuristicCheckCharacter =
jdom_init@afterHeuristic:
org.jdom.Verifier#checkCharacterData(String name)
```

First one is alias named **beforeHeuristicCharacter** and after equal sign is method definition. First part of definition is specification of project with alias **jdom_init** and its revision with alias **beforeHeuristic**. In this revision is class **org.jdom.Verifier** with method **checkCharacterData** with one String parameter. That is the method for which the alias is defined.

The second alias **afterHeuristicCheckCharacter** is defined for the same method but in revision **afterHeuristic**.

The comparison of defined aliases is:

```
afterHeuristicCheckCharacter[gcharacters] <=
beforeHeuristicCharacter[gcharacters]
```

It means that method for which is defined alias **afterHeuristicCheckCharacter** with generator for which is defined alias **gcharacters** as its input is compared with method for which is defined alias **beforeHeuristicCharacter** with the same generator as its input. This comparison operator means that left side of comparison should run faster than or equal as the right side.

Verifier.checkAttributeName

The third **Verifier**'s method **checkAttributeName** wasn't almost influenced:

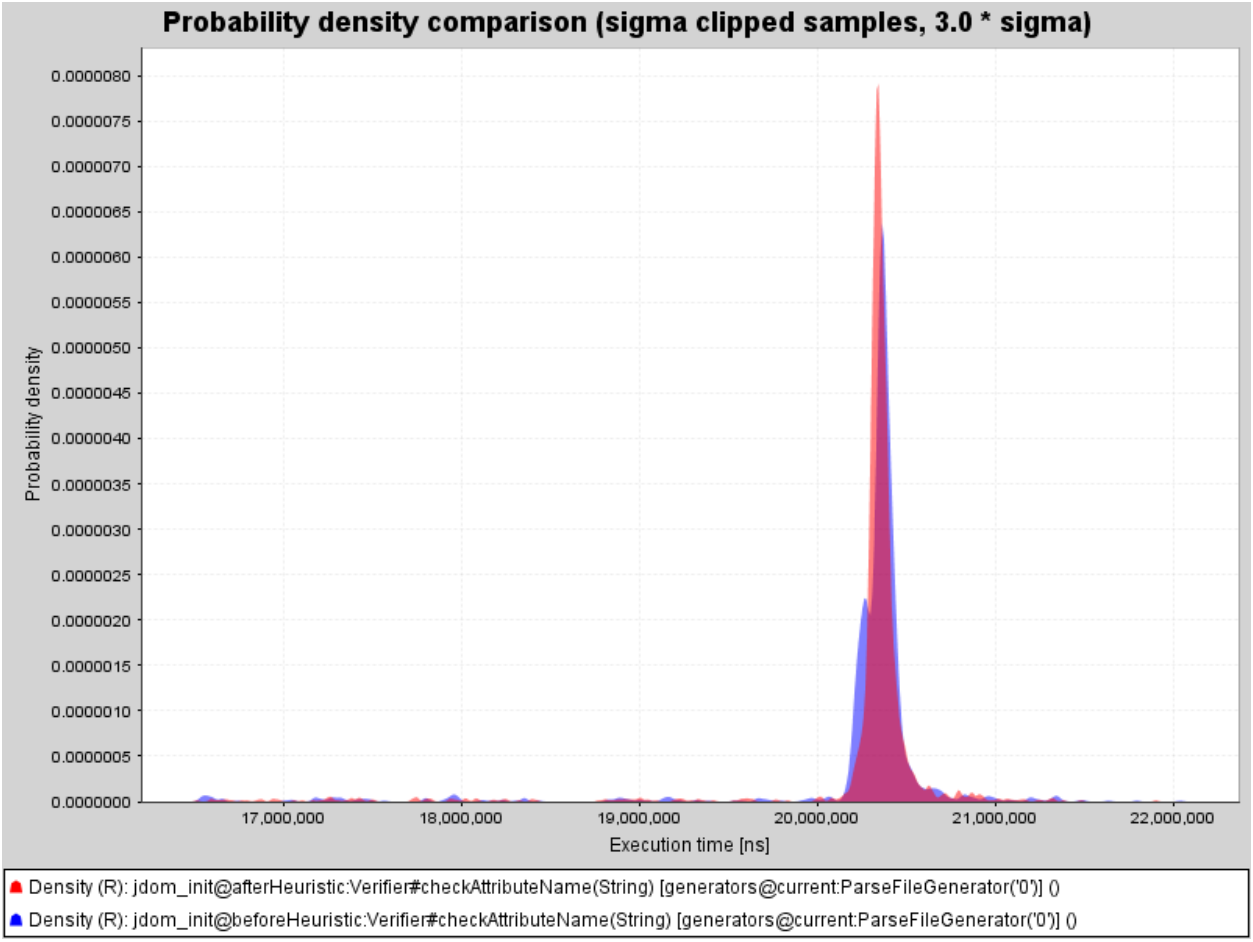


Figure 7.3: Verifier.checkAttributeName runs almost the same, red graph is after update.

Aliases used in the comparison also are similar to the previous two comparisons and only differences are names of aliases and measured method:

```
beforeHeuristicCheckAttributes =
jdom_init@beforeHeuristic:
org.jdom.Verifier#checkAttributeName(String name)

afterHeuristicCheckAttributes =
jdom_init@afterHeuristic:
org.jdom.Verifier#checkAttributeName(String name)
```

First one is alias named **beforeHeuristicCheckAttributes** and after equal sign is method definition. First part of definition is specification of project with alias **jdom_init** and its revision with alias **beforeHeuristic**. In this revision is class **org.jdom.Verifier** with method **checkAttributeName** with one String parameter. That is the method for which the alias is defined.

The second alias **afterHeuristicCheckAttributes** is defined for the same method but in revision **afterHeuristic**.

The comparison of defined aliases is:

```
afterHeuristicCheckAttributes[gattributes] <=
beforeHeuristicCheckAttributes[gattributes]
```

It means that method for which is defined alias **afterHeuristicCheckAttributes** with generator for which is defined alias **gattributes** as its input is compared with method for which is defined alias **beforeHeuristicCheckAttributes** with the same generator as its input. This comparison operator means that left side of comparison should run faster than or equal as the right side.

SAXBuilder

This improvement doesn't have any significant influence on the whole process creating XML structure with **SAXBuilder** which runs almost the same as before the change:

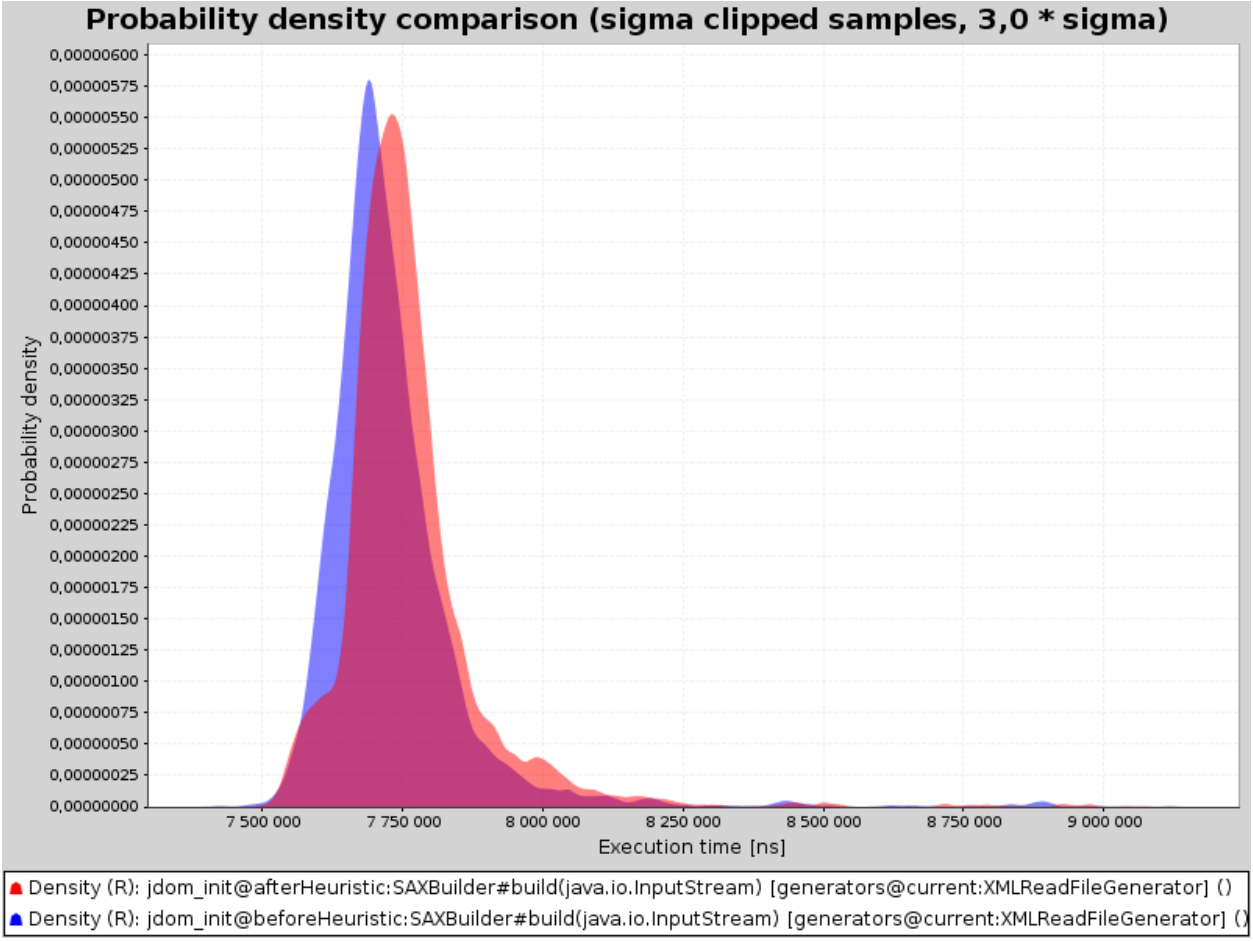


Figure 7.4: SAXBuilder runs almost the same after Verifier improvement, red graph is after update.

For this comparison are used this aliases:

```
beforeHeuristicSAX =
jdom_init@beforeHeuristic:
org.jdom.input.SAXBuilder()#build(java.io.InputStream)

afterHeuristicSAX =
jdom_init@afterHeuristic:
org.jdom.input.SAXBuilder()#build(java.io.InputStream)
```

In the first part is created alias **beforeHeuristicSAX** and its definition. The definition means from project **jdom_init** at revision **afterHeuristic** create instance of class **org.jdom.input.SAXBuilder** by constructor without parameters [4] and on this instance choose method **build** with one **java.io.InputStream** as its input parameter. The second part defines alias **afterHeuristicSAX** for the same method but from **afterHeuristic** revision.

The comparison of defined aliases is:

```
afterHeuristicSAX[gxmlread] = beforeHeuristicSAX[gxmlread]
```

It means that method for which is defined alias **afterHeuristicSAX** with generator **gxmlread** as input is compared with method for which is defined alias **beforeHeuristicSAX** with the same generator as input.

This comparison uses short cut = for equality comparing with default 5% tolerance interval. Details of this type of comparison are described in section [Comparison operators on page 6].

[4] Parenthesis () after class name are optional in this case. They have meaning only if constructor with String parameter should be used and then the parameter has to be specified inside them. For static methods or methods called on an instance created by constructor without parameters it doesn't matter if they are present or not. They are written there only for better readability to distinguish that this method is called on instance.

DOMBuilder

Also for creating XML structure using **DOMBuilder** the results are similar. Improvement of **Verifier** doesn't change its performance:

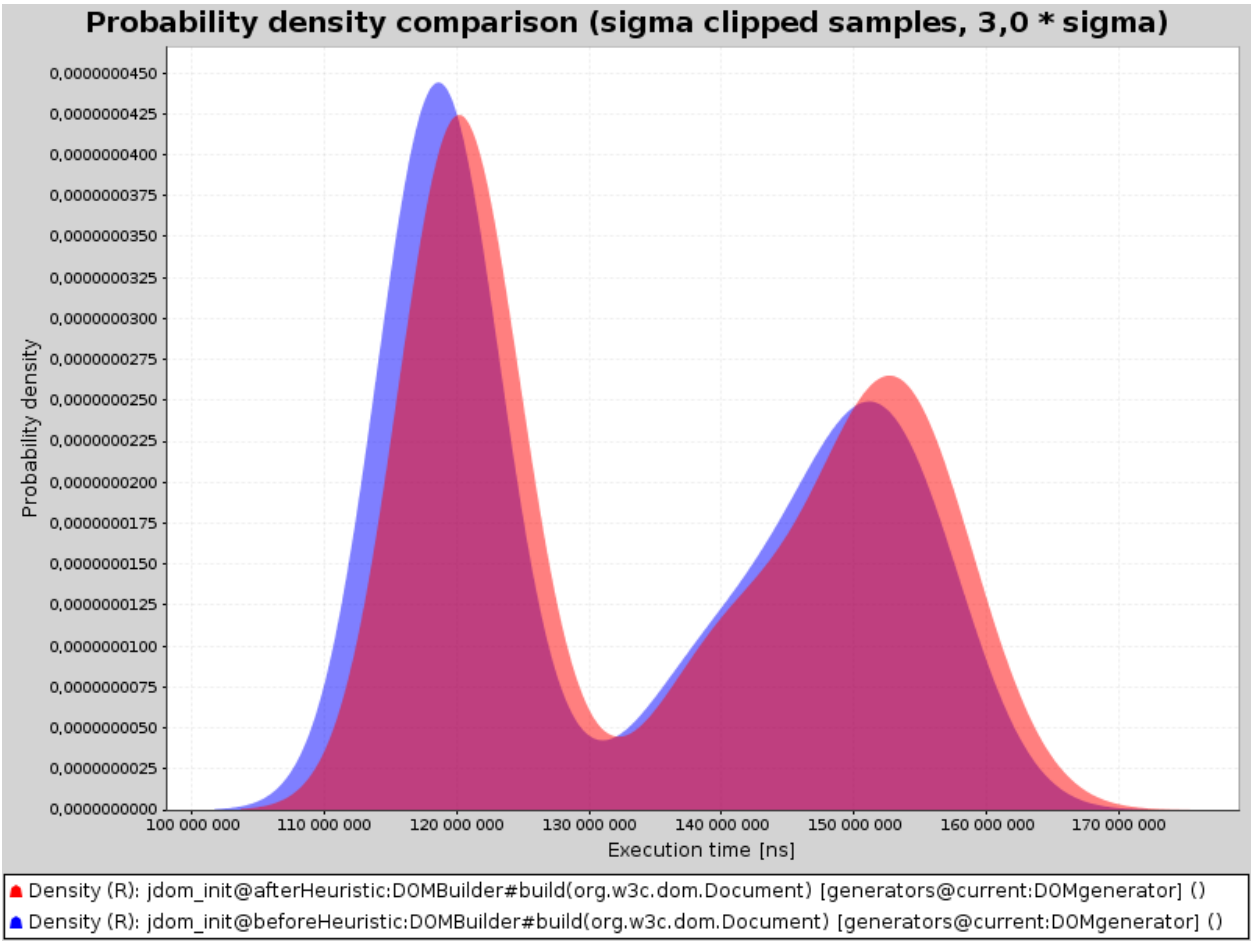


Figure 7.5: DOMBuilder runs almost the same after Verifier improvement, red graph is after update.

For which this aliases are used:

```
beforeHeuristicDOM =
jdom_init@beforeHeuristic:
org.jdom.input.DOMBuilder()#build(org.w3c.dom.Document domDocument)

afterHeuristicDOM =
jdom_init@afterHeuristic:
org.jdom.input.DOMBuilder()#build(org.w3c.dom.Document domDocument)
```

In the first part is created alias **beforeHeuristicDOM** and its definition. The definition means from project **jdom_init** at revision **beforeHeuristic** create instance of class **org.jdom.input.DOMBuilder** by constructor without parameters and on this instance choose method **build** with one **org.w3c.dom.Document** as its input parameter. The second part defines alias **afterHeuristicDOM** for the same method but from **afterHeuristic** revision.

The comparison of defined aliases is:

```
afterHeuristicDOM[gdomgenerator] = beforeHeuristicDOM[gdomgenerator]
```

It means that method for which is defined alias **afterHeuristicDOM** with generator **gdomgenerator** as input is compared with method for which is defined alias **beforeHeuristicDOM** with the same generator as input and the result should be equality with 5% tolerance interval as in previous SAXBuilder comparison.

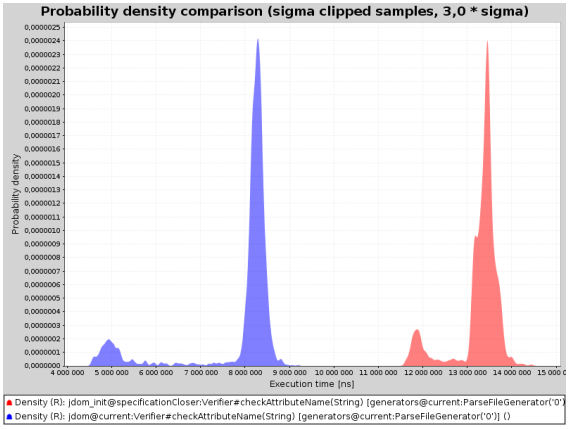
7.3.2 Successful speed up

Now have a look at the most successful updates where the JDOM developers improved performance. In this section *speedup* means result of

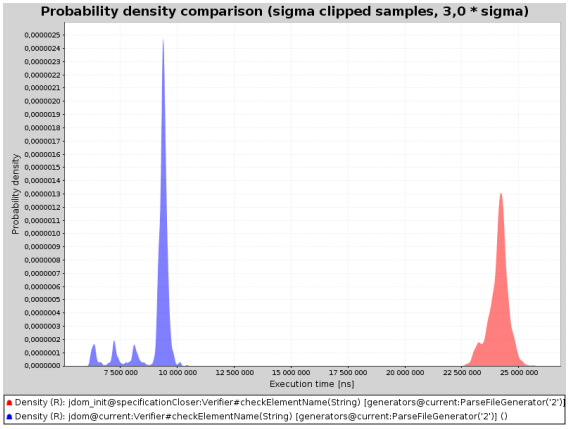
$$speedup = time_{old}/time_{new}$$

where $time_{old}$ means median of measured time of running the method from older revision and $time_{new}$ is median of measured time of running the method from newer revision.

To see performance of the **Verifier** in a long term aspect revision called **specificationCloser** from 23–7–2000 where it should be in line with specification is compared with **current** revision from 8–11–2012. Method **checkAttributeName** *speedup* was about 1.5 and method **checkElementName** *speedup* was from 1.6 to 2.2. **checkCharacterData** method cannot be measured because it wasn't present in **specificationCloser** revision yet.



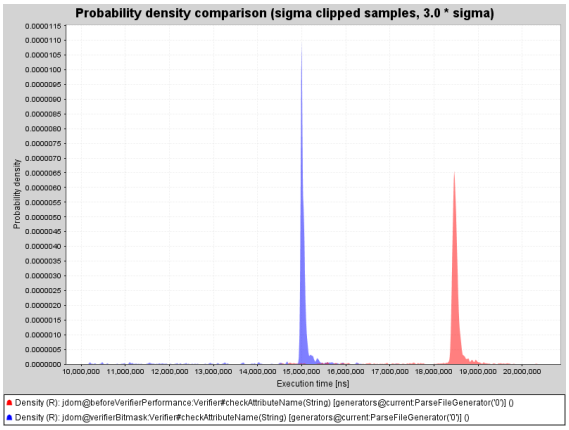
(a) checkAttributeName speed up, blue graph is newer.



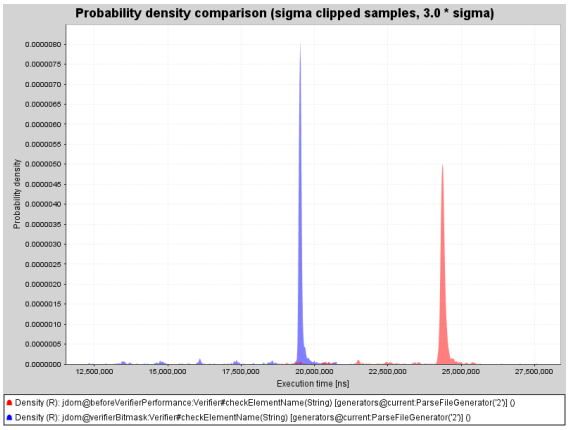
(b) checkElementName speed up, blue graph is newer.

Figure 7.6: Verifier speed up between revision **specificationCloser** and **current**

Another speed up and maybe the most visible in a short term aspect was made not long ago after the developers made their own performance testing aimed to improve **Verifier** performance. In comparison of revision **beforeVerifierPerformance** on 2–9–2012 and **verifierBitmask** on 3–9–2012 **checkAttributeName** method gain *speedup* about 1.2 and **checkElementName** *speedup* is 1.25.



(a) checkAttributeName speed up, blue graph is newer.



(b) checkElementName speed up, blue graph is newer.

Figure 7.7: Verifier speed up between revision **specificationCloser** and **current**

Also between revision **verifierBitmask** and **afterVerifierPerformance** on 9–9–2012 **checkCharacterData** method gain *speedup* about $1,25$.

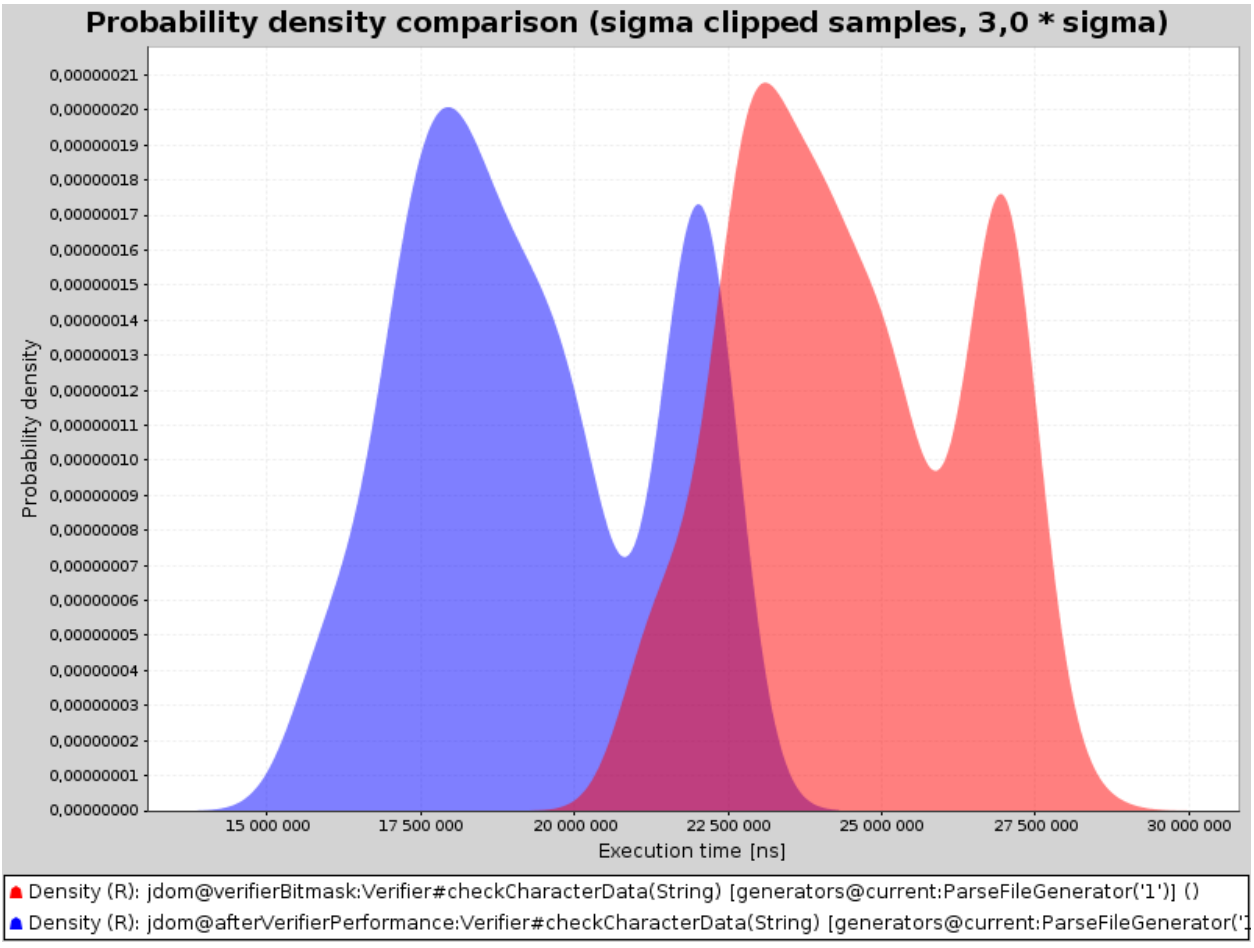
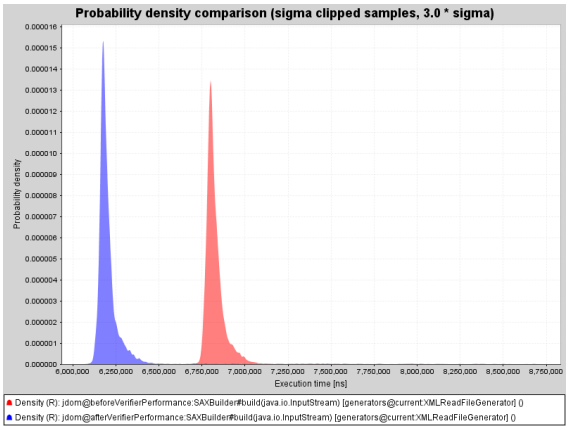
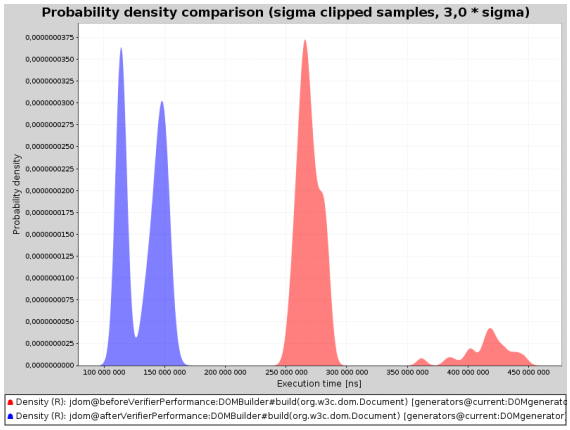


Figure 7.8: checkCharacterData speed up, blue graph is from newer revision.

This improvements reflected on the performance of **SAXBuilder** by *speedup* about 1.1 and **DOMBuilder** gain even further *speedup* about 2.



(a) SAXBuilder speed up, blue graph is newer.



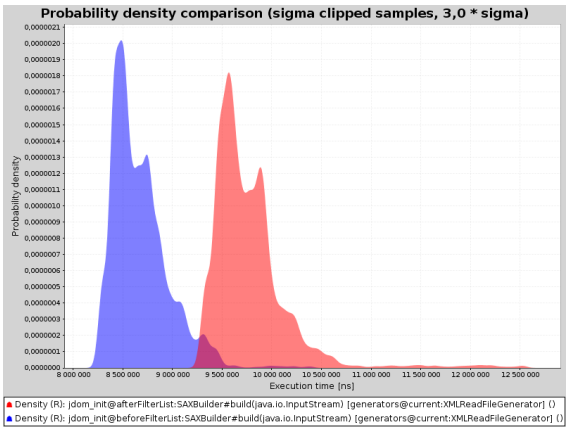
(b) DOMBuilder speed up, blue graph is newer.

Figure 7.9: Builders speed up between revision **beforeVerifierPerformance** and **afterVerifierPerformance**

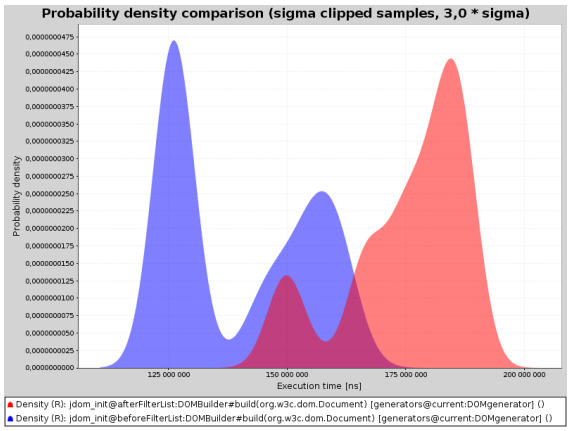
7.3.3 Dark side of updates

Not every update is successful and even if the change should speed up the software the result can be other. That is the case when *SPL Tools Framework* can be helpful maybe more than in a successful case. It is nice when framework is used to check that some change really speed up the software. However if it detects that some update slow down the program even when the developers doesn't expect it can be more practical and developers will know about it on time.

JDOM developers made such update between revisions **beforeFilterList** and **afterFilterList** both made on 11–12–2001 when the repository log says: "Now instead of using the slow and broken PartialList to make lists live, we'll be using a faster and smarter FilterList mechanism." But the measurement result of the builders was opposite. **SAXBuilder** *speedup* was about 0.85 and **DOMBuilder** *speedup* was 0.75.



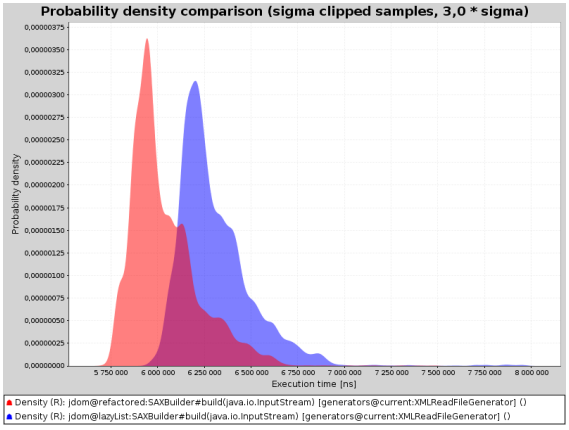
(a) SAXBuilder slow down, red graph is newer.



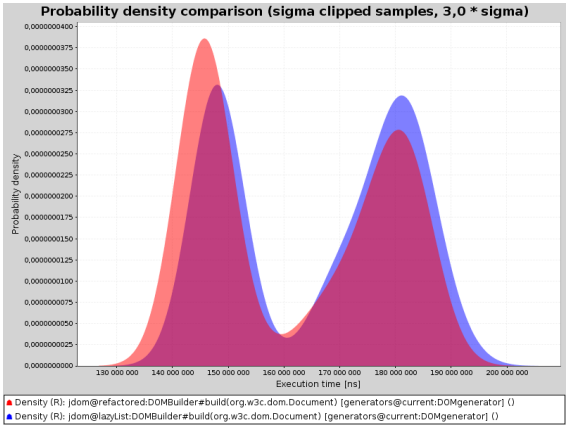
(b) DOMBuilder slow down, red graph is newer.

Figure 7.10: Builders slow down between revision **beforeFilterList** and **afterFilterList**

Another unsuccessful update was made between revision **refactored** on 2–10–2011 and **lazyList** on 11–10–2011 when log says: "The Re-Work includes a major performance upgrade because FilterList and FilterListIterator are now 'lazy' in the sense that they only check as much data as needed to satisfy the user request." However when builders was measured it shows that **SAXBuilder** *speedup* was about 0.94 and **DOMBuilder** *speedup* was 0.95.



(a) SAXBuilder slow down, blue graph is newer.



(b) DOMBuilder slow down, red graph is newer.

Figure 7.11: Builders slow down between revision **refactored** and **lazyList**

7.3.4 Unusual measurements

Measuring Java programs can be tricky and some results look differently than one can expect. In this section some of that behaviour is shown.

Usually it is expected that many measurement samples have most values at the bottom of the time diagram and only couple of samples differs and runs slower which can be caused by running garbage collector or some other program. But when measuring **Verifier**'s **checkAttributeName** method in revision **beforeHeuristic** on 19–12–2000 the diagram looks inverted. It can be caused by repeating the same code so few samples can hit between two collector runs but major part doesn't.

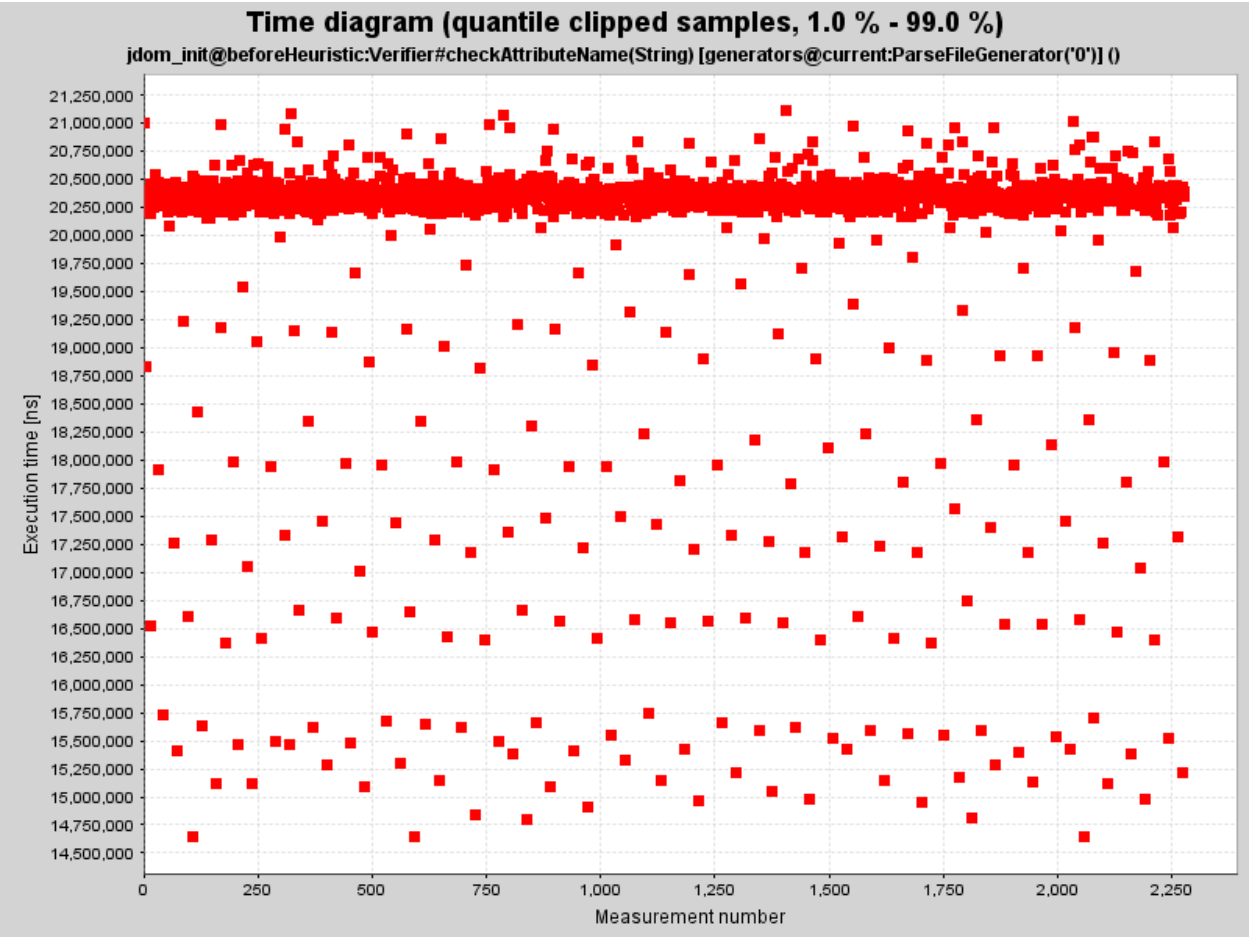


Figure 7.12: Verifier.checkAttributeName measurement time diagram looks inverted.

Some comparisons have repeating instability. When comparing **SAXBuilder** in revision **afterTransferring** on 22–6–2001 and **afterEarlyReturning** on 17–8–2001 the measurement in many cases looks similar and their median time is almost the same, their *speedup* is around 0.99.

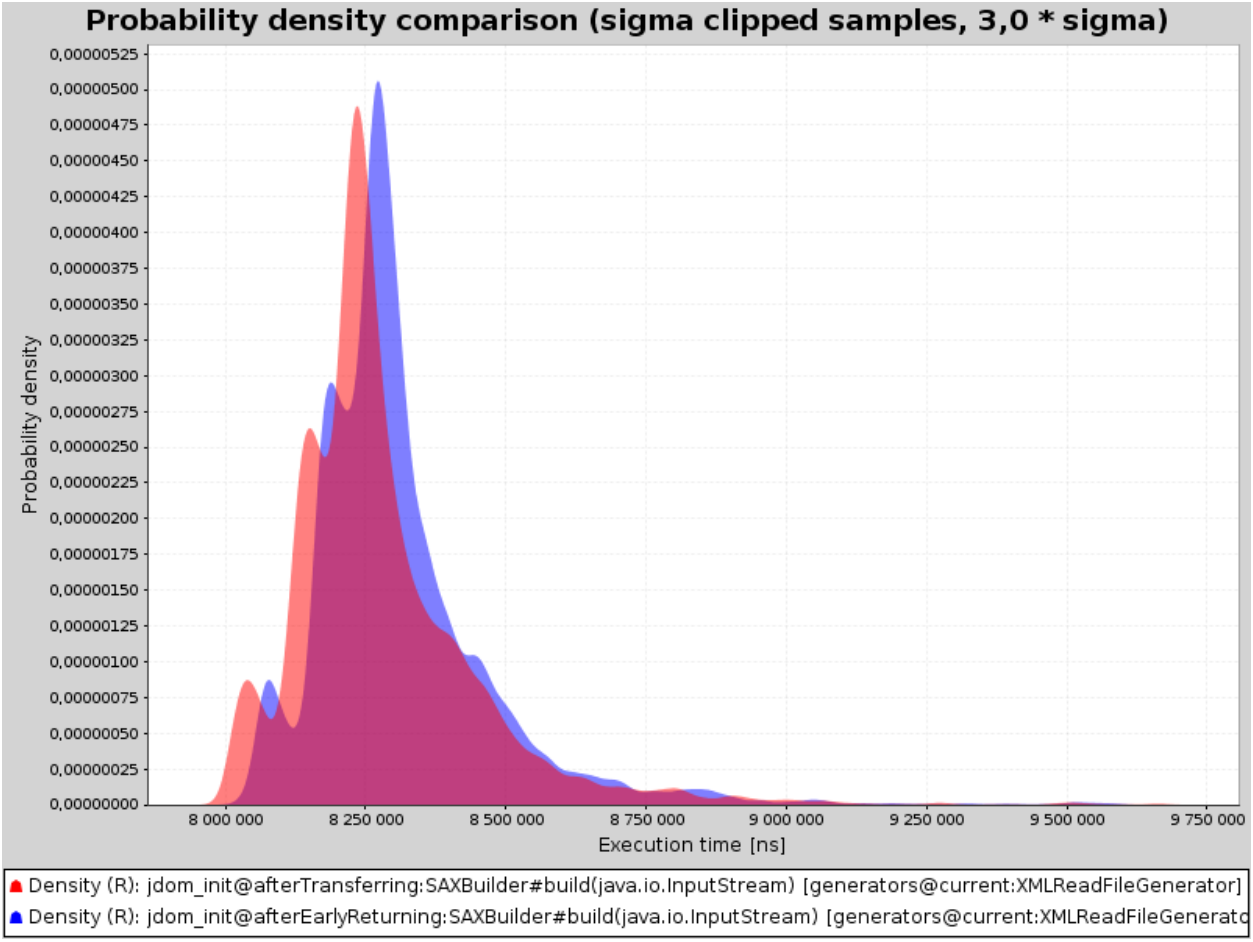


Figure 7.13: SAXBuilder runs similar in both revisions, blue graph is newer.

But on other computer with different operating system it repeatedly has *speedup* about 1.1. It can be caused by different processor cache size.

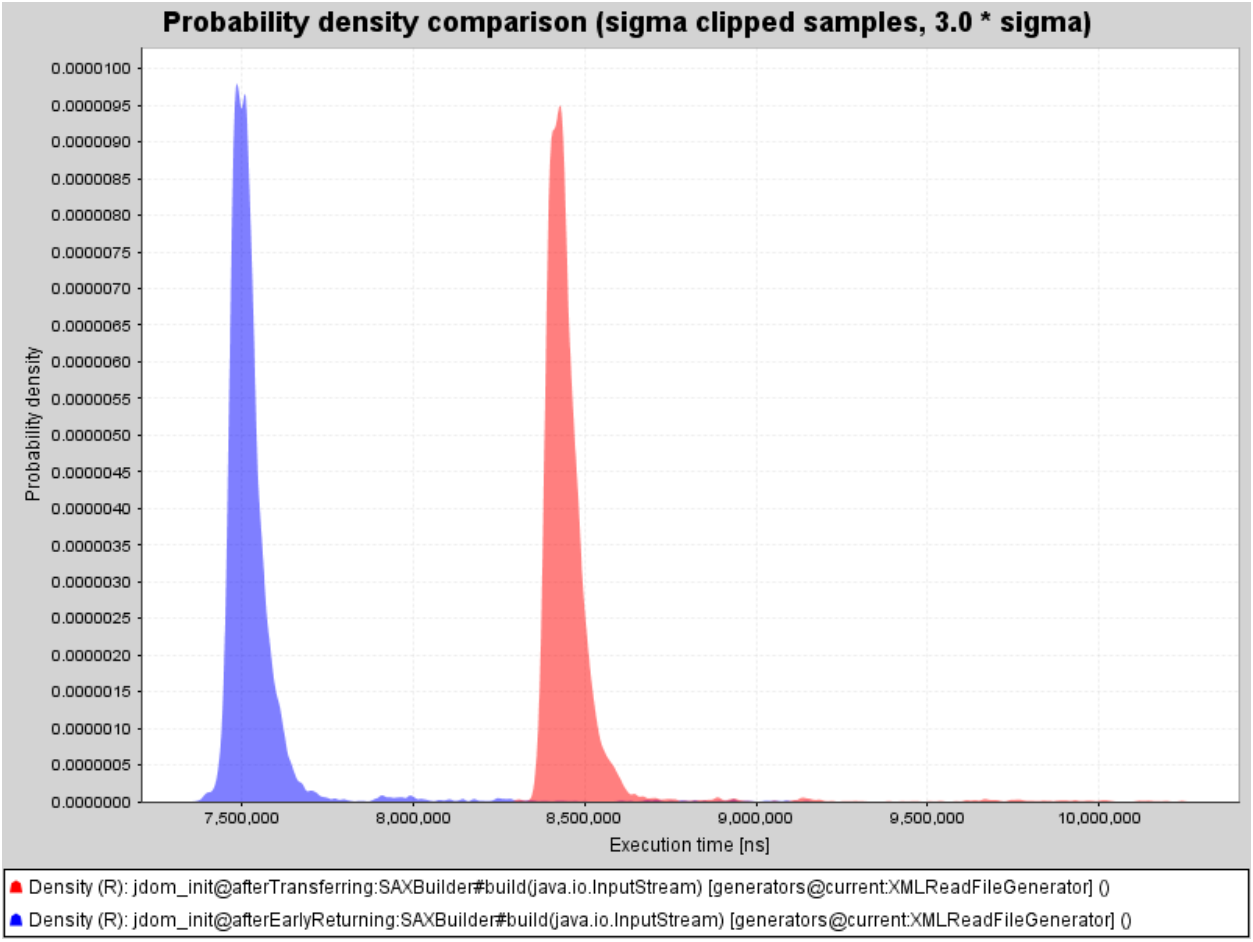


Figure 7.14: SAXBuilder runs different on other computer, blue graph is newer.

7.4 Contribution

In the case study is demonstrated how the *SPL Tools Framework* can be used on a real world application. Many revisions were inspected and for number of methods in different revisions annotations were written.

Thanks to measurement comparisons is easy to see performance impact of updates between revisions. Many changes have only small effect but some of them really speed up or on the other hand slow down the program. The most interesting results are part of the study and it is explained how annotations were written and what is their meaning.

SPL can be used to check that updates really speed up the software or to detect slow down even when developers don't expect it. It can be useful to check performance of the application from time to time because single change rarely affect performance notably. But every change can cause just a little slow down and if the downgrades accumulate over time suddenly there can be a problem.

Last but not least the study helped to test framework and it contributes to overall quality of the software by finding many bugs, detecting unexpected behaviour and propositioning improvements that make the program more user friendly.

If SPL eventually becomes part of JDOM project it should improve its performance mon-

itoring. Current formulas present in the study may serve as base for additional formulas. Developers can use study results for regressive examination of code changes and find origins of performance problems that could possibly arise. They can use it to insure that a performance update is really beneficial or to prove that the next release is faster than previous versions. New formulas based on examples in the study can be simply added for this purpose. Editing existing formulas is usually useful only after their creation if the measurement results are different than the expectation or if in formula is used master revision that is different after every commit. Other formulas use already measured data and if they aren't considerably rewritten their editing doesn't bring new value.

8. SPL Tools Eclipse Plug-in

The *SPL Tools Eclipse Plug-in* is integration of the *SPL Tools Framework* into the Eclipse IDE[1]. We will refer to the *SPL Tools Eclipse Plug-in* just as the *Plug-in* in this chapter. The *Plug-in* allows you to add and edit SPL annotations in your source code, perform SPL measurement and evaluation from within the Eclipse, browse the SPL results inside Eclipse and create and edit necessary configuration files for the *Framework*.

8.1 Installation

Plug-in is installed to Eclipse instance through Eclipse update site which is available on URL:

`http://sourceforge.net/projects/spl-tools/files/eclipseupdatesite/`

Follow these steps to install *SPL Tools Eclipse Plug-in*:

1. Start your Eclipse IDE.
2. Select menu **Help** and item **Install new software...**
3. Add Xtext update site (This step is not required for Eclipse Juno) as plug-in requires Xtext 2.3.0 or newer: `http://download.itemis.com/updates/releases/`
4. Add or allow default Eclipse update sites for your particular Eclipse release as plug-in uses Xtext and it requires Eclipse Modelling Tools. Those update sites are usually named like "Indigo" or "Juno" for your Eclipse release.
   - Eclipse Juno (4.2): `http://download.eclipse.org/releases/juno/`
   - Eclipse Indigo (3.7): `http://download.eclipse.org/releases/indigo/`
5. Add *Plug-in* update site: `http://sourceforge.net/projects/spl-tools/files/eclipseupdatesite/`
6. Select SPL Tools plug-in update site and wait for Eclipse to load available features.
7. Select ***SPL Tools Eclipse Plug-in Feature*** in group ***SPL Tools***
8. Finish *Plug-in* installation using Eclipse wizard.

Note: Eclipse Indigo (3.7) does not handle referenced updates sites integrated into features yet, so it is necessary to either manually install Xtext or add its update site as described above.

[1] Eclipse home page `http://www.eclipse.org/`

8.2 Provided views

The *Plug-in* provides number of views. All of them are described further in this chapter. Those views are:

SPL Annotation Overview
Displays information about annotations in currently opened Java editor.

SPL Execution View
Allows to run SPL Tools Framework execution directly from Eclipse.

SPL Results Overview
Displays results of SPL evaluation.

To see how to add a view to Eclipse perspective visit [Add SPL views to Eclipse perspective on page 69].

8.2.1 SPL Annotation Overview

This view shows overview about syntactical correctness of all SPL annotations which are present in document of currently selected editor.

You can have only one instance of this view per Eclipse perspective.

This view should be visible in every Java perspective, which is created after plug-in installation or just reset to defaults.

Presentation of annotations

Main part of this view is used for presentation of SPL annotations in currently opened Java source code editor. Annotations are presented in same order in which they occur in the document.

Information about annotations is presented in the tree structure where annotations have sub-nodes with declaration types (generator aliases, method aliases and formulas).

Declaration type nodes have particular declarations as their sub-nodes.

Particular declaration node can have multiple problem nodes associated with it when there are some errors or warnings during validation.

Validation result of every item in annotation presentation tree structure is marked with one of following validation result images:

-  OK - item is valid
-  Warning - item is valid, but the *Framework* found some warnings
-  Error - item is not valid because the*Framework* found some errors

Warnings and errors are typically issued by *SPL Tools Framework* annotation parser.

Warnings and errors indication is propagated from declarations to the presented annotation summary information. So when declaration is not valid (it contains an error), then annotation will be marked with Error image.

Available actions

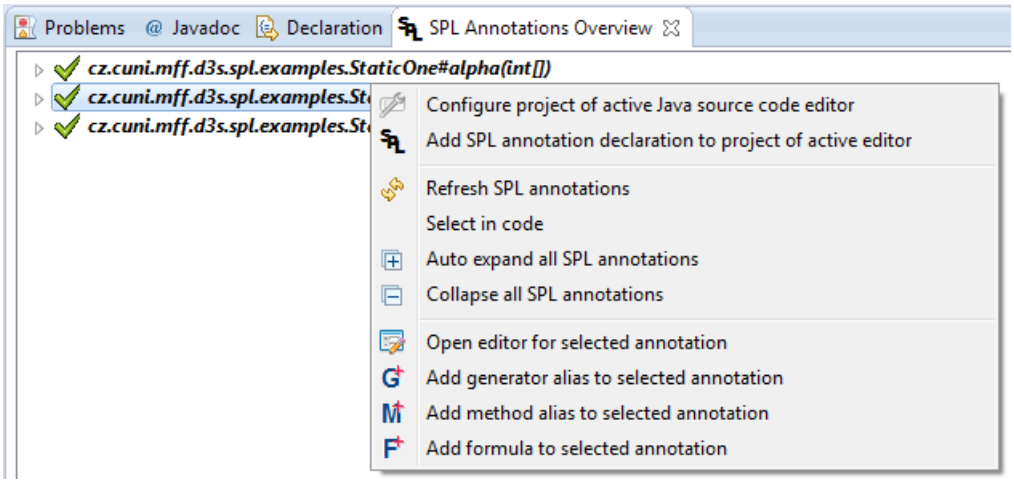


Figure 8.1: SPL Results Overview – available actions

Add SPL annotation declaration to project of active editor
Checks if project of active Java source code editor already contains declaration for SPL annotation `cz.cuni.mff.spl.SPL`. If this declaration is not present, then it adds source file `SPL.java` in package `cz.cuni.mff.spl` to source folder for project of active Java source code editor.

For more details see [Prepare project to use SPL on page 69].

Configure project of active Java source code editor
Allows to set and edit SPL configuration file for project of active Java source code editor.

Shows project browser with currently selected SPL configuration file for project of active Java source code editor. If no SPL configuration file is set for Java project, then default *spl-config.xml* is created in Java project root. After project configuration file is set, then **SPL Configuration Editor** is opened for it.

You can add new SPL configuration file using Eclipse new wizard in Package explorer - find category **SPL Tools** and select **New SPL Tools XML configuration file**.

Select in code
Selects code in editor for currently selected item in annotation presenter tree.

Refresh SPL Annotations
Refreshes shown SPL annotations. Parses document in currently selected Java source code editor and validates all annotations in it. When no Java source code editor is selected, then it does nothing.

Annotations are automatically refreshed on background.

Auto expand all SPL annotations
All shown SPL annotation and their information will be automatically expanded when this action is checked. Default behaviour is to expand only path to errors and warnings and to expand the annotation fully when just one is present.

Collapse all SPL annotations

Collapses all shown SPL annotations. Disables **Auto expand all SPL annotations** if it was enabled before.

Open editor for selected annotation

Shows dialog allowing to edit annotation selected in annotation presenter tree. Editor automatically edits the selected declaration inside annotation overview tree.

You can also double click on declaration you want to edit.

Add generator alias to selected annotation

Shows dialog for adding generator alias to annotation selected in annotation presenter tree.

Add method alias to selected annotation

Shows dialog for adding method alias to annotation selected in annotation presenter tree.

Add formula to selected annotation

Shows dialog for adding formula to annotation selected in annotation presenter tree.

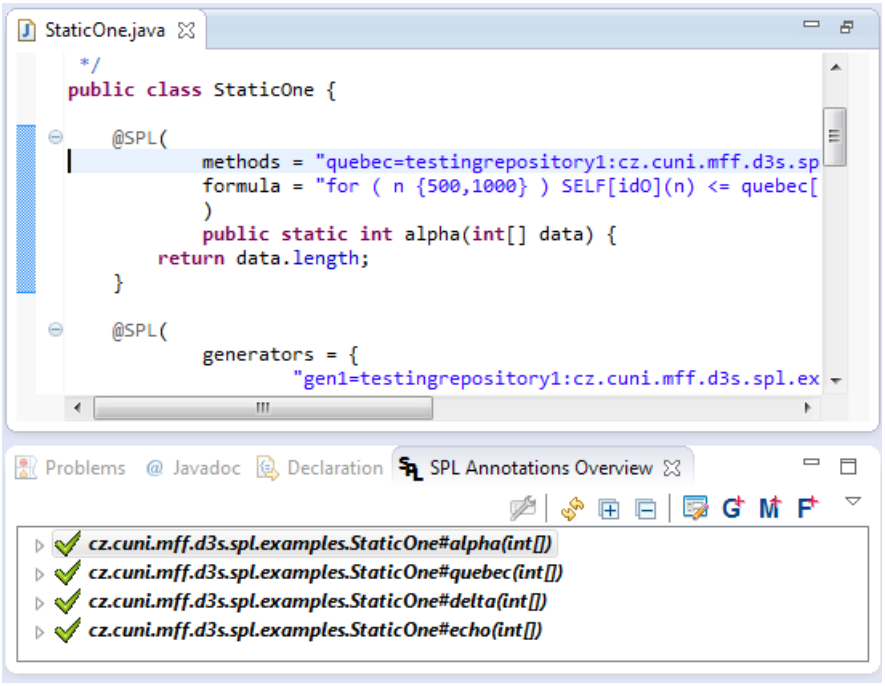


Figure 8.2: SPL Annotation Overview

Note: If you place annotation to anonymous inner class, then its name in the annotation list will be just ***<anonymous class>****.*

8.2.2 SPL Execution View

This view allow you to execute the *Framework* from within Eclipse.

This view has three tabs. First allows you to specify configuration to run execution. Second serves to observe running execution and third is used to browse execution results.

Configuration tab is depicted on following image. For more details on how to run execution see [Run execution from Eclipse on page 74].

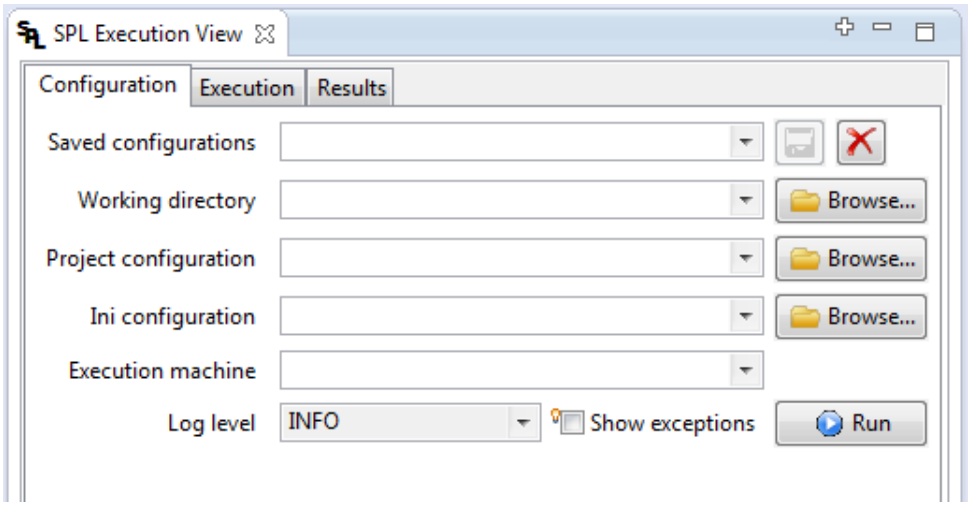


Figure 8.3: SPL Execution View – execution configuration

Tab with execution progress is shown on following image.

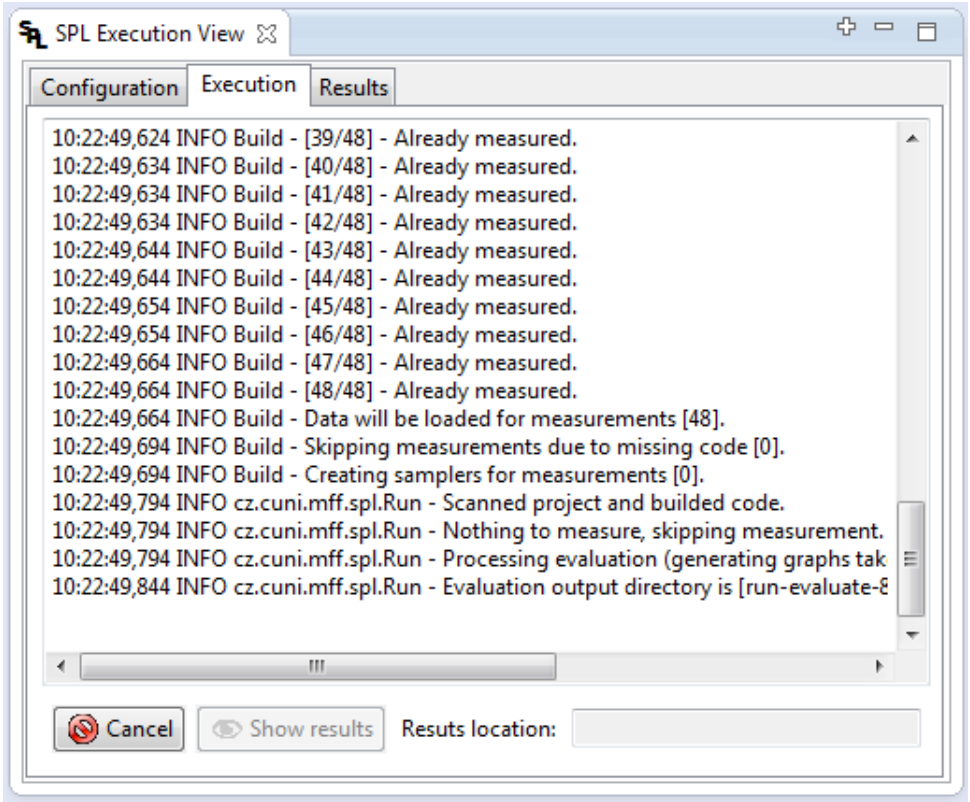


Figure 8.4: SPL Execution View – execution in progress

This tab shows console output of the *Framework* execution and allows you to cancel it or to show the execution results when the execution is finished. The execution results are then shown on the third tab which contains the same content as **SPL Results Overview** which is described in the next section.

8.2.3 SPL Results Overview

This view allows to browse SPL execution results interactively inside Eclipse.

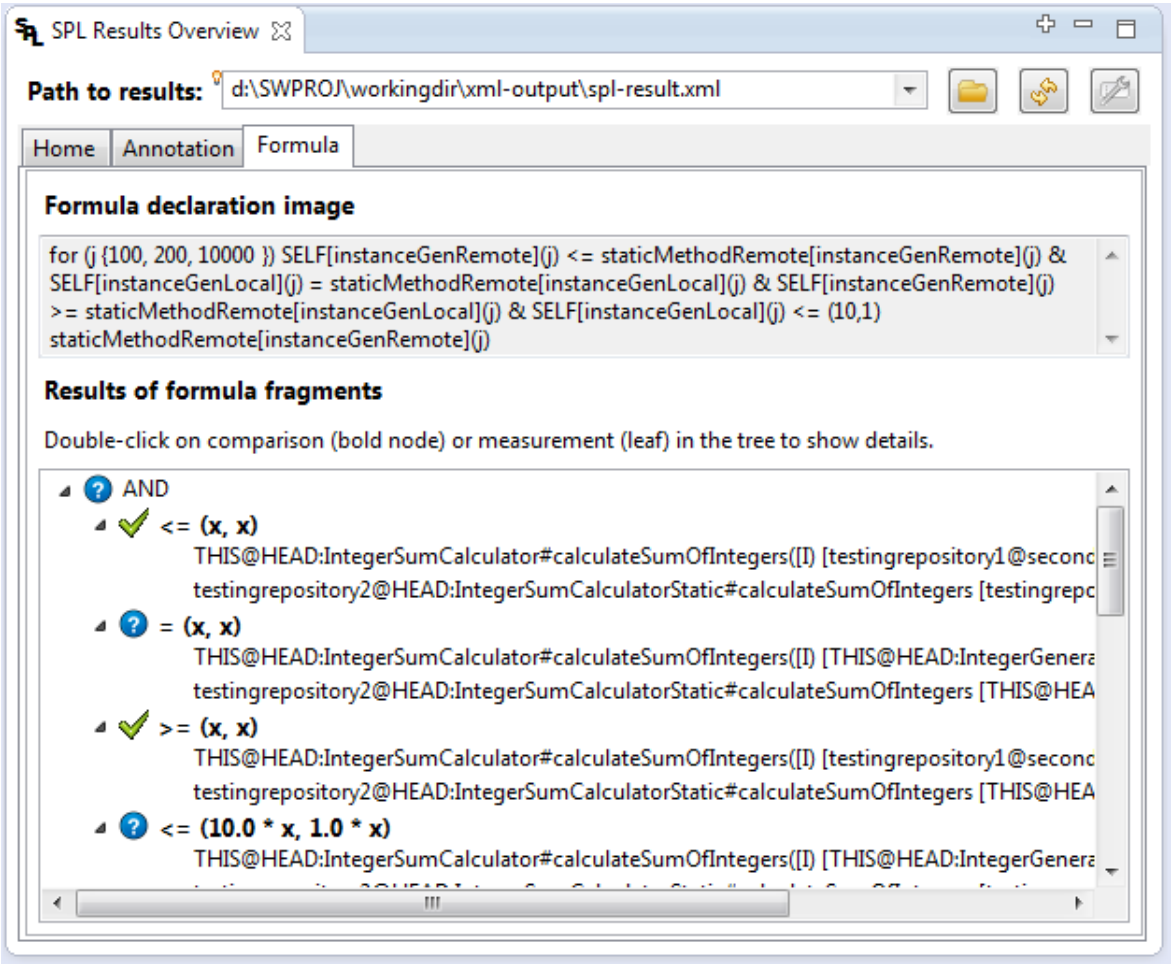


Figure 8.5: SPL Results Overview

For detail on how to use this view see [View results of execution on page 75].

8.3 Provided editors

Plug-in provides following editors:

SPL Configuration Editor
: Allows editing the XML configuration file (its format is described in the section [XML configuration file on page 23]).

INI Configuration Editor
: Allows editing the INI configuration file (its format is described in the section [INI configuration file on page 30]).

Both editors are associated with specific file names only. To see how to open an arbitrary file with user-specified editor visit [Open file with specific editor on page 70].

Editors save changes in the same way as other Eclipse editors by clicking on menu **File** and there click on **Save** item or by pressing the keyboard shortcut **CTRL+S**. If editor detects error it indicates it by different color or picture and if the content is saved with error then the saved file is marked with error sign.

8.3.1 SPL Configuration Editor

The editor should be automatically associated with files that have name **spl-config.xml** or **spl.xml** or have **spl-xml** extension.

This editor allows editing referenced projects, global generators, global methods and parameters. There is one tab for each item on the bottom of the editor.

Projects Configuration

First tab is used for editing projects configuration and is shown on following figure.

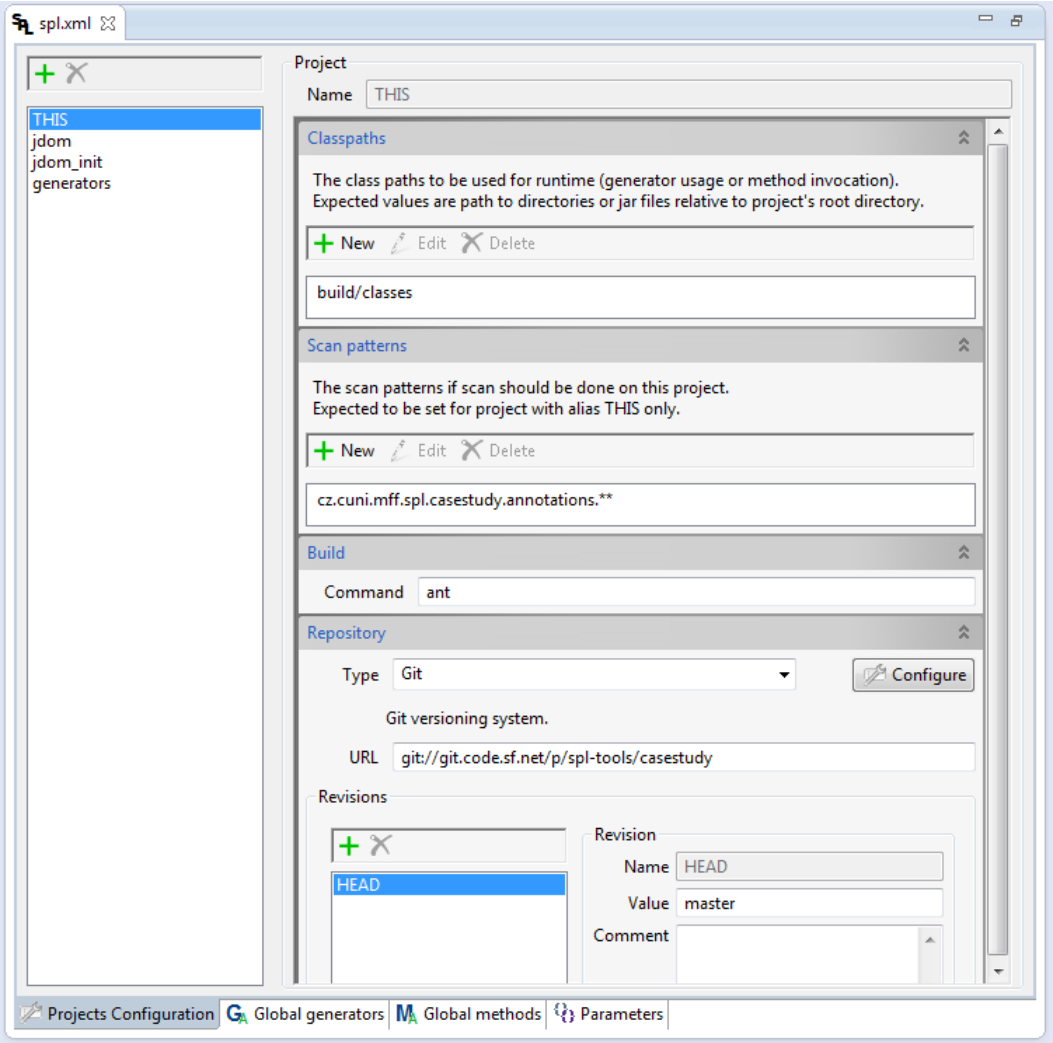


Figure 8.6: SPL Configuration Editor – Projects configuration tab

On the left side is the list of all projects. The tool bar with buttons for creating new and

deleting existing project is above the list. For editing single project it is needed to choose it in the list. Project with name **THIS** is the project with SPL annotations and cannot be deleted or renamed.

On the right side in the frame named **Project** is field **Name** for editing project's name which is used in annotations to identify project and cannot be empty and under it are four expandable items:

Classpaths
: There is list of classpaths where framework will search for classes during measurement. New classpaths can be added and existing ones can be edited or deleted. For editing new or existing classpath new dialog appears where the classpath value have to be written and it cannot be empty.

Scan patterns
: This part is similar like **Classpath** part but for editing scan patterns which are used by scanner when annotations are searched. In this version patterns have sense only for **THIS** project. In other project they are ignored. Pattern value cannot be empty.

Build
: Its only value is a command used for building this project.

Repository
: Project repository including its revisions is configured here.

  Type field serves for choosing repository type from predefined values. Another value can be written here too but it may result to problems when the project will be used and the *Framework* will not support specified type of repository.

  Next to repository type is button **Configure** which opens dialog for detailed repository configuration. Which values can be configured there depends on the repository type. The correct type must be chosen before opening the repository configuration.

  These values are saved into the INI file so it is necessary to chose proper INI configuration file in the pop-up dialog first and after then the new dialog for repository details configuration appears. This dialog is the same as one tab of INI editor described in section [INI Configuration Editor on page 67].

  In **URL** field it is specified URL address or local path of the repository. Value cannot be empty.

  In the frame named **Revisions** on the left side is list of all revisions which can be measured in this projects and on the right side is configuration of a single revision. Over the revision list is toolbar for adding and deleting revisions.

  Existing revision can be edited after it is chosen in the revision list and its values are shown in frame named **Revision**. There is revision name in **Name** field which is used in annotations to identify revision and cannot be empty, **Value** field which identifies revision in the repository [2] and **Comment** field with optional revision description.

  Revision with name **HEAD** represents the most current revision in the repository and cannot be deleted or renamed. Adding revision doesn't make sense for **Source** and **SourceRelative** kind of repository and revisions are ignored for them.

[2] Its value depends on repository type. It can be label, hash for Git or number for Subversion.

Global generators, global methods

Global generator and method aliases can be configured on those tabs. Their usage and behaviour is the same as the generator, method and formula tabs of the **SPL Annotation editor**. Its usage is described in the section [Edit annotation on page 72].

Parameters

Parameters that can be used in annotation formulas in lambda expressions and as generator parameters are defined here. The tab is shown on following image.

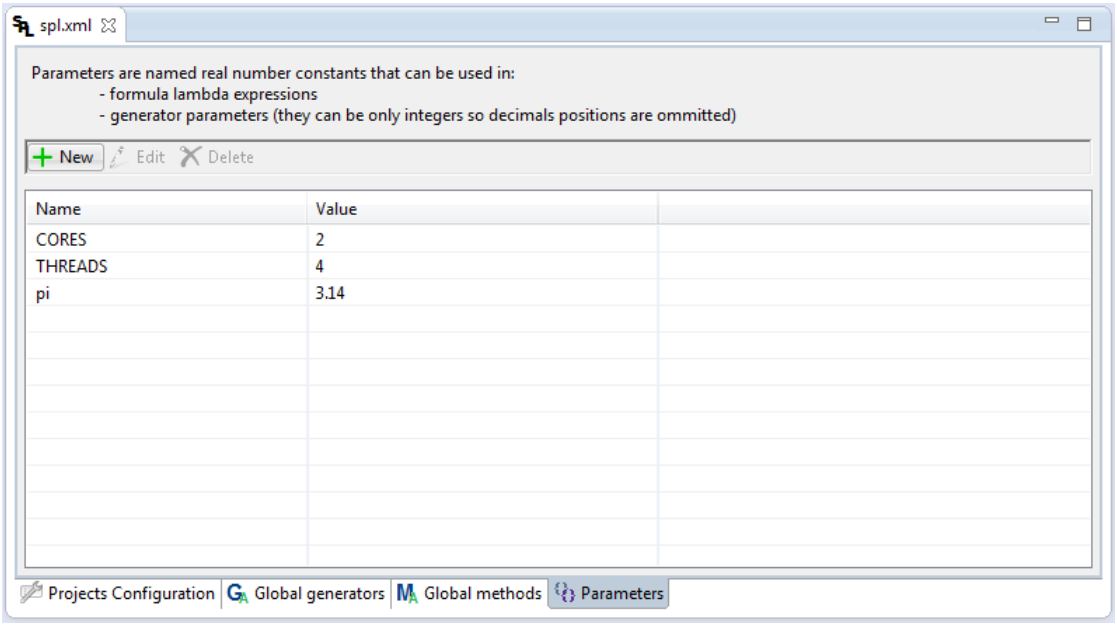


Figure 8.7: SPL Configuration Editor – parameter tab

There is a table with parameters names and their values. Every parameter can have real number as its value. Parameters can be added, edited and deleted using tool bar above the table. A new dialog is shown for editing parameter where name and value can be modified. Name nor value can be left empty.

8.3.2 INI Configuration Editor

The editor should be automatically associated with files that have name **spl-config.ini** or **spl.ini** or have **spl-ini** extension.

The editor allows configuring additional details of measurement process and is shown on the next figure.

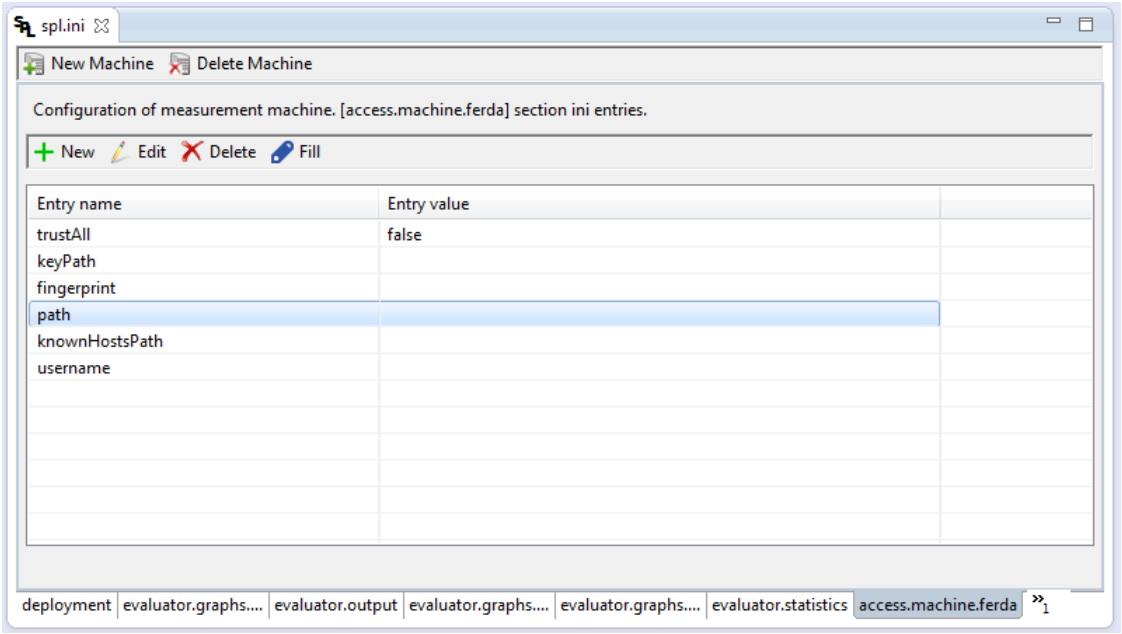


Figure 8.8: INI Configuration Editor

On the bottom is tab for every section in the configured INI file. The only exception is repository access configuration which have to be configured using XML configuration editor because repository type is stored in the XML configuration file and it is needed for creating right access configuration which is described in the section [Projects Configuration on page 66].

The list of sections is defined and cannot be changed except section that represents machines where measurements run. These sections have name with prefix **access.machine.** followed by the machine name. Machines can be added and removed using the toolbar at the top of the editor.

Every tab of the editor has table with names and values of section entries. Entries can be added, edited and deleted using toolbar right above the table. The last button of the toolbar is **Fill** and it fills in all default section entries with their default values which are not yet present in the section.

For editing single entry new dialog appears. It is displayed in the next picture.

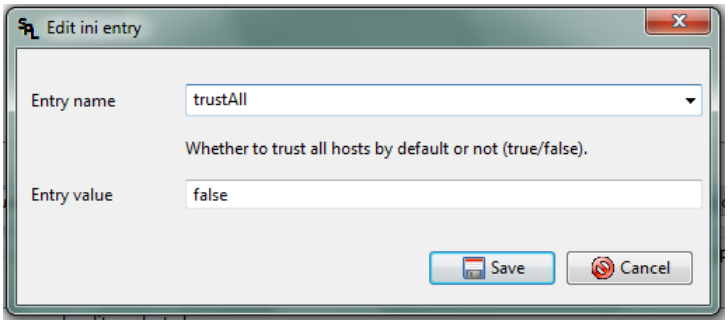


Figure 8.9: INI Configuration Editor - single entry dialog

There is field where name of the entry can be chosen from predefined values or user can write own entry name but that entry doesn't need to be supported by framework. Name cannot

be empty. Under that is label with description of the entry with this name followed by a field to enter the entry value.

8.4 How to...

This section describes what you can do with the *Plug-in* in and how to do it. It contains description on how to add SPL views to the Eclipse, how to prepare your project for usage of SPL, configure proper SPL context for the project, how to edit SPL annotations using provided editor dialog, how to run the SPL execution directly from within the Eclipse and how to browse the SPL evaluation results inside Eclipse.

8.4.1 Add SPL views to Eclipse perspective

Follow these steps to add one of SPL views to currently opened perspective:

1. Select menu **Window**, item **Show view** and item **Other...** at the bottom of shown sub-menu.
2. Either select group **SPL Tools** and item **SPL Annotation Overview** or search for **SPL** using integrated filter.
3. Confirm adding view with **OK** button.

Note that if you have opened standard Eclipse Java perspective, than all SPL related views are available directly in ***Show view*** *menu from step 1.*

If you want to have multiple views of same type in one Eclipse perspective (applies to ***SPL Execution View*** and to ***SPL Results Overview***), you need to use ✣ button in top right corner of already existing view.

8.4.2 Prepare project to use SPL

To use SPL annotations inside your project, you need to add a new dependency to it to allow Java compiler to recognize SPL annotations (Java type *cz.cuni.mff.spl.SPL*). You can either add entire *Framework*to your class path (but this includes more than 25 MB of data), or add just one source file to your project with SPL annotation type declaration.

You can use context menu entry ***Add SPL annotation declaration to project of active editor*** of view **SPL Annotation Overview** to create this file automatically.

SPL annotation file has to have this content (JavaDoc comments can be omitted):

```
package cz.cuni.mff.spl;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.annotation.ElementType;

/** Main SPL annotation. */
@Target(ElementType.METHOD)
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface SPL {
   /** SPL formula declarations. */
   String[] formula() default {};
   /**  Method aliases declarations. */
   String[] methods() default {};
   /** Generator aliases declarations. */
   String[] generators() default {};
}
```

As you can see, this Java type has no methods that can influence your code and it is the only dependency in your project, that *SPL Tools Framework* requires and you would not be able to compile your code without this annotation type declaration.

8.4.3 Configure SPL context for your project

If you want to use generators of parameters for measurement or measure classes in different projects or revisions, you need to configure SPL context for *Plug-in* which will be used for validation of project and revision aliases.

Every project in Eclipse workspace can have SPL XML configuration file set[3].

To set SPL XML configuration file for project follow these steps:

1. Open **SPL Annotations Overview**.
2. Open any Java source file in project you want to configure (preferred file is annotated file but any Java source file can be opened).
3. Select action **Configure project of active Java source code editor** in **SPL Annotations Overview**.
4. Select configuration file in opened editor. If you open configuration for the first time, default **spl-config.xml** with bare configuration will be automatically created in the project root.
5. Confirm your selection and selected file will be opened in **SPL Configuration editor**. If you have target file already opened in different editor you need to open **SPL Configuration Editor** manually as Eclipse will not open new editor on same input file automatically.

8.4.4 Open file with specific editor

To open file with specific editor follow these steps:

1. Click on that file with right mouse button in **Package Explorer** view.
2. Drop down **Open With** item and select **Other...**.
3. Choose demanded editor in the appeared dialog and confirm selection.

[3]Selected SPL XML configuration file is saved to project's persistent properties.

8.4.5 Add aliases or formulas to annotation

You have following options to add aliases and formulas to annotation in Java source code file:

1. You can fill them manually in Java source code editor with help of **SPL Annotation Overview** as validator, but it refreshes shown information with delay to prevent slowing down entire Eclipse.
2. You can use one of actions provided by the **SPL Annotation Overview**. This is briefly described in this chapter.
3. You can use **SPL Annotation Editor**. See next section for details.

If you want to add single declaration to SPL annotation, then you need to select it in the **SPL Annotation Overview** and invoke one of actions (either by button, or from context menu):

Add generator alias to selected annotation
Shows dialog for adding generator alias to annotation selected in annotation presenter tree.

Add method alias to selected annotation
Shows dialog for adding method alias to annotation selected in annotation presenter tree.

Add formula to selected annotation
Shows dialog for adding formula to annotation selected in annotation presenter tree.

Following figure illustrates dialog for adding new generator alias.

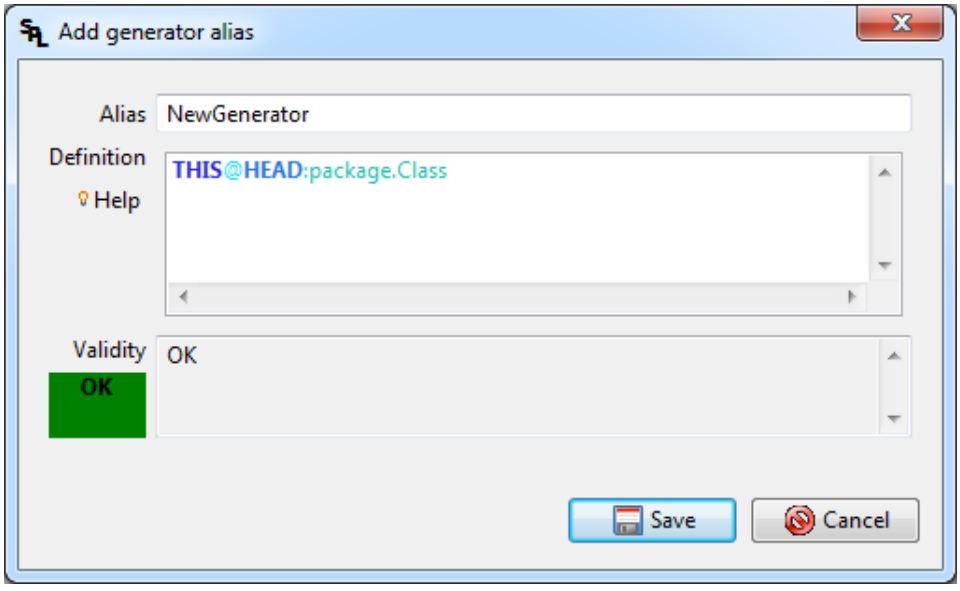


Figure 8.10: Add generator alias dialog

The definition text area has content assist support which can be shown with key shortcut `Ctrl+Space`. If you select item from the list and additional help for it is available, than little **Help** with bulb icon appears left of declaration field and you can hover over it to show the help.

When you are finished, you can save new declaration using dialog **Save** button located in bottom right. Note that this button is enabled only when none of generator alias editor, method alias editor and formula editor is in declaration editation state.

If you don't want to save changes, you can either close dialog with **Cancel** button located in bottom right, or close its window directly.

8.4.6 Edit annotation

The are two ways to open **SPL Annotation Editor**:

- Double click on annotation in the **SPL Annotation Overview**
- Select annotation in the **SPL Annotation Overview** and click on action **Open editor for selected annotation**.

Annotation editation dialog will be shown. If you have selected any particular declaration in annotation overview tree, than editation dialog will open its editor for you automatically.

Following figure shows editation dialog with formulas overview. Note that in example figure below is just one formula and its state is valid.

We will describe usage of this dialog on formula editor as generator and method aliases editors are used in the same way.

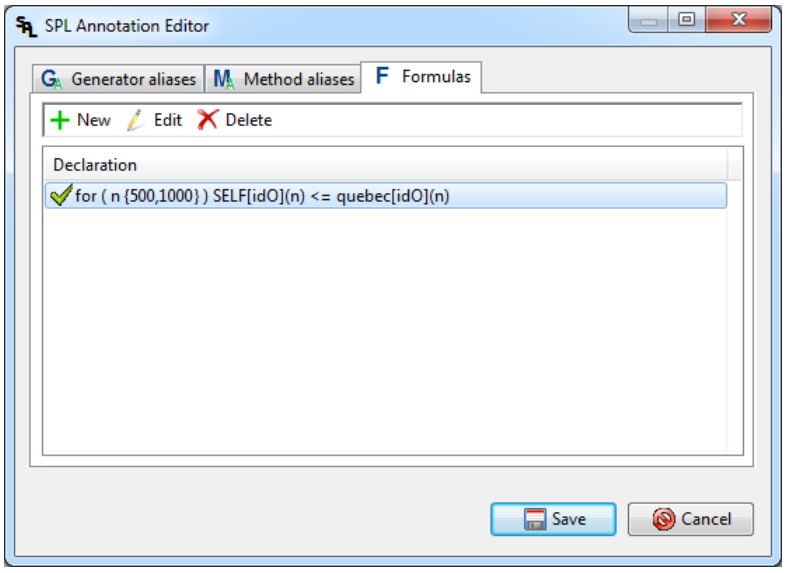


Figure 8.11: SPL Annotation Editor – formulas overview

Overview tool bar buttons:

New button +
: Switches view to editor for adding new formula.

Edit button
: Shows editor for selected formula. Double click on formula in the list has the same effect.

Delete button
Deletes selected formula.

If you add new formula or select formula for editation the following editor will be displayed.

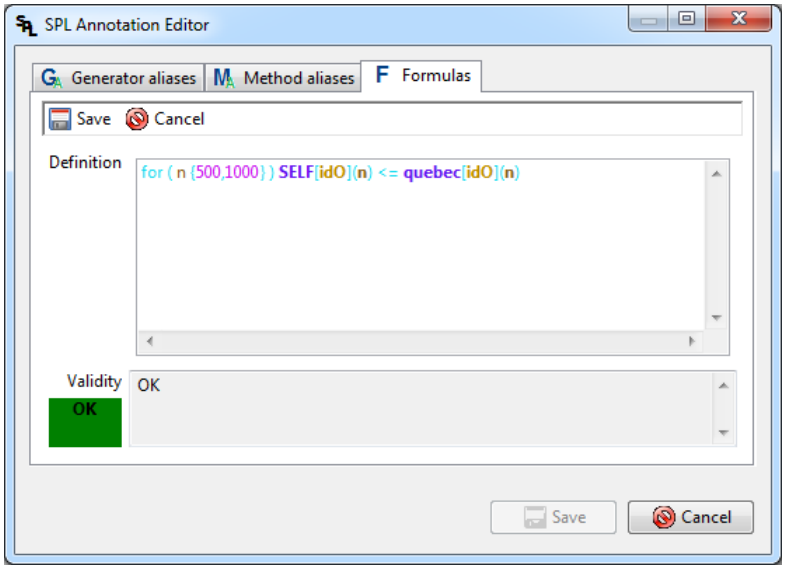


Figure 8.12: SPL Annotation Editor – formula editor

Formula editor contains tool bar, text field for edited definition and area for validity information presentation.

The definition text area has content assist support which can be shown with key shortcut **Ctrl+Space**. If you select item from the list and additional help is available for it, than **Help** text with little bulb icon appears left of the declaration field and you can hover over it to show the help.

Editor tool bar buttons:

Save button
Saves changes in formula and switches view to formula overview.

Cancel button
Does not save any changes in formula and switches view to formula overview.

When you are finished, you can save all changes using dialog **Save** button located in bottom right. Note that this button is enabled only when none of generator alias editor, method alias editor and formula editor is in declaration editation state.

If you don't want to save changes, you can either close dialog with **Cancel** button located in bottom right, or close its window directly.

8.4.7 Configure Plug-in

To open configuration of the *Plug-in* follow these steps:

1. Click on menu **Window** and select **Preferences**.
2. Expand **SPL Tools** item in the list of preferences.

3. There can be configured syntax coloring and templates used in editors of formulas and generator and method aliases.

8.4.8 Run execution from Eclipse

You can execute *Framework* directly from Eclipse in ***SPL Execution View***.

You need to specify configuration for the execution. This is done in the ***Configuration*** tab.

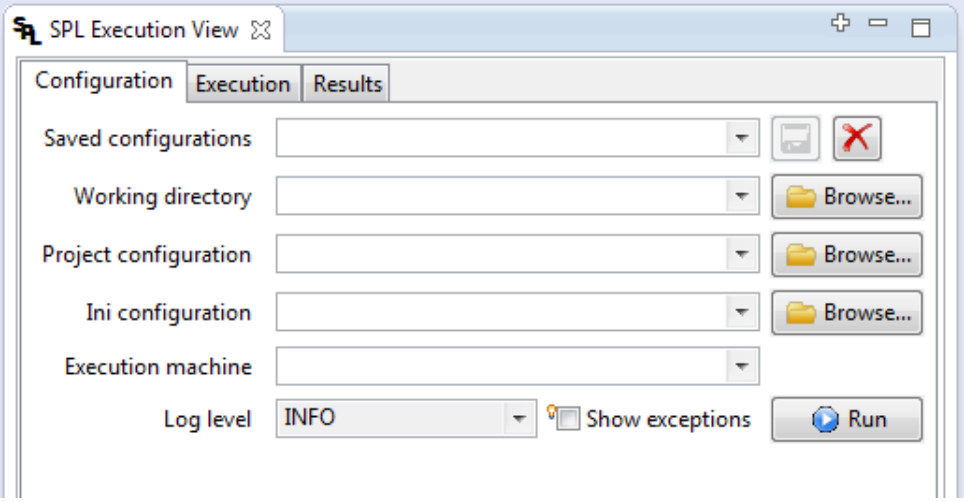


Figure 8.13: SPL Execution View – execution configuration

Configuration fields description:

Saved configurations
: You can save filled configuration values under some name using the save button or remove the selected one with the delete button .

  When you select name of previously saved configuration, than the values of the other fields will be set to the saved ones.

Working directory
: Path to folder which can be used as working directory for *Framework* execution.

Project configuration
: Path to SPL project configuration XML file (**spl-config.xml**).

Ini configuration
: Path to INI configuration file with additional configuration (such as remote measurement machine access details, see [INI configuration file on page 30] for details). Optional.

Execution machine
: Identification of machine where execution is to be done. Access details are obtained from INI configuration file which has to be set when this option is used. Optional.

Log level
: This sets which progress messages should be shown during processing execution. Log detail level value can be selected from following values ***NONE***, ***FATAL***, ***ERROR***,

WARN, ***INFO***, ***DEBUG***, ***TRACE*** which are ordered from most brief to most exhausting verbosity of details provided. Default value is ***INFO***.

Show exceptions
: Exceptions thrown during execution will be shown on ***Execution*** tab only when this option is checked. This is useful for debugging the *Framework*. Default is not checked.

You can start execution with ***Run*** button.

Running execution can be stopped with ***Cancel*** button.

Results of finished execution can be easily displayed in the ***Results*** tab with the ***Show results*** button. Note that this button is enabled only when execution finished and not when it was cancelled or when some fatal error occurred during execution.

8.4.9 View results of execution

First thing you need to do to view *Framework* execution results in ***SPL Results Overview*** is to set path to results XML description file (**spl-result.xml**) to the ***Path to results*** text field.

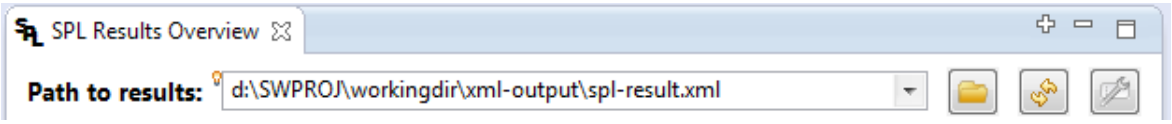


Figure 8.14: SPL Results Overview – path to results XML file

This file can be either local file or file accessible over HTTP (for example result of *Framework* execution on Hudson). Confirm path either by pressing Enter key in path to results text box or press ***Refresh*** button. Previously selected result locations can be selected from the list which is accessible through down arrow button (path to results are ordered by last use).

Use ***Browse*** button to select local file.

Use ***Configure*** button to set graphs shown in comparison or measurement detail tab and path to **Rscript** executable which is used to compute probability density functions.

Navigation to more detailed results can be done either with double-click, buttons or context menu which can be shown on table or tree entry (usually with right mouse click).

Double click on generator or method alias tries to show editor for declaration target (class or method).

Use generated navigation tabs to navigate to previously shown results (***Home***, ***Annotation***, ***Formula***, ***Comparison***, ***Measurement***).

Graphs shown on ***Comparison*** and ***Measurement*** tab are interactive when raw data for presented measurements are available (graph is marked with icon). When there are no raw measurement data available, than graph presenter can show only only a PNG image generated during evaluation (graph is marked with icon).

9. SPL Tools Hudson Plug-in

The **SPL Tools Hudson Plug-in** is integration of the *SPL Tools Framework* for the **Hudson Extensible continuous integration server**[1]. We will refer to **SPL Tools Hudson Plug-in** just as *Plug-in* in this chapter.

At the time of *Plug-in* development, Hudson latest production version was 2.2.1 and Hudson 3.0.0 was just in Release Candidate phase of life cycle. So plug-in is intended for **Hudson version 2.2.1** and tested with most recent 3.x version of Hudson at the time.

Plug-in allows you to run*SPL Tools Framework* evaluation on a regular basis for any Hudson job. Frequency of its execution is defined by the Hudson job configuration (for example on every commit or once a day).

9.1 Obtaining the *Plug-in* binary package

The preferred way of obtaining the *Plug-in* binary package is to download the latest version from the SPL Tools web site on the following URL:

`http://sourceforge.net/projects/spl-tools/files/HudsonPlugin/`

If you want to compile the package yourself than please refer to the **Stochastic Performance Logic Development Documentation** which contains full instructions on how to compile the *Plug-in*.

9.2 Installation

When you have file **spl-tools-hudson-plugin.hpi**[2] you can install it to an instance of Hudson.

Plug-in is compatible with Hudson version **2.2.1** and **3.0.0 RC**. Installation procedure for plug-ins in form of standalone file differs a little between those two versions.

9.2.1 Hudson 2.2.1

Login to Hudson as administrator with rights to add new plug-ins.

Go to administration, select item ***Manage plugins*** from the list, select tab **Advanced**. In section ***Upload Plugin*** select file **spl-tools-hudson-plugin.hpi** and confirm plug-in upload with button ***Upload***.

After file upload has been finished, Hudson shows tab ***Updates*** automatically, but it won't show any special message about upload result. The only indication about uploaded plug-in should be visible on tab ***Installed***, where you should see red message below installed plug-ins table saying ***New plugins will take effect once you restart Hudson***.

[1] Hudson web page `http://hudson-ci.org/`

[2] Actual name of *Plug-in* distribution file may also contain version information.

Restart your Hudson instance and check ***Installed*** plug-ins, where you should finally see entry for ***SPL Tools Hudson Plug-in***.

9.2.2 Hudson 3.0.0

Login to Hudson as administrator with rights to add new plug-ins.

Go to administration, select item ***Manage plugins*** from the list, select tab ***Advanced***. Select file **spl-tools-hudson-plugin.hpi** in section ***Manual Plugin Installation*** and confirm with button ***Upload***.

After file upload has been finished, Hudson shows message about upload result. When message says ***Plugin spl-tools-hudson-plugin.hpi sucessfully uploaded.***, than everything is all right.

Plug-in will be available after Hudson instance is restarted (this information is mentioned in description of section ***Manual Plugin Installation*** and also shown right of Advanced tab after plug-in upload).

Restart your Hudson instance and check ***Installed*** plug-ins, where you should finally see entry for ***SPL Tools Hudson Plug-in***.

9.3 Usage

This section describes the usage of the *Plug-in* from its configuration to the viewing of the SPL evaluation results.

9.3.1 Configuration

To use *Plug-in* for Hudson job you need to add new Build Step with name **SPL Tools Hudson Plug-in** to it.

You can add multiple **SPL Tools Hudson Plug-in** build steps to one Hudson job.

Basic configuration

Basic configuration requires only specification of path to SPL Tools projects configuration file[3] for configured Hudson job.

Specified path is expected to be relative to Hudson job workspace root, but you can also specify absolute path which points to any file on machine running Hudson.

Note that if specified path does not end with **.xml**, plug-in will show warning that file does not seem to be XML file.

[3] See [XML configuration file on page 23]

Advanced configuration

Advanced configuration allows to modify additional details of SPL Tools Framework execution behaviour. To show advanced configuration of SPL Tools Hudson Plug-in build step press button **Advanced...** and available advanced configuration options will be shown.

Advanced configuration allows to set following settings:

INI configuration file
: This option allows to specify path to SPL configuration INI file[4]. Path can be either relative to Hudson job workspace root, or absolute on machine running Hudson.

Working directory
: This option specifies path to SPL Tools Framework working directory [5] which is used to store measured values, evaluation results and temporary files needed for measurement (such as referenced projects and generated measurement code). Default value is **.spl-tools** (folder .spl-tools inside Hudson job workspace root). Working directory can't be Hudson job workspace root itself. You can also specify absolute path (such as **/tmp/hudson/myJobSpl**).

Execution machine ID
: This option allows to specify ID of execution machine, where measurements will be executed. Access information for remote machine (host name, port, user name, etc.) is retrieved from INI file. Default is empty value which means local execution.

Choose log details level
: This option allows to specify how many details about SPL Tools execution should be retrieved form SPL Tools Framework to build log file. Specify exactly one value from***NONE***, ***FATAL***, ***ERROR***, ***WARN***, ***INFO***, ***DEBUG***, ***TRACE***, ordered from most brief to most exhausting verbosity. Value can be entered in any combination of lower and upper case letters.

Show exceptions in output
: - This option toggles full stack trace for exceptions. This can be useful for finding various errors which occurred in *Framework* execution, but usually not needed (as the output is a lot longer with this option enabled). All stack traces of exceptions can be found in full debug log.

Hudson URL configuration

Plug-in uses Hudson server URL for navigation to results and providing URL for access from Eclipse.

If URL is miss-configured, you may not be able to use links generated by *Plug-in*.

To set URL or check its current value go to administration, select item ***Configure System*** from the list and find section ***E-mail Notification*** and find value ***Hudson URL***.

[4] See [INI configuration file on page 30]
[5] See [Working directory on page 21]

9.3.2 Execution results

SPL Tools Hudson Plug-in archives the *Framework* execution result to build folder and provides basic summary for each performed SPL build step you defined for Hudson job.

This summary is located in build status information and it has one of the forms shown on following figure.

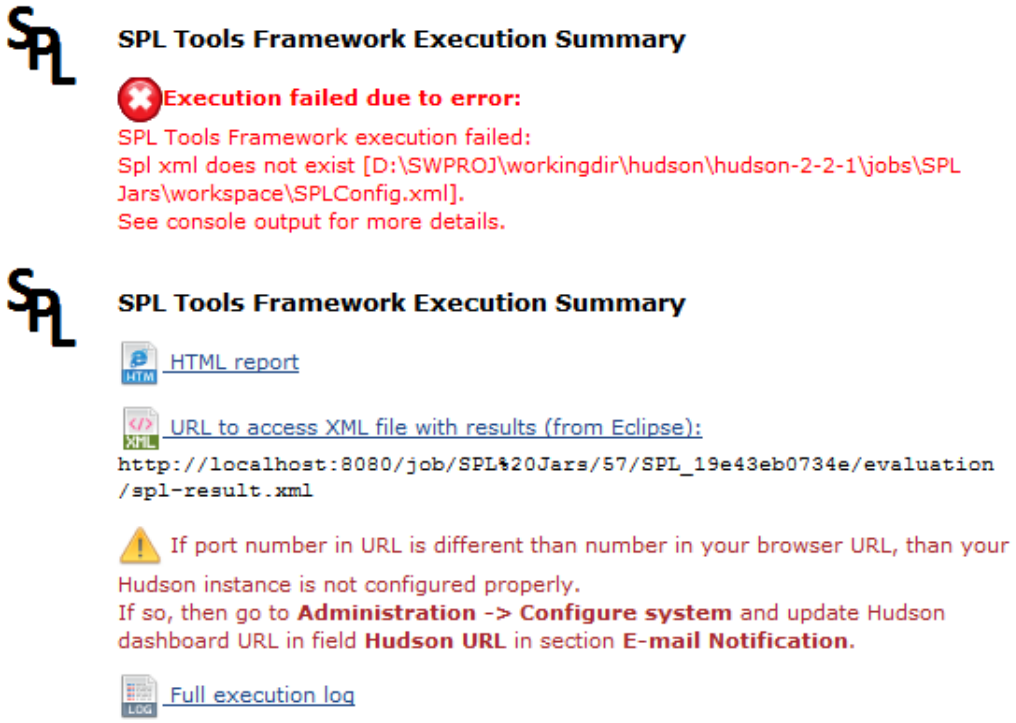


Figure 9.1: Hudson – build results

The first result with SPL logo icon shows summary for failed *Framework* execution. In this case the cause of error was missing XML configuration file. There are no more results available in in this case as *Framework* did not finish successfully.

The second result shows summary for successful *Framework* invocation. You can see links for HTML report, XML file for Eclipse and full execution log. Each of those links is visible only when target files are available (i.e. link to HTML report is visible only if HTML report was generated).

If HTML report (which is enabled by default) was generated, you will see link for opening it in the build side menu as shown on following figure.

Figure 9.2: Hudson – build menu

There will be entry for HTML report for all successful SPL build step witch generated HTML report. This link allows to inspect *Framework* execution results inside Hudson interface.

When you access results from Eclipse using *SPL Tools Eclipse Plug-in*, you can view interactive graphs as long, as performed measurement files in *Plug-in* SPL working directory are available. *Plug-in* will provide them to *SPL Tools Eclipse Plug-in* over HTTP communication protocol.

Links to *Framework* execution results are also printed to Hudson job build console output.

9.4 Integration with *SPL Tools Eclipse Plug-in*

The URL for XML report file is available in build summary and printed to the build console output when it was generated by the *Framework* execution.

Note that actual URL can be provided in two different forms depending on running Hudson instance URL configuration (see [Hudson URL configuration on page 78]).

Provided URL for viewing results in Eclipse can be either absolute (i.e. it contains protocol, server, port and path to Hudson instance dashboard from Hudson configuration), or relative to Hudson dashboard (when configuration is not set at all).

9.5 Example configuration for testing repositories

This section provides simple tutorial for configuring example Hudson job for usage with **SPL Tools Hudson Plug-in**.

9.5.1 Basic job configuration

Follow those steps to create basic job:

1. Go to Hudson dashboard and select ***New Job***
2. Enter name **SPL Example**, select ***Build a free-style software project*** and confirm with ***OK*** button.
3. In section ***Source Code Management*** select ***Subversion***.
4. Enter following URL to field ***Repository URL***:

   `svn://svn.code.sf.net/p/spl-tools/testingrepository3/`

Figure 9.3: Hudson - Tutorial - Repository URL

5. Now go to section ***Build*** and add build step **SPL Tools Hudson Plug-in**.

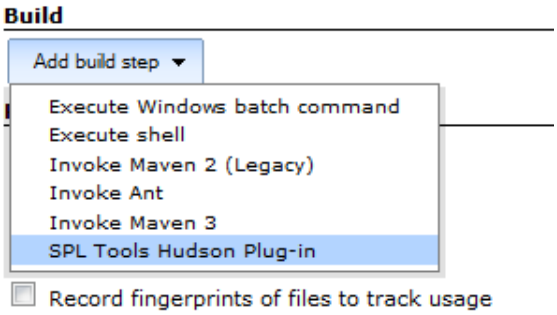


Figure 9.4: Hudson tutorial - add build step

6. Default build step configuration will be added which is sufficient for this example - XML configuration file is named ***spl-config.xml***, working directory can be ***.spl-tools*** in Hudson job workspace root, no additional INI configuration file is used now and we will run execution locally (those values will be displayed when you press ***Advanced...*** button).

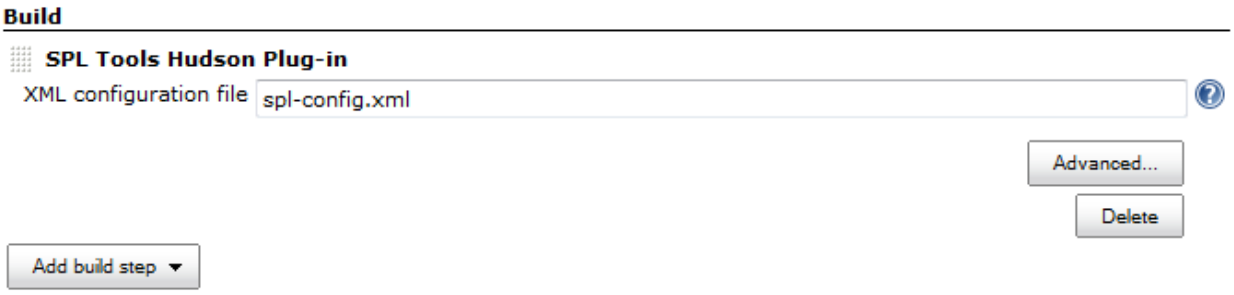


Figure 9.5: Hudson tutorial – default SPL build step

7. Save new Hudson job with ***Save*** button on bottom of the page.

9.5.2 Basic job execution

To run configured Hudson job, issue ***Build Now*** action in job menu.

You can observer progress in ***Console output*** page of started build, where you will see SVN checkout progress and then *Plug-in* execution.

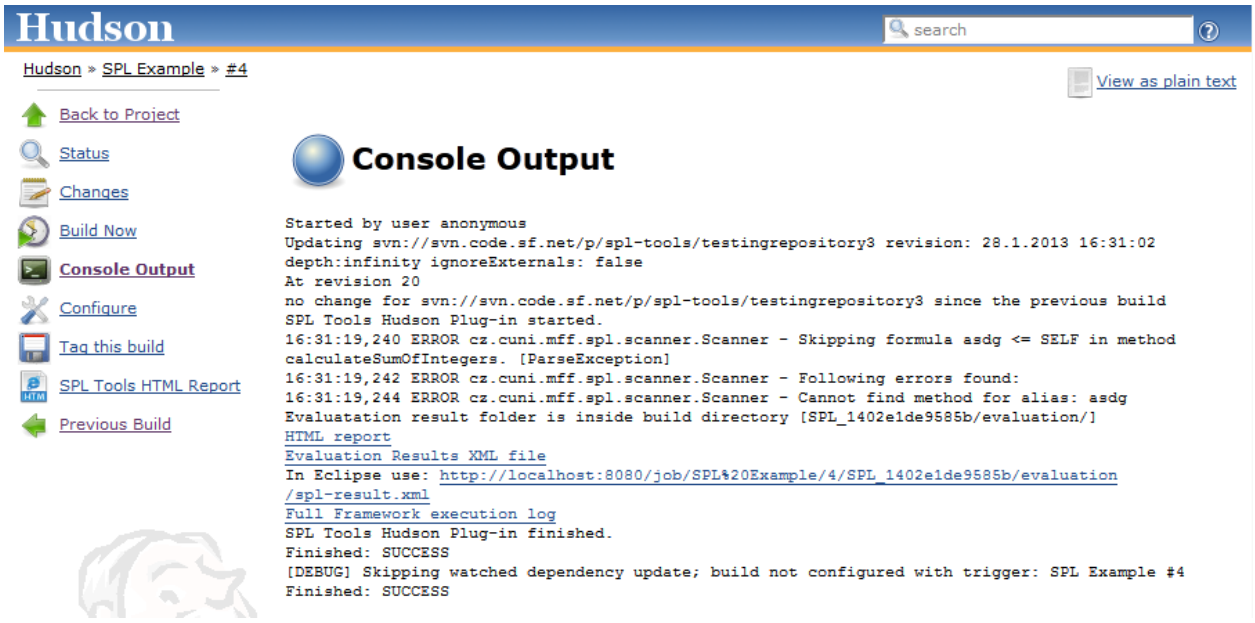


Figure 9.6: Hudson tutorial – build console output

As you can see in figure above, *Framework* execution found some errors in one formula. Those errors are there for demonstrating error reporting. All errors or warnings in annotations are reported to be quickly reviewed during execution.

You can cancel running *Plug-in* execution with red cancel button in Hudson UI same way as for any other Hudson task.

9.5.3 Results

You can view summary on build status page and inspect evaluation HTML report using link ***SPL Tools HTML Report*** in build side menu.

The HTML report is interactive so you can navigate to more detailed results from the summary page.

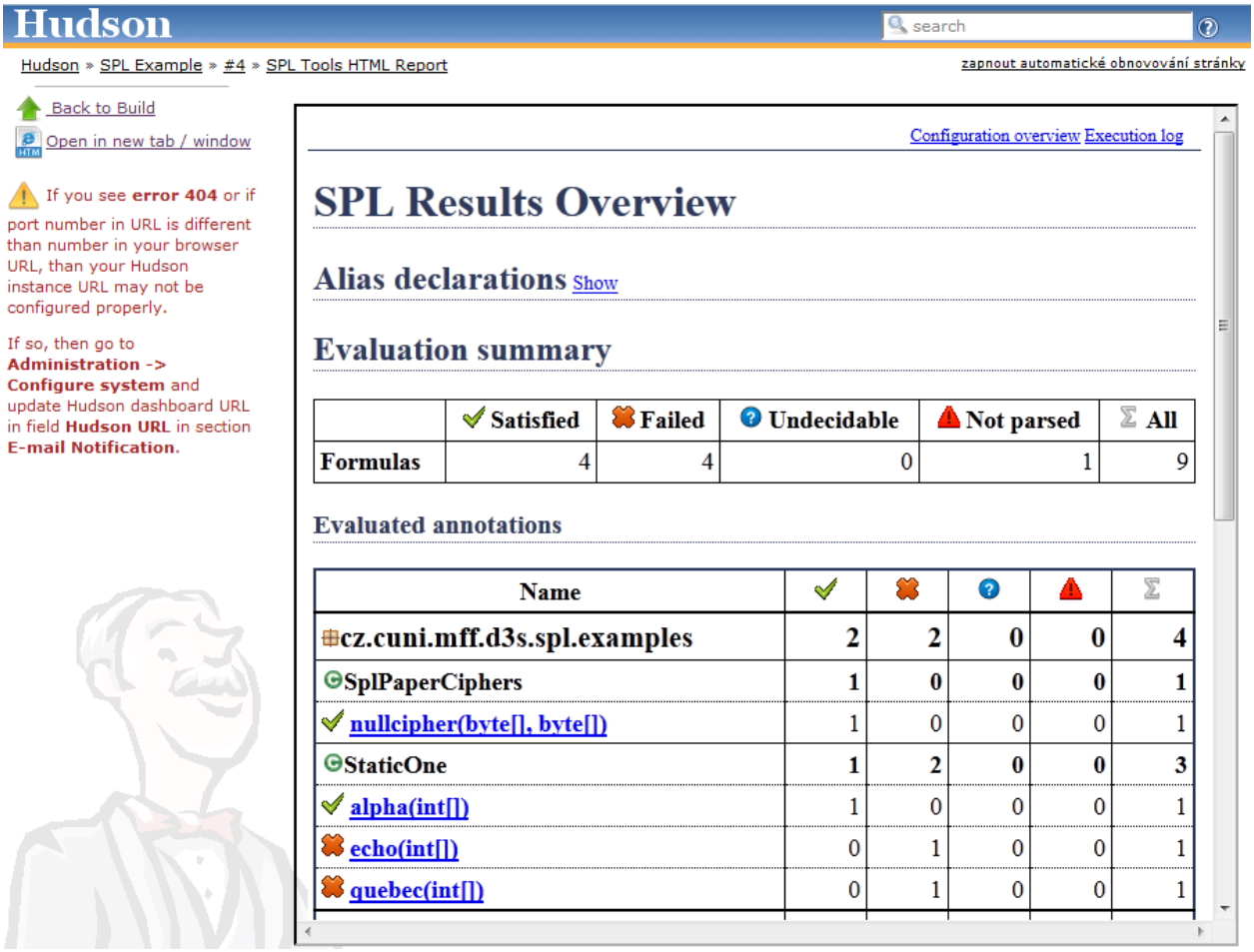


Figure 9.7: Hudson tutorial – HTML report

HTML report contains two overview summary tables. The first one with overview summary of the entire execution and the second more detailed one with summaries for packages, classes and finally methods where SPL annotations with formulas are placed.

9.5.4 Using INI configuration file

You can use INI configuration file[6] to affect *Framework* behaviour.

The repository of this tutorial example contains simple INI file **spl-brief.ini** with three configuration options:

- set number warm-up samples to 10
- set number of measured samples to 20
- do not generate graph images as this takes a lot of time

Follow these steps to apply this configuration file:

1. Go to Job configuration.
2. Write value **spl-brief.ini** to field ***INI configuration file*** in **SPL Tools Hudson Plug-in** build step configuration.

[6]See [INI configuration file on page 30]

3. Save configuration changes.

Plug-in will pass this INI configuration file to the *Framework* for all newly started executions. Note that the warm-up sample count and measured sample count configuration overrides will not have effect in our example as we have already measured all necessary samples and they will be used for new measurements only. The only override which will take effect now will be the one for not generating graph images.

10. Known Issues

10.1 Git conflicts

During work on the case study a difficulty with the library that's used to access Git repositories has showed up. Check outs to seldom revisions of the JDOM library fail due to unknown conflicts on cloned and clean repository. However, these conflicts never show up when using command line Git from main stream. To solve this problem fall back to Git on system path has been added and when a conflict occurs *Framework* seamlessly switches to the Git on system path.

10.2 OutOfMemoryError: PermGen space error

This error is caused, because the running Java Virtual Machine (JVM) did not have enough memory allocated for permanent generation objects (classes, static variables etc.).

Solution is to increase JVM memory allocated for permanent generation objects. This can be done through adding JVM parameter `-XX:MaxPermSize=256M` to the Java executable. This argument sets maximum permanent generation memory size to 256 MB (which should be enough for Eclipse or Hudson plug-in, if not then increase the number).

The example command to run the *Framework* from the command line:

```
java -XX:MaxPermSize=256M -jar SPL.jar
```

List of Figures