Stochastic Performance Logic Development Documentation

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.

This document contains development documentation describing implementation details of the *SPL Tools Framework*. If you are looking for usage instructions, than refer to the **Stochastic Performance Logic User Manual** document[2].

License agreements

[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] Stochastic Performance Logic User Manual download page `http://sourceforge.net/projects/spl-tools/files/documentation/`

1. Main project overview

SPL Tools project consists of a command line utility, an Eclipse plugin and a Hudson plugin. These tools are stored in separate repositories. The basic functionality is placed in the main "code" repository and is used by both Eclipse and Hudson plug-in as a library.

The code repository is located on following URL:

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

To compile and run the code **Java Development Kit 1.7** (JDK7)[1][2] and **Ant 1.8**[3] or higher is required. The code is built using ant targets. The most important targets are the following:

```
ant              - builds the code
ant dist         - builds packed jar distribution
ant zip-dist     - builds and packs jar and dependencies to zip file
ant doc          - builds JavaDoc documentation
ant test-junit   - builds and runs unit tests
ant case-study   - downloads actual case study files and runs it
```

The structure of code repository follows classical conventions and is shown in following diagram.

```
(d) .git              (GIT folder)
(d) lib               (folder for SPL library JAR files)
(d) src               (various source files)
(d)  |- examples      (XML and INI examples)
(d)  |- java          (main Java source files)
(d)  |- script        (shell wrappers)
(d)  |- test          (test files)
(d)    |- junit       (unit tests)
(d)    |- projects    (integration tests, performed by dynamic junit tests)
(d)  |- uml           (documentation diagrams)
(d)  |- xslt          (xml to html evaluator transformation files)
(d) tools             (various tools)
(-) .classpath        (Project Java class path configuration file)
(-) .gitignore        (GIT ignore file)
(-) .project          (Eclipse file with project configuration)
(-) LICENSE.txt       (SPL license file)
(-) SPLcodetemplates  (Eclipse file)
(-) SPLformater       (Eclipse file with formatting options)
(-) SPLsaveactions    (Eclipse file)
```

To know more details about development of Eclipse or Hudson plugin see [Eclipse Plug-in on page 40] and [Hudson Plug-in on page 58].

[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/`

2. Annotations

This chapter describes structure of SPL annotation and the process how to obtain informations necessary for creating measurement code and running measurements. Also the structure used for representing this informations is described here.

2.1 Structure

Annotations are used to describe what methods are measured, what is the input for them and what performance relations should hold between them.

The SPL annotation declaration specifying the syntax follows

```
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;

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

It has three parts:

methods
: It defines which methods are measured. That includes the project and revision specification.

generators
: It defines what objects are used to create input for measured methods. These objects can be from different projects and revisions than the measured method.

formula
: It defines performance relations between methods with specific generators that create method input. There can be used methods and generators defined in the other two parts of annotations or they can be defined right here.

The grammar used for writing annotations is described in detail in user part of the documentation in the chapter **Annotations**.

2.2 Parsing

Parsing single part of annotation is made by parser created by tool **JavaCC**[1] . The parser is generated from source **Parser.jj** in package **cz.cuni.mff.spl.formula.parser** to the same package before project compilation. The main generated class for using parser is **cz.cuni-.mff.spl.formula.parser.Parser**.

Objects from package **cz.cuni.mff.spl.annotation** are used for returning value of formula specification and declaration of generator or method.

The parser requires the string to be passed as first parameter and an instance of class **cz-.cuni.mff.spl.formula.context.ParserContext** as second parameter. The context should be a clone of the context initialized from the project configuration. The parser uses it for searching projects, revisions and machine. Parser itself modifies the context only by storing problems occurred during parsing into it. The parser context is modified during formula expansion by the by **Expander** when all generators and methods in the formula are stored into the context.

Parser usage is very simple. Generator or method can be parsed with static following methods according to the expected type of definition:

with alias

This means that the definition is in the form *alias* = *definition* and then those methods are used:

- **parseGeneratorAlias(String, ParserContext)**
- **parseMethodAlias(String, ParserContext)**

without alias

The definition contains only the part *definition* without the *alias* = prefix. This parsing feature used in Eclipse plug-in editors where alias definition parts are separated. Those methods are then used:

- **parseGeneratorDeclaration(String, ParserContext)**
- **parseMethodDeclaration(String, ParserContext)**

The generator and method parsing runs only parser itself and can produce errors (not found project or revision) placed to the parser context. Generator parsing can produce warnings (generator declared as class in default package). Method parsing can not produce any warning.

Whole formula is parsed by the static method:

- **parseAndExpandFormula(String, ParserContext)**

This method runs the parser first. Then the expansion for resolving variable names to values and generating objects with concrete values is called on the structure created by the parser in previous step. The expansion can produce errors (duplicate variable, not declared variable, not found project or revision, not found alias, wrong number of sequence variables and their values) and warnings (not used variable, conversion of real parameter to integer value), all of them all placed into the parser context.

[1]JavaCC web page `http://javacc.java.net/`

2.3 Formula expansion

The formula contains only variable names after it is parsed and the initial formula structure is obtained, it contains no specific integer values yet. The class **cz.cuni.mff.spl.formula-.expander.Expander** is used to process the variable names to concrete values. Expansion is not used for parsing generator and method definitions because they contain no variables.

The **Expander** combines every value of every variable step by step and for every combination of values traverse through the objects created by parser. It creates new formula objects with these concrete values and joins them by conjunction.

To show it on example the **Expander** gets formula defined as:

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

And transforms it to the expanded form:

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

During expansion new objects are stored in the parser context in a set. They are compared for equality and if there is already equal object it is used instead of the new one. That cause that measurements represented by equal objects are measured only once. Even two measurements written in a different way in the formula but with the same meaning will be recognized as equal and measured only once for whole formula.

The **Expander** class contains only the main entry point for formula expansion. It prepares the mapping of variable names to their values. Each object present in the formula is responsible for creating their properly expanded copy.

The expanded formula instance contains no more references to variable names (instances of the class **cz.cuni.mff.spl.annotation.ParserVariable**), it contains only variable values (instances of the class **cz.cuni.mff.spl.annotation.ExpandedVariable**). Only expanded formulas are processed further in the *Framework*.

2.4 Info Object

Object **cz.cuni.mff.spl.annotation.Info** is used for storing information gathered by SPL scanner from project configuration and annotations of methods. It uses the other classes from the package **cz.cuni.mff.spl.annotation** to represent every information necessary for the measurement. The **Info** instances are used for exchanging data between different parts of the framework and to preserve persistent configuration in the XML configuration file.

On the beginning of the SPL execution process the first **Info** class instance is deserialized from the XML configuration file by the **Castor**[2] library for XML serialization and deserialization. The structure of the XML configuration file is described in detail in user part of the documentation in the chapter **XML configuration file**.

[2]Castor library web page `http://castor.codehaus.org/xml-framework.html`

First it is necessary to know how to obtain the project with SPL annotations. The **Info** created from the XML configuration is used during the creation of the binary for collecting performance data to determine what needs to be created.

When the project code is built it is needed to find annotations. Scan patterns in the configuration are used by scanner to specify which classes or packages are scanned for annotations.

Parser is then called to parse found annotations. The parser context is initialized from the **Info** object with projects and their revisions and the machine where measurement will run. After parsing is done the created objects representing measurements with methods and generators are stored again into the **Info** object.

The result **Info** contains data needed for checking out all necessary projects and their revisions to generate measurement sampling binaries.

After measurement is done the informations in **Info** object are used by the formula evaluation process to for statistical evaluation and creation of the measurement output fragments.

2.5 Scanning

This section describes scanning process that reads annotations stored in the annotated project's code. That's an essential thing, as SPL annotations contain formulas that describe performance relations between various methods. Knowing that, *Framework* measures methods referenced and evaluates formulas declared.

From the technical point of view, all information read and parsed from annotations is stored inside **Info** object which was described in the previous chapter. This object combines configuration declared in project's configuration XML file with information acquired from SPL annotations in the annotated project. Scanner's task is to walk trough all classes on paths specified in the XML configuration file for the annotated project and pass all found SPL annotations to the parser. See [Parsing on page 4] for more information about annotation parsing.

The scanning process is backed by the **Scanner** class from **cz.cuni.mff.spl.scanner**.

2.5.1 Patterns

The Scanner expects, that all classes to be scanned are prepared on the class path either as class files in a directory tree or in a jar file.

However scanner can be configured to scan only certain classes or packages. To accomplish that scan pattern may be specified in the XML configuration file. By default all classes found on class path are scanned for annotations.

Following scan patterns are supported:

"*" Scan everything found.

"package.*"
Scan all classes in the package but not the classes in the sub packages.

“package.”**
: Scan all classes in package the package and also all classes in all sub packages.

“package.Class”
: Scan just the class specified.

The scanning process less time and consumes less memory when the scan patterns are specified. The ideal situation occurs when the only scan patterns point to the classes with the annotations.

Class listing

At first URLs that the class loader has specified are extracted and all jar files and directories that these class paths point to are listed.

Then, all classes that can be found in these jar files and directories are compared to the scan patterns specified in the project configuration and are listed in case of match.

This list of classes is prepared to be scanned for annotations.

It’s necessary to walk trough all classes on the class path and parse these classes by hand because Java has no standard API which would provide list of all classes located on the classpath.

Class scanning

The initialized URL class loader is used again to scan for annotations. All listed classes are loaded using this class loader and scanned for SPL annotations.

If a SPL annotation is found inside the scanned class, than the annotation is processed by the method **processAnnotation** which uses the annotation parser to acquire all definitions and the result is saved into the **Info** object instance.

3. Sampler Creation

This chapter will describe the creation of a binary which is composed of the method to be measured, the generator providing arguments for this method and our measuring code that triggers and controls the sampling process. The purpose of such binary is to collect performance data of the selected method. From now on this binary will be called as a **sampler**.

All classes in this chapter are implicitly from the **cz.cuni.mff.spl.deploy.build** package. This whole process is backed by the **Build** class.

3.1 Getting complete Info

The first step is to determine what needs to be created. To accomplish this the XML configuration file is loaded. This file, apart from other things, contains instructions how and where to obtain the code of the main revision (which is referenced as **HEAD** revision) of the main project (which is referenced as **THIS** project). The XML file is loaded into instance of class **cz.cuni.mff.spl.annotation.Info**. See [Info Object on page 5] for more details.

Based on the repository type specified the corresponding implementation to access such type of repository is used and the **HEAD**'s code is checked out into a local directory. See [Repository Access on page 15] for more details on repository types and [File system organization on page 14] for more details on file system organization.

The code is obtained but it's not yet sure if there are source files or already built binaries. If **THIS** project has **build** command specified it's supposed the source files have been checked out and the command is executed to produce binaries. Otherwise binaries are expected. See [Execution on page 18] for more details on build command execution.

Then, according to the **class paths** which are in fact Java class paths specified relatively to the code's root directory and according to the **scan patterns** the scanner is initialized. The scanner enhances the **Info** object with information from the **HEAD**'s binaries.

This is performed by **BuilderScanner** class.

3.2 Knowing what to create

When the complete **Info** has been obtained there's a list of all formulas, comparisons and measurements that should be processed and evaluated. The statistical evaluation requires that every measurement is performed and measured data are available.

Measurement of a single method references two revisions. The revision of the method to be measured and the revision of the generator that will provide input arguments to the measured method. These revisions may be identical or they may be different but come from the same project (repository) or they may be totally different and origin in even different projects (repositories).

Measurement can be uniquely identified based on the functions' revisions it's composed of. This unique identification is utilized in the data store[1]. To determine which measurements must be performed and which can be loaded the store is checked for measurement presence.

To get the complete list of already performed measurements the identification is created from measurement properties and is checked against the store. However, there's a difficulty in case the revision's identification in not absolute[2] and such revision must be at first resolved which may take some additional time as the repository must be accessed.

The measurements that are not found are listed and samplers will be created for them. All revisions referenced by these measurements are prepared in the same way the **HEAD** revision has been.

This is performed by **BuilderContext** class.

3.3 Sampler details

Now that the dependencies for creation of all samplers have been prepared the retrieval of a single sampler's details for a single measurement will be described.

This is performed by **Assembler** class.

3.3.1 Reflection

To get more information two class loaders (one for the generator, one for the measured method) are instantiated with the corresponding class paths to their dependencies. Classes implementing the generator and the method will be loaded using their respective class loaders and reflection will be used to get type information about these classes.

Reflection is used only to retrieve information about classes. It's not used during measuring. The necessary information acquired through reflection is used to determine if the referenced generator is a static class and which particular method of the referenced SPL method should be used for generator arguments as the code generation has to prepare proper type conversions for the arguments.

3.3.2 Generator

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 generator might be represented as a whole class, as a static function or as a member function. To determine this the **Info** and mentioned reflection are used. In case the class

[1]That is useful in big projects where tens of measurements are present. When such projects advances and new revisions are added only the measurements from new revisions must be performed, the measurements from old revisions may be just loaded because the measuring was already performed for them.

[2]For example "master" or "HEAD" instead of full hash or revision number

kind is used all super classes and implemented interfaces are checked for interface match on the specified generator data type. If the generator is of method kind (static or member) the return type is checked for type match.

Few more already known details are needed to use the generator. Annotations enable passing of some arguments to the generator. A single string might be passed to the generator constructor. And unlimited number of integer arguments might be passed to the generator static or member function. These arguments are parsed from annotations and are therefore present in the **Info** object.

3.3.3 Method

The measured method may be a member or a static function.

The method's arguments might be specified in the annotation but don't have to. If the arguments are not specified than the method's name can't be overloaded because it's undecidable which declaration should be used. If the arguments are specified the specified overloaded variant is picked.

If the method is a member function a single string argument may be passed to the class' constructor.

3.3.4 Example

Lets measure two sorting functions on integer arrays.

```
public class Method {
    public static void sort1(int[] a) {
        java.util.Arrays.sort(a);
    }
    public static void sort2(int[] a) {
        java.util.Arrays.sort(a);
    }
}
```

Now an integer array generator is needed. The important think to note is the type that `generate()` returns. The data it returns have in this case three dimensions.

First dimension is the number of elements in the array list. Every element suffices a single call on sort function.

Second dimension is the number of arguments the sort function takes, actually wrapped in `Object[]`. In this case it's a single argument, an integer array.

Third dimension is actually not interesting for SPL. It's the size of the array to sort.

```
public class Generator {
    Random rnd = new Random();
    int calls;
    int args;

    public Generator(String calls_args) throws Exception {
        String split[] = calls_args.split(";");
        calls = Integer.valueOf(split[0]);
        args  = Integer.valueOf(split[1]);
    }

    public ArrayList<Object[]> generate() {
        ArrayList<Object[]> allCalls = new ArrayList<>();
        for (int i = 0; i < calls; i++) {
            int[] argument = new int[args];
            for (int j = 0; j < args; j++) {
                argument[i] = rnd.nextInt();
            }
            Object[] callArguments = new Object[] {argument};
            allCalls.add(callArguments);
        }
        return allCalls;
    }
}
```

Declaration of the formula follows. Interesting part here is the way arguments are passed to the generator. Due to limits of Java annotations most arguments must be passed as strings and later on parsed like in the Generator constructor. In this case generator will create data for 30 calls and each call will sort an array of 1000 integers.

```
import cz.cuni.mff.spl.SPL;

public class Measurement {
    @SPL(
            generators = {
                    "generator=main.Generator('30;1000')#generate()"
            },
            methods = {
                    "sort1=main.Method#sort1",
                    "sort2=main.Method#sort2"
            },
            formula = {
                    "sort1[generator] < sort2[generator]"
            })
    void compareSorts() {}
}
```

3.4 Sampler Code

To generate sampler code classes templates and **Apache Velocity** template framework is used. The sampler is made out of three parts compiled separately. This separate compilation is necessary due to class path conflicts, for the generator and the measured method may origin in different revisions and have colliding dependencies. Therefore they might not be called directly from the main part of the sampler but are called via generic interfaces.

This is performed by **Code** class.

This part is very tricky to implement due to some issues.

Code generation
: There's little support in Java for code generation on the level needed here. Essential in our project is to have easily editable templates where values like class names, functions and arguments are filled at run-time. Therefore **Apache Velocity** template library has been picked.

Glue code
: Another interesting thing is to create code that can combine functions from various projects and revisions and keep return value's and arguments' types right. This would not be that tough if reflection was used but due to performance issues it's not.

  To manage this a system of interfaces and class loaders is used. The generator and method are essentially hidden behind generic interfaces. Implementations of these interfaces are created and compiled against generator and method projects respectively. That way one can work with the interface and use URL class loaders to provide the correct implementation that wraps the original project's code. When implementations are loaded class loaders are linked to the class loader hierarchy.

  Lesson learned from our case study is that context class loader must be set prior to every call to generator class and method class because even the called code may use class loader tricks.

3.4.1 Class structure

The structure of sampler is following:

IGenerator
: This is the generic interface to access the generator. To use the generator specified there must be an implementation of this interface.

```
Iterable<Object[]> newInstance() throws Throwable;
```

CGenerator
: This is measurement specific implementation *IGenerator*. The overridden method instantiates and returns generated data based on the generator's type, kind and all arguments specified.

IMethod
: This is the generic method interface that has a function to instantiate the object to call the measured method on and a function to call the measured method on the instantiated object.

```
void newInstance() throws Throwable;
void call(Object[] arguments) throws Throwable;
```

CMethod
: This is measurement specific implementation *IMethod*. The `newInstance()` initializes the object if the kind of method is a member one or at least makes virtual machine to load the class if the kind is static. The `call(Object[] arguments)` casts the array of arguments to match the signature and calls it.

Measurement

This is the main class of the sampler. Its task is to load the generator and measured method using explicit class loaders. The measurement is performed only via the *IGenerator* and *IMethod* interfaces. The implementation is loaded from specified folders where the actual implementation is located. When the classes are loaded next task of this class is to perform warm-up and measuring based on parameters specified and store the measured data.

3.4.2 Dependency structure

Apart from the generated classes there's a structure of dependencies. These dependencies are placed in directories named corresponding to their purpose starting with "generatorCP" and "methodCP" and numbered incrementally to avoid collisions. However, their creation and usage is the same for both.

These dependency directories and their content is based solely on **class paths** specified in the configuration XML and files located on these path inside the revision's code. There might be a class folder or a jar file on the class path. In both cases a directory is created and the content of the folder or the jar file is copied inside the directory.

3.4.3 Compilation

Compilation is handled with Java built in compiler. Java Development Kid is required. The generator part, the method part and the main sampling part are compiled separately.

Compilation functions are in **BuildUtils** class.

3.4.4 Archive

For easier distribution and transfer all generated classes and dependencies in the sampler directory are packed into a zip archive.

3.5 Command execution

The project's build command is expected to be simple and cross platform. Java execution mechanism executes only bare binaries on the system's or user's path. But build command is usually a shell script therefore it's necessary to invoke it inside the platform's specific shell.

Windows
: The command is invoked in the Command Prompt.

```
cmd /c command
```

Linux
: The command is invoked in the Bash.

```
/bin/bash -c command
```

Other
: On other platform standard shell is expected to be present.

```
/bin/sh -c command
```

Configuration
: If any of these settings does not satisfy project's needs it's possible to switch off guessing of the platform's shell. In this case the specified build command must itself invoke shell binary and the command inside it.

```
command
```

3.6 File system organization

The whole build process resides in a single output directory. There's stored an **Info** object prior to the build process and post the build process. There is also separate directory for revisions' source code and generated sampling code.

In the source code folder there's a directory for each revision's code. There are also cache directories for cloned repositories so they can be cloned once and afterwards accessed locally.

In the generated code folder there's a directory for each created sampler.

4. Repository Access

This framework supports access to code from multiple kind of sources. There's support for **Git** repositories, **Subversion** repositories and local directories.

Implementation for access to code stored in repositories is in **cz.cuni.mff.spl.deploy.-build.vcs** package.

4.1 Interface

The generic interface **IRepository** has a single method:

```
String checkout(String what, File where) throws VcsCheckoutException;
```

The semantic of this function is to check out specified revision "what" to the "where" directory and return permanent identification of the checked out revision.

Some version control systems allow to reference revisions with non-permanent identifications (e.g. "master", "remote/mybranch", "HEAD", etc.) and these aliases are not suitable for long term storage of measurements because their semantic may change in time. Therefore it's needed that when revisions with these aliases are checked out also the permanent identification is resolved so the measurement can be stored.

The format of argument "what" is specific for every class implementing the interface.

4.2 Factory

To call the mentioned function to check out code the **IRepository** object must be at first obtained. There is a class **RepositoryFactory.java** that has method for obtaining such correct object:

```
IRepository parse(
        String type, String url, Map<String, String> values,
        File xml, File cache, InteractiveInterface interactive)
            throws VcsParseException
```

Description of arguments:

type
: This argument is a type of the repository. There are four possible values {*git*, *subversion*, *source*, *sourceRelative*}. These values come from **RepositoryFactory.RepositoryType** enumeration and are parsed to this enumeration. Based on this value corresponding implementation of **IRepository** is picked.

url
: This argument specifies location of the repository. In case the repository type is *git* the URL is a location of the repository. Same goes for *subversion.* In case the repository type is *source* the url is expected to be an absolute path to the code directory. In case the repository type is *sourceRelative* the url is expected to be a path to the code directory relative to XML configuration file.

values
: These are optional configuration values loaded from INI configuration. Semantic of these values is specific to each repository implementation. In case of *git* and *subversion* it may contain further credentials to access private repositories where log in is required.

xml XML configuration file location.

cache
: This is temporary cache directory that repository implementation may use to store temporary data. *Git* for example at first clones remote directory there so further work is performed locally only.

interactive
: This is optional argument. If some interaction is needed this is the interface to perform it trough.

4.3 Git

Git implementation is based on **JGit** [1] library from **Eclipse** project. JGit is a pure Java implementation of Git. It supports access to public or private repositories using no authentication at all, interactive password authentication, key authentication or interactive pass-phrase key authentication.

Unfortunately in some repositories and on some revisions there has been problem using the **JGit** library. It seems that some chain of commands which is perfectly valid and correct in original **Git** implementation is not suitable for **JGit**. This is solved with fall-back to system git binary if such is present. This fall-back is seamless. However only public repositories are supported in this fall-back mode.

The **IRepository** is in **Git** class and the fall-back implementation in **GitSystem** class.

4.4 Subversion

Subversion implementation is based on **SVNKit** [2] library. This is also pure Java library. The implementation supports access to public or private repositories using no authentication at all, interactive password authentication, key authentication or interactive pass-phrase key authentication.

In this case there has been no need for any fall-back mechanism.

4.5 Extension

To extend the framework with another version system few steps must be accomplished.

- Implement another **IRepository** class.

[1]JGit library web page `http://www.eclipse.org/jgit/`
[2]SVNKit library web page `http://svnkit.com/`

- Add its type to **RepositoryFactory.RepositoryType** and add condition to the method `parse(...)` of **RepositoryFactory** so the new type will be parsed to the new class.
- If further information such as login credentials are needed add section to **cz.cuni.mff-.spl.configuration.SplAccessConfiguration**.

5. Execution

This chapter describes the process of execution prepared samplers and collecting data. Execution is handled by a forked server process. Framework controls the execution via client classes that communicate with the server.

cz.cuni.mff.spl.deploy.execution.server
: This package contains server classes that are packed (together with few other dependencies) and forked on the target machine of execution.

cz.cuni.mff.spl.deploy.execution.run
: This package contains client interface for deploying and controlling the server.

5.1 Overview

There are some good reasons to control execution by a new process.

- It's easier to implement execution on remote machines because samplers are always controlled by local process. That means easier logging, troubleshooting and killing in case of timeout.
- More code can be shared in local and remote execution implementation.
- Remote execution may be started in such way that two server will never run at the same time. Otherwise the precision of measured data could be harmed.
- In case of remote execution if a connection is lost the server can still go on and results can be retrieved after the connection has been established again.

Server executes samplers in batches. That is sufficient for this purpose because all samplers are created at once and only once per run. It also makes the process simpler.

5.2 Binary

Binary of the server is not prepared by a build system. It's packed into a jar file at run-time. Standard classes are used for that. This sort of packing makes testing and running simpler and more flexible because it's not tied to any sort of build system.

5.3 Initialization

Client interface follows few steps to make the server running.

client
: At first a target machine (local or remote) and a path where to start the server must be known.

client
: Unique identification is generated for the server and server's binary is named using the identification and is copied into the destination path. The unique server name is used

to avoid collisions if two servers are being started concurrently in the same location. When the binary is successfully copied it's started.

server
: The very first thing server does is it tries to acquire a file lock on well known file to make sure there's no other server running. If lock is successfully acquired success status is returned via standard output to the client and output and error streams to client are closed. Otherwise failure status is returned and server is shut down.

client
: If failure status has been reported client tries to restart the server process in regular intervals until success status is received.

client
: After server has been successfully started samplers and configuration is copied to the server's directory named using the unique identification. Configuration file is described below. When all samplers and configuration is successfully copied a start file is created.

server
: Server waits until detects start file. With file detected the configuration is read execution started on samplers one by one.

Configuration file contains following values.

Job identification
: This number identifies the job and determines execution order. Server uses this number as a name for the job's directory.

Archive name
: Name of the ZIP archive that contains sampling code. Content is extracted to the job's directory.

Command
: Command that's used to execute the sampling code.

Timeout
: Time in seconds how long execute may take until it's killed and marked unsuccessful.

These values are Base64 encoded and (different) entries are separated by a new line .

5.4 Structure

Following diagram describes the file system structure that execution server uses.

```
(-) .spl-server-lock  (lock all servers compete for)
(-) #.jar             (server binary)
(d) #
(-)  |- data          (configuration of samplers)
(-)  |- out           (std out of server)
(-)  |- err           (std err of server)
(-)  |- start         (for server - samplers ready, start measuring)
(-)  |- stop          (for server - shut down)
(-)  |- finished      (for client - server shut down)
     |
(-)  |- @.zip         (sampler packed in archive)
     |
(d)  |- @             (sampler unpacked according to configuration)
         |- (...)     (sampler files and directories)
         |
(-)      |- result    (samples measured)
(-)      |- out       (std out of sampler)
(-)      |- err       (std err of sampler)
(-)      |- log       (troubles starting sampler)
(-)      |- start     (for client - sampling started)         |
(-)      |- success   (for client - successfully finished)
(-)      |- timeout   (for client - killed, timeout exceeded)
(-)      |- error     (for client - error occurred)

# - unique id of server, multiple different may be present
@ - id of sampler, multiple different may be present
```

5.5 Configuration

Configuration of samplers contains following values for each sampler:

Identification
: Must be unique and be a valid directory name.

Archive name
: Must reference sampler archive file.

Command
: Command to execute sampler with.

Timeout
: Timeout in seconds when to kill the sampler.

These values are listed line by line. Every four lines values for another sampler start. All values are Base64 encoded.

5.6 Sampling

When server has started measuring client retrieves information about status of samplers via checking existence of files marking sampler progress or result and checking files marking server status shown in the structure description.

The only further interaction possible is creation of stop file which makes server to stop measuring earlier and shut down. This is triggered in case of interruption on the thread controlling execution.

When all samplers are measured the server lock is released and server exits.

5.7 Secure shell

For remote execution file access and command execution is processed via secure shell. Implementation is based on **JSch** [1] library.

[1]JSch library web page `http://http://www.jcraft.com/jsch/`

6. Store

Data storage of the framework follows a strict structure. The organization is managed by classes placed in **cz.cuni.mff.spl.deploy.store** package.

Store is implemented for concurrent access. Directories that may suffer from race conditions are protected via lock files named **".spl-lock"**. These files are acquired for single operations only.

6.1 Measured data

This section contains details regarding persistence of the measured sample data. The sample data are stored in the folder named **measurement**.

6.1.1 Identification

Every measurement sample must be uniquely identified.

This identification is composed of:

- Method identification that is internally composed of:
  - Method canonical name
  - Method constructor initialization arguments
  - Method revision
    - Project alias.
    - Revision permanent identification
- Generator identification that is internally composed of:
  - Generator class canonical name
  - Generator constructor initialization arguments
  - Generator parameters
  - Generator revision
    - Project alias.
    - Revision permanent identification

This identification is constructed using **cz.cuni.mff.spl.deploy.build.SampleIdentification** class.

6.1.2 Sample file format

Measured data created by samplers are stored in human readable plain text format.

They follow this format:

- First line of the file is a serialized identification of a sampler. The identification is preceded with "#" character to signalize it's a commentary not a sampled time value.
- Next lines are properties set by the sampling code. These lines start also with "#" then continue with a name of the property followed with "=" and end with the value of the property.
- After the property section there's is a special line "#begin" that marks start of the data section.
- Data section contains measured values of the method in nanoseconds. There's a single value per line. This section ends with special line "#end" and that is also the end of the data file.

Example of sample file follows.

```
#...identification...
#date=Mar 6, 2013 4:06:44 AM
#warmup=359
#count=555
#begin
43667794
52970600
42707108
36247536
...
#end
```

6.1.3 Storage

In a store all measurements are placed into a single directory usually located in the root of the store directory and named **measurement**.

This directory is organized into sort of hash table. Sample file content is stored as a file named with hash of it's identification followed by number of already existing files with the same name. This solves hash collisions.

Prior to saving new measurement there is a check if the measurement is not already present. All files with colliding hash must be opened and their first line must be read and compared.

Store is not designed to support removal of individual sample files. In case a file is removed only if its name is colliding with other troubles may arise. These files would look like "id.0.dat", "id.1.dat", "id.2.dat" and so on. If a file removed is somewhere inside this chain all following files will be as if invisible.

6.1.4 Lock

The directory containing measurements is locked if a file is being saved, loaded or its existence is being checked.

6.2 Evaluation

6.2.1 Storage

Evaluation is composed of multiple files therefore there's always created an independent directory for each evaluation. These directories are placed into the directory that is located in the root of the store directory and named **evaluation**. Single evaluation directory is named **run-evaluate-#** where "#" is number of previously created evaluation directories.

6.2.2 Evaluation directory format

Layout of the evaluation directory is unknown to the store and is created by evaluation process.

6.2.3 Lock

The directory containing evaluation directories is locked if a new evaluation directory is being created.

6.3 Temporary

There is also need for temporary files as for example noted in the [Sampler Creation on page 8] chapter. Temporary directories are lock protected, for otherwise other instances of the framework could purge them if they were not locked.

Every temporary file created is placed into its special directory so it can be locked too.

6.4 Remote access

To enable access to measured or evaluated results on remote machines there is support for HTTP protocol. This requires that local store implementation index directories so their content can be listed remotely. This must be done due lack of remote directory listing support of HTTP protocol.

Directory indexes are implemented as special files of well known name **".spl-index"** that contains Base64 encoded file names of these directories.

For faster access measurement directory contains slightly improved index where not only encoded file names are listed but also full identifications.

These indexes are checked and recomputed every time framework initializes store directory.

Example of simple "spl-index" file with only encoded file names listed.

```
bS00MWEyOWYyOC4wLmh0bWw= ~
bS1kMWRhNzQ2Zi4wLmh0bWw= ~
...
```

Example of cache “spl-index” file with encoded sample identifications and file names listed.

```
id1 m-02ac6b50.0.dat ~
id2 m-06112751.0.dat ~
...
```

7. Annotation Evaluation

This chapter describes implementation of annotation evaluation part of the *Framework* with the main focus on evaluation of SPL formulas.

The implementation is located in package **cz.cuni.mff.spl.evaluator** and its subpackages.

7.1 Main entry class and usage

Evaluation entry class is **cz.cuni.mff.spl.evaluator.Evaluator** and its method **evaluate** with following signature.

```
public static void evaluate(
  ConfigurationBundle
      configuration,
  Info
      evaluationContext,
  MeasurementSampleProvider
      measurementSampleProvider,
  IStore.IStoreDirectory
      outputStoreDirectory,
  File
      temporaryDirectory
);
```

Valid call to this method has to specify specify all arguments:

ConfigurationBundle configuration
: The configuration which is to be used for the evaluation. Note that evaluation does not expect, that configuration details about access to remote machines are part of configuration as the evaluation works locally.

Info evaluationContext
: The instance of Info class with formulas to evaluate.

MeasurementSampleProvider measurementSampleProvider
: The instance of class implementing this interface. It provides mapping of instances of class **cz.cuni.mff.spl.annotation.Measurement** to instances of class **cz.cuni.mff.spl.evaluator.statistics.MeasurementSample**. Implementation used in the *Framework* is **cz.cuni.mff.spl.evaluator.input.CachingMeasurementSampleProvider**.

IStore.IStoreDirectory outputStoreDirectory
: The instance of writeable folder abstraction used by *Framework* which is used to save files generated during evaluation.

File temporaryDirectory
: The directory which can be used by evaluation to store temporary files.

Two structures are created from values specified in evaluator part of the configuration bundle before the evaluation process is started:

- The statistical values checker which is used for statistical evaluation and validation of measured values.

  This checker is represented by the interface **cz.cuni.mff.spl.evaluator.statistics-.StatisticValueChecker** and currently used implementation is the class **cz.cuni-.mff.spl.evaluator.statistics.StatisticValueCheckerImpl**.

- The evaluation output implementation represented by interface **cz.cuni.mff.spl.evaluator.output.EvaluatorOutput**.

  Currently used implementation is class **cz.cuni.mff.spl.evaluator.output.Evaluator-OutputAggregator** which serves as container for variable number of the **EvaluatorOutput** implementations (HTTP, XML, graphs) which are dynamically created according to values specified in evaluator part of configuration bundle.

The evaluation process starts after then and is described in the next section.

7.2 Evaluation process

The basic idea of the evaluation processing:

1. Iterate over all annotations in context and for each of them iterate over all its formulas and process them one by one.
2. As formula is is a tree composed of expressions in inner nodes and comparisons in leaves:
   (a) **Expression node** Evaluate sub nodes and then create evaluation result for node.
   (b) **Comparison node** Evaluate comparison using statistical t-test and create comparison evaluation result.
3. When evaluation is finished for one evaluated item (annotation, formula or comparison) then it is send immediately to the output.

The evaluation of formulas as described above is located in class **cz.cuni.mff.spl.evaluator .EvaluatorImpl**.

Comparison node evaluation uses statistical t-test[1], which utilizes p-value from statistical value checker to decide, whether comparison is satisfied or not.

7.2.1 Logical operation evaluation details

The expression node evaluation uses three-state logic by Kleene definition[2]. This evaluation reflects situations when some measurements have failed or don't have enough measured data (t-test needs at least two samples). The logical values representation is defined in enumeration **cz.cuni.mff.spl.evaluator.output.results.StatisticalResult** and values are named OK, FAILED and NOT_COMPUTED. Method **combine** of class **StatisticalResult** takes logical operation representation from enumeration in **cz.cuni.mff.spl.annotation.Operator**

[1]Student's t-test on Wikipedia `http://en.wikipedia.org/wiki/T-test`

[2]Three state Kleene logic `http://en.wikipedia.org/wiki/Three-valued_logic#Kleene_logic`

and two instances of **StatisticalResult** as operands and evaluates the logical operation represented by operator as defined in following truth tables (the logical operation has operator in a row as left argument and operator in a column as right argument).

row **AND** *column*	**OK**	**FAILED**	**NOT_COMPUTED**
OK	OK	FAILED	NOT_COMPUTED
FAILED	FAILED	FAILED	FAILED
NOT_COMPUTED	NOT_COMPUTED	FAILED	NOT_COMPUTED

Table 7.1: AND logical operation in Kleene three-state logic

row **OR** *column*	**OK**	**FAILED**	**NOT_COMPUTED**
OK	OK	OK	OK
FAILED	OK	FAILED	NOT_COMPUTED
NOT_COMPUTED	OK	NOT_COMPUTED	NOT_COMPUTED

Table 7.2: OR logical operation in Kleene three-state logic

row $\implies$ *column*	**OK**	**FAILED**	**NOT_COMPUTED**
OK	OK	FAILED	NOT_COMPUTED
FAILED	OK	OK	OK
NOT_COMPUTED	OK	NOT_COMPUTED	NOT_COMPUTED

Table 7.3: Implication logical operation in Kleene three-state logic

Note: NOT logical operation declaration is not described as it is not supported by SPL grammar. It can be found on referenced Wikipedia page

7.2.2 Comparison statistical evaluation details

Comparison evaluation uses the **Apache Commons Math**[3] library.

The comparison evaluation implementation is in the class **cz.cuni.mff.spl.evaluator.statistics.ComparisonEvaluator**. It uses class **org.apache.commons.math3.stat.inference.TTest** to run the t-test on provided comparison.

See the t-test method Javadoc[4] for its usage instructions.

Most important part is the one for testing whether one sample is lower than the other. It is necessary to check first if it is valid for the means and if so then following two methods are used:

[3]Apache Commons Math library `http://commons.apache.org/proper/commons-math/`

[4] double TTest.tTest(...)
`http://commons.apache.org/proper/commons-math//apidocs/org/apache/commons/math3/stat/inference/TTest.html#tTest%28org.apache.commons.math3.stat.descriptive.StatisticalSummary,%20org.apache.commons.math3.stat.descriptive.StatisticalSummary%29`
boolean TTest.tTest(...)
`http://commons.apache.org/proper/commons-math//apidocs/org/apache/commons/math3/stat/inference/TTest.html#tTest%28org.apache.commons.math3.stat.descriptive.StatisticalSummary,%20org.apache.commons.math3.stat.descriptive.StatisticalSummary,%20double%29`

double TTest.tTest(StatisticalSummary sampleStats1, StatisticalSummary sampleStats2)
The returned p-value has to divided by two according to the Javadoc documentation.

boolean TTest.tTest(StatisticalSummary sampleStats1, StatisticalSummary sampleStats2, double alpha)
The parameter **alfa** has to be equal to the limit p-value (which is obtained from statistical value checker) multiplied by two according to the Javadoc documentation.

When the left mean is not lower than the right one, than the comparison is definitely not satisfied and the p-value in the result is set to 0.

Calls to the **tTest** methods listed above in case of the equality check is standard - no multiplication or division by two.

7.3 Generating output

Evaluation output generation is integrated into evaluation process as described in the previous section. This integration allows to generate output continuously as the evaluation result parts for evaluated items (annotation, formula, comparison or measurement sample).

The evaluation output is represented bythe interface **cz.cuni.mff.spl.evaluator.output-.EvaluatorOutput**.

```
public interface EvaluatorOutput {

    void init(
        ConfigurationBundle configuration,
        Info context,
        StatisticValueChecker statisticValueChecker,
        IStoreDirectory outputStoreDirectory
    ) throws OutputNotInitializedException;

    void generateMeasurementOutput(
        MeasurementSample measurementSample
    );

    void generateComparisonOutput(
        ComparisonEvaluationResult result
    );

    void generateFormulaOutput(
        FormulaEvaluationResult formulaEvaluationResult
    );

    void generateAnnotationOutput(
        AnnotationEvaluationResult annotationEvaluationResult
    );

    void close();
}
```

The description of the interface methods:

init SPL evaluator calls this method to initialize the evaluator output implementation with runtime configuration, evaluated context, statistical value checker and output directory. The output implementation can throw exception **OutputNotInitializedException** which is defined in the **EvaluatorOutput** interface.

generateMeasurementOutput
This method is called after creating comparison evaluation result with the measurement samples compared in it (instances of class **cz.cuni.mff.spl.evaluator.statistics-.MeasurementSample**).

generateComparisonOutput
This method is called with created comparison evaluation result (instance of class **ComparisonEvaluationResult**).

generateFormulaOutput
This method is called with created formula evaluation result (instance of class **FormulaEvaluationResult**).

generateAnnotationOutput
This method is called with created annotation evaluation result (instance of class **AnnotationEvaluationResult**).

close
SPL evaluator calls this method when evaluation is finished.

*Note: Classes **ComparisonEvaluationResult**, **FormulaEvaluationResult** and **AnnotationEvaluationResult** are located in package **cz.cuni.mff.spl.evaluator.output-.results**.*

The evaluation output implementation has to implement all methods which are described above, but it can process only those calls to **generate** methods which are necessary in the specific output implementation. For example graph generation output handles only calls to the methods **generateMeasurementOutput** and **generateComparisonOutput** as only the measurement and comparison evaluation has graphs in the output.

7.3.1 HTML output

The implementation for generating HTML report about SPL evaluation results is located in package **cz.cuni.mff.spl.evaluator.output.impl.html2**.

HTML report is the most complex currently implemented type of evaluation output. It uses XSLT[5] for the HTML pages creation.

Main class **Html2EvaluatorOutput** collects evaluation result objects produced by evaluation process and stores internally for HTML creation. HTML files are produced when the **Html2EvaluatorOutput** instance method **close** is called, because we need to prepare all file names to be able to produce links between created HTML pages.

[5]XSLT 2.0 specification `http://www.w3.org/TR/xslt20/`

Used XSLT templates are located in project folder `/src/xslt/` in package **spl.xslt**. Those files are compiled to separate JAR file with name **SPL-xslt-resources.jar** to allow easy modifications of HTML output templates.

Important files in package **spl.xslt** - those files have to be present in order to generate HTML output:

main.xsl
: Main XSLT template which is used by HTML output to generate all HTML pages. Type of page is determined by name of root element of transformed XML file. This template is invoked once for each generated page with created transformation descriptor (instance of ***ResultDescriptor**) serialized to XML.

  The XSLT transformation is performed for every created HTML file.

copy-to-output.txt
: List of files from this package, which should be copied to the output directory (such as CSS files, images or JavaScript files). Each line not starting with character # is name of one file to copy to the HTML output directory.

Important notes about main.xsl

main.xsl is a simple XSLT script.

It defines following two parameters:

CURRENT_TIME
: Sets the time of the HTML output generation which is be used in the transformation template.

BACKLINK
: Sets the target file for the back link shown in the page header.

Then there are includes with special protocol with name **spl** which allows to include other templates which are located in the same location as **main.xsl** (i.e. on JVM class path in package **spl.xslt**). No sub-folder/sub-package location is supported. Those included scripts describe transformation of specific result descriptor XML files which are transformed with **main.xsl**.

The only template rule in this script matches root node of document. It loads *shared document* with special name **?shared-info.xml** using protocol **spl**. This load is here to allow sharing of stable context information between calls to XSLT template. This *shared document* contains evaluated context information, build and evaluation configuration and mapping of HTML page names to evaluated objects to allow linking pages. The XSLT transformation is run on modified XML document, which is created by merging original document with the *shared document* while preserving root node name.

How to modify used XSLT templates

There are two options how to modify HTML output templates. You can modify them directly in code folder and recompile whole framework or you can unpack **SPL-xslt-resources.jar**, modify templates and update them in **SPL-xslt-resources.jar** (as JAR file is just ZIP archive).

7.3.2 XML output

The implementation for generating XML file describing **cz.cuni.mff.spl.evaluator.output-.results.ResultData** is located in package **cz.cuni.mff.spl.evaluator.output.impl.xml**.

The class **XmlEvaluatorOutput** is implementation of **EvaluatorOutput** interface and it just collects evaluation result objects produced by the evaluation process and stores them to the internal instance of **ResultData** which is serialized to the output file with name **spl-result.xml** when the **XmlEvaluatorOutput** instance method **close** is called.

7.3.3 Graph output

The implementation for generating graphs for measurements and comparisons is located in package **cz.cuni.mff.spl.evaluator.output.impl.graphs**.

The implementation has two classes. The main implementation class is **GraphEvaluatorOutput** and support class **GraphKeyFactory**.

The class **GraphEvaluatorOutput** is implementation of **EvaluatorOutput** interface. Types of generated graphs are specified in the evaluation configuration. This class provides access to mapping of generated graphs to keys produces by class **GraphKeyFactory**.

Any evaluation output which needs access to generated graphs has to have access to instance of **cz.cuni.mff.spl.evaluator.output.BasicOutputFileMapping** provided from the instance of **GraphEvaluatorOutput** and has to be called after this instance (otherwise there would not be mapping entries for the processed measurement or comparison). User of the **BasicOutputFileMapping** instance gets proper key for mapped object and graph type via call to **GraphKeyFactory.createGraphKey**.

For more details about generated graph types see the next section.

7.4 Graph generation support

Graph generation support is located in the package **cz.cuni.mff.spl.evaluator.graphs-.GraphDefinition**. It uses **JFreeChart**[6] library.

Support for following graph types is implemented:

histogram
: Histogram for collection of measurements.

execution times graph
: Graph describing the measured times. A dot is plotted for every measured sample value.

probability density graph
: Graph which shows probability density. Adds normal density comparison to plot when created for one measurement sample (normal density is computed with mean and standard deviation of measured sample).

[6]JFreeChart library web page `http://www.jfree.org/jfreechart/`

Graph of any type from the list is generated by the definition described with class **GraphDefinition**. This graph definition contains graph type and measured data clip type (none, sigma or quantile) with parameters.

Measured data clip type:

none
: All measured samples should be plotted.

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.

quantile
: Measurement samples that are lower specified lower percentile or higher that specified upper percentile of measurement dataset are removed. Lower and upper percentile is specified as double number in the interval [0,1].

The graph generation factory class is **GraphProvider**. Its constructor needs instance of evaluation configuration and working directory. It has two methods:

JFreeChart createChartFor(GraphDefinition graphDefinition, MeasurementSampleDescriptor... samples)
: Creates chart instance for specified graph definition and measurement samples.

byte[] createChartPNGFor(GraphDefinition graphDefinition, MeasurementSampleDescriptor... samples)
: Creates chart instance for specified graph definition and measurement samples and encodes it to byte array in PNG format. Image dimensions are configured in evaluation output configuration, the default dimensions are 800 x 600 pixel.

Those methods can throw the **cz.cuni.mff.spl.evaluator.input.MeasurementDataProvider.MeasurementDataNotFoundException** when measurement data were not found for any samples.

The details for graph types implementations are described in the following text. Each implementation provides one method to get **JFreeChart** instance of graph and one method to get **byte** array with encoded plot PNG image.

7.4.1 Histogram support details

The histogram creation implementation is in class **HistogramCreator**. It can be configured with minimum and maximum number of bins that can be shown in the plot. Default bin count limits are 100 and 10000.

The actual number of bins that will be plotted to plot is calculated by the static method **int HistogramCreator.calculateBinCount(double[] data, int minimumHistogram-BinCount, int maximumHistogramBinCount)**. This method tries to determine number of bins according to the algorithm described below and if it does not fit into requested range, than it is set to the closest value in the range.

Algorithm to determine the estimated bin count for histogram:

Source: `http://www.ehow.com/how_8485512_determine-bin-width-histogram.html`

1. Calculate the value of the cube root of the number of data points that will make up your histogram.

   For example, if you are making a histogram of the height of 200 people, you would take the cube root of 200, which is 5.848.

2. Take the inverse of the value you just calculated.

   The inverse of 5.848 is 1/5.848 = 0.171.

3. Multiply your new value by the standard deviation (s) of your data set. The standard deviation is a measure of the amount of variation in a series of numbers.

   If the standard deviation of your height data was 2.8 inches, you would calculate (2.8)(0.171) = 0.479.

4. Multiply the number you just derived by 3.49. The value 3.49 is a constant derived from statistical theory and the result of this calculation is the bin width you should use to construct a histogram of your data.

   In the case of the height example, you would calculate (3.49)(0.479) = 1.7 inches. This means that, if your lowest height was (for example) 5 feet, your first bin would span 5 feet to 5 feet 1.7 inches. The height of the column for this bin would depend on how many of your 200 measured heights were within this range. The next bin would be from 5 feet 1.7 inches to 5 feet 3.4 inches, and so on.

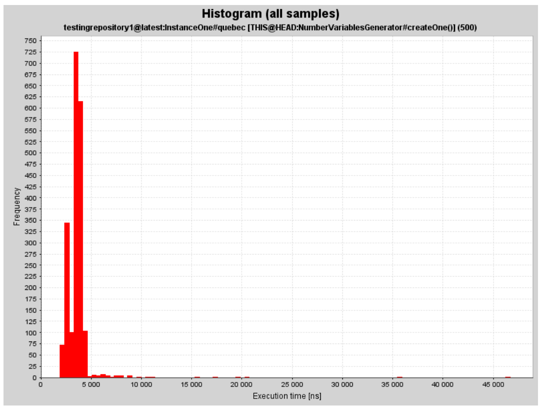

Figure 7.1: Histogram graph example

7.4.2 Execution times graph support details

The time diagram creation implementation is in class **TimeGraphCreator**. It has no special configuration options and it just plots execution times.

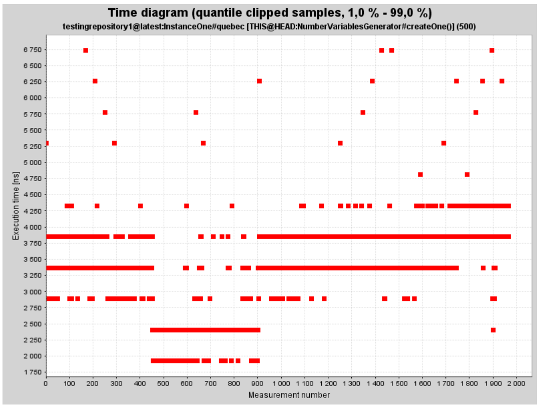

Figure 7.2: Execution times graph example

7.4.3 Probability density plot support details

The probability density plot creation implementation is in class **ProbabilityDensityGraph-Creator**. It can be configured with maximum Y axis value to show (the default is 10^{-5}).

The plot creation uses **The R Project for Statistical Computing**[7] to obtain coordinates for plotting probability density function for set of measured values as the **Apache Commons Math library** does not have equivalent methods.

If the **R Project** executable is not available, than probability density estimation is computed using algorithm desribed in `http://people.sc.fsu.edu/~hnguyen/Presentation/hoa_Auburn_modified_0327.pdf`. This algorithm is implemented in the class **cz.cuni.mff-.spl.evaluator.graphs.ProbabilityDensityGraphCreator.LambdaVoronoi**. This is just a fall-back implementation in case that **R** is not available and its results are just to sketch a preview on probability density function.

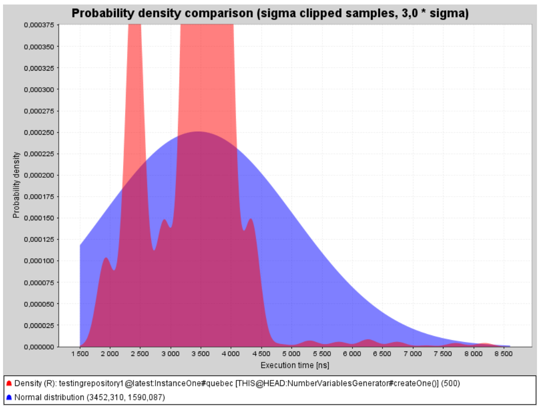

Figure 7.3: Probability density comparison graph example

[7]The R Project web page `http://www.r-project.org/`

7.4.4 The R Project invocation

The R Project invocation is implemented in class **cz.cuni.mff.spl.evaluator.r.RProject-Caller**.

Instance of this class needs two parameters in constructor - path to Rscript executable (default is **Rscript** in constant **R_RUNTIME_DEFAULT**) and file representing directory where temporary files will be created.

Each call to the method **getDensitySeries** requires parameters for double array of data to compute series for and name for the series. The return value is an instance of **XYSeries** from the package **org.jfree.data.xy** which is used in the graph creation. The Rscript executable is used to get values for **XYSeries**:

1. Files for R script, X axis values and Y axis values are created in temporary folder used by the **RProjectCaller** instance (*spl-rscript-<#>.r*, *spl-rscript-x-<#>.txt*, *spl-rscript-y-<#>.txt*, where <#> is number of Rscript execution this JVM instance).
2. R script is prepared and written to prepared file (*spl-rscript-<#>.r*). It has the following content:

   ```
   # SPL Evaluator generated temporary file
   density.adjust <- 1
   data <- as.double(c(
   1, 2, 3, ..., N # measured values to compute probability series for
   ))
   dst <- density(data, adjust=density.adjust)
   write(dst$x, file="full path to spl-rscript-x-<#>.txt")
   write(dst$y, file="full path to spl-rscript-y-<#>.txt")
   ```

3. The **Rscript** executable command is executed with full path to the created R script file as parameter and **XYSeries** is constructed after it terminates from X axis and Y axis files and is returned.

8. Using SPL as a library

This chapter describes how to use the *Framework* as a library in other application to invoke, observe and control the execution process of the *Framework*.

There is special class **cz.cuni.mff.spl.InvokedExecution** which encapsulates the *Framework* execution.

Create new instance and call method **run** (usually in a separate thread) to run the execution process.

If you need to cancel the running execution, than call method **cancelExecution**.

8.1 InvokedExecution class

The **run** method starts the execution with the provided arguments as its parameters. Note that there can be only one running execution inside one JVM instance started through class **InvokedExecution**. The execution processing is synchronized on the class type and if you run more instances concurrently, than one will process the whole execution and the others will wait in a queue for the first one to finish.

The main reason to serialize the execution processing is to prevent mixing of the log messages.

There are two ways to run more than one execution concurrently.

The first one is to run them in separate JVM instances.

The second one is to load the *Framework* library with separate class loaders inside one JVM. This way of the concurrent execution is not recommended as you might run to problems with the permanent generation memory (**PermGenError**) in the running JVM. All classes are loaded multiple times and the JVM does not perform garbage collection on loaded classes by default.

The full signature of the **run** method:

```
String cz.cuni.mff.spl.InvokedExecution.run(
  boolean abortOnThreadInterrupt,
  File xml,
  File wd,
  File ini,
  String machine,
  InteractiveInterface interactive,
  Appendable outputTarget,
  int logDetailLevel,
  boolean acceptExceptions
) throws SplRunError
```

Description of InvokedExecution.run method arguments

boolean abortOnThreadInterrupt
: This flag indicates if the execution should abort, when the running thread is interrupted.

java.io.File xml
: File with the SPL project configuration details.

java.io.File wd
: File instance pointing to the working directory which should be used for the execution.

java.io.File ini
: INI file with additional configuration details. Can be `null`.

java.lang.String machine
: The identification of machine, where to process the measurements.

cz.cuni.mff.spl.utils.interactive.InteractiveInterface interactive
: Instance of the interactive interface used when remote machine or repository requires further authentication details, or `null` when interactivity is not allowed (any access to the remote system which requires further authentication will then result in a failure). How to get one instance is described further in this section.

java.lang.Appendable outputTarget
: This arguments allows calling application to receive log output from the running execution through the provided **Appendable** instance. This argument can be `null` when no log output passing is requested by the caller. For more details see the notes further in this section.

int logDetailLevel
: This argument specifies which log messages should be passed to the **Appendable** instance provided in argument **outputTarget**. The value is ordinal number for desired item in the enumeration **cz.cuni.mff.spl.utils.logging.SplDynamicTargetLoggerFactory.LEVELS**. The main reason why the argument is not `null` is that it is easier to call the method over reflection (for example from the Hudson plug-in which does not have SPL library on the class path). For more details see the notes further in this section.

boolean acceptExceptions
: This flag indicates whether to include exception stack traces to the log messages passed to the **Appendable** instance provided in argument **outputTarget**.

InteractiveInterface interface argument notes

The *Framework* provides two basic implementations of the **cz.cuni.mff.spl.utils.interactive.InteractiveInterface**. The first one uses console input and output and is implemented in class **cz.cuni.mff.spl.utils.interactive.InteractiveConsole**. The second one implements simple graphical user interface using Swing[1] and is located in class **cz.cuni.mff.spl.utils.interactive.InteractiveSwingGui**. Just create a new instance of those classes with default constructor.

[1]Swing usage tutorial `http://docs.oracle.com/javase/tutorial/uiswing/`

Appendable interface argument notes

The **java.lang.Appendable** interface is used because of the following reasons:

- It is part of standard Java library, so users of the *Framework* can easily implement specific instances without any dependency on the *Framework* (for example when it is being used through class loader as in **SPL Tools Hudson Plug-in**).
- This interface is for example implemented by **java.lang.StringBuilder**, **java.lang-.StringBuffer**, **java.lang.CharBuffer** or **java.io.PrintStream**.
- We need just one method witch will accept an instance of **java.lang.String** with created log message text.

Valid user implementation of **Appendable** needs to implement only method **Appendable append(java.lang.CharSequence csq)** as only this method is used for passing log messages and it is guaranteed that the whole message is passed in one call.

Log detail level notes

Following table describes possible values for the **logDetailLevel** argument of the **run** method.

#	**LEVELS item**	**Behaviour**
< 0	-	Pass no messages
0	FATAL	Pass only fatal error messages.
1	ERROR	Pass all error and more severe messages (above in the table).
2	WARN	Pass all warning and more severe messages (above in the table).
3	INFO	Pass all informational and more severe messages (above in the table).
4	DEBUG	Pass all debug and more severe messages (above in the table).
>= 5	TRACE	Pass all messages.

8.2 InvokedExecutionConfiguration class

This class provides implementation for cancelling the started invoked execution. It behaves as a singleton and has only static methods. Its method **checkIfExecutionAborted** is called during the *Framework* execution to check if the execution was aborted and if so, then the unchecked exception **SplRunInterrupted** (defined as static inner class) is thrown.

You you should not need to manipulate with this class when you are using the *Framework* as a library.

9. Eclipse Plug-in

This chapter describes *SPL Tools Eclipse Plug-in*. We will refer to it just as the *Plug-in* in this chapter.

The *Plug-in* has three functional parts:

1. Part with the support for editing SPL annotations in Java source code.
2. Part with SPL execution support.
3. Part with support for viewing SPL execution and evaluation results.

9.1 Source code repository

The *Plug-in* source code GIT repository is located on the following URL:

`http://sourceforge.net/p/spl-tools/eclipseplugin/`

It can be cloned for example with one of the following GIT commands:

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

There are multiple branches in the repository. The production stable code is located in the branch `master` and the other branches serve for testing and implementing new features.

9.1.1 Directory layout of the Eclipse project

```
(d) .git              (GIT folder)
(d) icons             (icons and images used in user interface)
(d) lib               (folder for SPL library JAR files)
(d) META-INF          (Eclipse plug-in project meta information folder)
(-)  |- MANIFEST.MF   (Eclipse plug-in manifest)
(d) src               (Eclipse plug-in source code)
(d) src-xtext         (source code for Xtext editors)
(d)  |- src           (Xtext editors source code with  implementation)
(d)  |- src-gen       (generated Xtext editors source code files)
(-) .classpath        (Project Java class path configuration file)
(-) .gitignore        (GIT ignore file)
(-) .project          (Eclipse file with project configuration)
(-) build.properties  (build properties file)
(-) license.txt       (Eclipse plug-in license file)
(-) readme.txt        (Readme file with current development notes)
(-) plugin.xml        (Eclipse plug-in descriptor XML file)
```

9.2 Requirements

Following Eclipse plug-ins are required for running the *Plug-in*.

Xtext plug-in in version 2.3.0 or newer

The Xtext plug-in can be installed from Eclipse Marketplace or from its update site with following URL:

`http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/`
or
`http://download.itemis.com/updates/releases/`

Xtext plug-in is necessary for both compilation and runtime of the *Plug-in*.

Eclipse Modelling Tools

This group of plug-ins is required for generating Xtext files and it should install automatically with Xtext. If you are using **Eclipse Modeling Tools** release than you already have it.

Eclipse update site URLs:

- Eclipse Juno (4.2): `http://download.eclipse.org/releases/juno/`
- Eclipse Indigo (3.7): `http://download.eclipse.org/releases/indigo/`

See user documentation on further details on how to install an Eclipse plug-in.

The next dependencies which need to be satisfied are JAR files of the *Framework* which need to be placed (copied) to the **lib** directory manually.

9.3 Compilation

The *Plug-in* project is meant to compile by the Eclipse IDE and no build script is used.

You need to perform following steps to prepare the *Plug-in* project for full compilation upon its repository is cloned:

1. Run **src/cz/cuni/mff/spl/eclipseplugin/xtext/GenerateSPL_xtext.mwe2** as ***MWE2 Workflow*** to generate Xtext related files which are not present in the public repository.

   *Note: If you do not see the **MWE2 Workflow** in the Eclipse context menu **Run As**, than you are missing some of the Plug-in requirements listed in the previous section.*

   The common issues with the ***MWE2 Workflow*** execution are discussed further it this section.
2. Copy the *Framework* distribution JAR files to the **lib** directory of the *Plug-in* project.
3. Refresh the Eclipse project.
4. Rebuild the Eclipse project. Applies only when automatic rebuild in Eclipse is not enabled.

9.3.1 MWE2 Workflow related issues

Running the MWE2 Workflow to generate Xtext required source code files is very sensitive on the folder naming. Following issues were observed during the development. Each issue has a brief solution description.

Couldn't find module cz.cuni.mff.spl.eclipseplugin.xtext.GenerateSPL_xtext
: Make sure that value of variable **projectName** in the file **src/cz/cuni/mff/spl/eclipseplugin/xtext/GenerateSPL_xtext.mwe2** matches name of the *Plug-in* project folder (otherwise some files may be generated to bad directory) and that folders **src-xtext/src** and **src-xtext/src-gen** already exist in the project. The folder name is case sensitive. Default value:

```
var projectName = "eclipseplugin"
```

java.io.IOException: The path '/eclipseplugin/.../*.java' is unmapped
: Make sure that node **name** in the file **.project** matches the name of your project folder, otherwise URI mapping in the MWE2 does not work properly.

Default name of the *Plug-in* project folder is **eclipseplugin** (case sensitive even on Windows machines due to URI mapping).

MWE2 Workflow execution complains about class path issues
: Add the **src** folder to the MWE2 Workflow run configuration manually.

9.4 Package structure overview

This section describes package structure of the *Plug-in* project. All classes are placed in the package **cz.cuni.mff.spl.eclipseplugin** or its sub-packages. The following package description list uses **EP** as a short-cut for **cz.cuni.mff.spl.eclipseplugin**.

Folder `src`

EP Root package with the main *Plug-in* classes

EP.ast
AST manipulation classes

EP.dialogs
SWT dialog implementations.

EP.guiparts
GUI componets implementations.

EP.guiparts.annotationsoverview
GUI components for annotation in source code presentation.

EP.guiparts.binding
SWT binding related classes.

EP.guiparts.binding.converters
Converters for the SWT bindings.

EP.guiparts.binding.validators
Validators for the SWT bindings.

EP.guiparts.editors
GUI editation componets implementations.

EP.guiparts.editors.annotation
GUI components for annotation editors.

EP.guiparts.editors.configuration
GUI components for configuration editor.

EP.guiparts.execution
GUI compontens for the *Framework* execution.

EP.guiparts.model
Proxy classes for SWT binding to the *Framework* classes.

EP.guiparts.results
GUI components for the evalution result presentation.

EP.guiparts.shared
Basic shared GUI components implementation.

EP.preferences
The *Plug-in* preferences pages for configuration.

EP.tools
Various tools used across the *Plug-in*.

EP.tools.code
Tools related to the source code manipulation.

EP.tools.execution
Tools related to the *Framework* execution.

EP.tools.listeners
Eclipse IDE context listeners.

EP.tools.parser
Tools encapsulating the SPL parser usage.

EP.views
The Eclipse vies implementation.

EP.views.configuration
SPL project and INI configuration editors.

EP.views.execution
The *Framework* execution view.

EP.views.overview
The SPL annotations overview in the source code.

EP.views.results
The SPL evalution results presentation view.

EP.wizards
The new file wizards.

EP.xtext
The Xtext grammar declaration files.

org.eclipse.wb.swt
The resource managers used in the *Plug-in*.

Folder `src-xtext/src`

EP.xtext
: The generated Xtext grammar modules.

EP.xtext.formatting
: The Xtext formatters.

EP.xtext.ui
: The Xtext UI editors implementation.

EP.xtext.ui.contentassist
: The Xtext content assist implementation.

EP.xtext.ui.highlighting
: The Xtext highlighting implementation.

EP.xtext.ui.labeling
: The Xtext labeling providers.

EP.xtext.validation
: The Xtext input validation implementation.

Folder `src-xtext/src-gen`

This folder contains files generated by the Xtext which should not be edited by hand. They are generated after each grammar change and all manual changes would be lost.

9.5 Binding to the Eclipse IDE

The **SPL Annotation Overview** needs to listen to the event when the active editor in its perspective is changed. It adds the instance of class **cz.cuni.mff.spl.eclipseplugin.tools-.listeners.PagePartChangedListener** to the instance of **org.eclipse.ui.IWorkbench-Page** where it is located.

The second binding created dynamically managed by the **SPL Annotation Overview** is usage of the **cz.cuni.mff.spl.eclipseplugin.tools.listeners.DocumentChangedListener**. This document change listener is assigned to the active Eclipse editor and removed when the editor stops being active editor or when it is closed. This listener informs the annotation overview that the document was changed and the annotation list should be refreshed.

9.6 Source code manipulation

This section describes how the *Plug-in* accesses and manipulates the source code files, how it obtains the list of annotations and how it updates them.

9.6.1 Parsing annotations from the source code

Methods described in this section are used for obtaining the list of annotations in the active Eclipse editor. Those methods are is used in the ***SPL Annotation Overview***.

The implementation is located in the package **cz.cuni.mff.spl.eclipseplugin.ast**.

The implementation is based on the **Java Abstract Syntax Tree** which is represented by the class **org.eclipse.jdt.core.dom.CompilationUnit**. This class represents Java syntax tree and is used by the Eclipse Java editor to represent the syntactical tree for every compilation unit. The compilation unit is abstraction for both the source code file and the class file.

The class **ASTHelper** is used to obtain the CompilationUnit instance of the currently selected editor. It tries to get the shared instance of the CompilationUnit class which is used by the Eclipse IDE. This shared instance may not be present when the Eclipse active editor is not the Java editor or when the request is made before it is created on the background. If the shared instance is not present, than the **ASTHelper** requests one to be created on demand.

After acquiring the CompilationUnit instance, than it is passed to the **SPLAnnotation-Finder** class instance which uses AST visitors to inspect the Java syntax tree. The visitors find all SPL annotations in the syntax tree and create instances of **AnnotationDeclaration** class for them.

The **AnnotationDeclaration** class instance represents one SPL annotation and contains annotation field declarations (one for each of generators, methods and formula field) which are instances of the class **AnnotationFieldDeclaration**.

The **AnnotationFieldDeclaration** class instance contains list of instances of the class **StringDeclaration**. Note that the annotation field can be in code represented in one of the following ways:

- `generators = "generator alias declaration as one string";`
- `generators = "generator alias declaration " + "as multi-part string";`
- `generators = { "multiple " + "generator", "alias declarations"}`

The **StringDeclaration** class instance represents one string declaration in the annotation field (see the string examples above) which contains as many instances of the class **StringDeclarationPart** as there are concatenated strings.

The **StringDeclarationPart** class instance contains the original string literal from the source code and its position.

9.6.2 Writing modifications to the source code

Modifications are written to the source code using two different concepts. The first one uses Abstract Syntax Tree (AST) mentioned in the previous section. The second one just replaces the annotation with the newly created content.

The modification of the AST is used for adding one generator/method alias declaration or formula declaration. It is implemented in the class **cz.cuni.mff.spl.eclipseplugin.tools-.code.JavaCodeManipulator**.

The annotation replacement is used when user edits annotation with annotation editor. The implementation is in the class **cz.cuni.mff.spl.eclipseplugin.tools.code.SPLAnnotationReplacer**. The implementation finds the original annotation in the AST and replaces its location with the new content which is generated to the StringBuilder. Each declaration string is split to parts to allow smart adding new lines to the generated output and not to overflow reasonable text width.

9.7 Project configuration

This section contains information about persistent configuration which is stored in the Eclipse project which is annotated.

The Eclipse project which uses SPL has to have assigned one spl-config.xml file. This file is used for validating SPL annotations in the source code files inside the project.

The project XML configuration file is set by using the **Configure project of the active Java source code editor** button in the **SPL Annotations Overview**. The configuration file selection dialog with the project tree is shown. If the XML configuration file was set before, than it is selected. The configuration editor is opened on the selected file after the ***OK*** button is pressed.

The path to the selected XML configuration file is saved into the project settings. This setting is located in the following file under the key `ProjectSplConfig`:

`.settings/cz.cuni.mff.spl.eclipseplugin.SPLToolsEclipsePluginConfiguration.prefs`

When no XML configuration file is set for the project, than the *Plug-in* tries to load the configuration from the default file `spl-config.xml` in the project root. If this file does not exist, than the *Plug-in* tries to create it (this may fail when the project file system location is not writeable).

9.8 Basic GUI composition concepts

The graphical user interface (GUI) is created using the Standard Widget Toolkit (SWT).

The basic concept used in the *Plug-in* implementation is its composition of reusable components.

We distinguish the following types of the GUI components:

views

The views have no input file and they either reflect some information in the active editor, show another information to the user, or allow user to do some operations from within Eclipse.

The **SPL Annotations Overview** is the example of the view reflecting information in the active editor.

The **SPL Execution View** and **SPL Results Overview** provide additional SPL support for the user inside the Eclipse IDE.

The views are located in the package **cz.cuni.mff.spl.eclipseplugin.views** where each view has its sub-package.

editors
The editors allow to edit files with the improved experience than the text-only editor can offer.

The *Plug-in* provides two such editors. The first one allows editing the XML configuration file (**cz.cuni.mff.spl.eclipseplugin.views.configuration.SplConfigurationEditor**) and the second one INI configuration file (**cz.cuni.mff.spl.eclipseplugin.views.configuration.IniConfigurationEditor**).

dialogs
The dialogs are shown on top of the Eclipse IDE.

The dialog implementation classes are placed in the package **cz.cuni.mff.spl.eclipseplugin.dialogs**. There is implementation of the SPL annotation editor (**SPLAnnotationEditorDialog**), add method/generator alias (**AddAliasDialog**), add formula dialog (**AddFormulaDialog**) and a few more.

All types of the GUI parts described above use the reusable specialized components for the inner implementation to allow easier transformation of the one type to another with minimized effort (for example transformation the INI configuration editor to the dialog). Those reusable parts are located in the package **cz.cuni.mff.spl.eclipseplugin.guiparts** and its sub-packages. Those sub-packages are named by the purpose of the components inside. See the section [Package structure overview on page 42] for the details.

Those reusable components utilize both the inheritance hierarchy and the type parametrization. The basic example is the **SPL generator alias editor** with the following type hierarchy (classes without package are from the package **cz.cuni.mff.spl.eclipseplugin.guiparts.editors.annotation**):

- **org.eclipse.swt.widgets.Composite**
  - **SPLAliasEditor<cz.cuni.mff.spl.annotation.Generator>**
    - **SPLGeneratorAliasEditor**

The type parametrization is mainly used to allow common implementation of the GUI component for the generator and method aliases.

9.9 Xtext usage

The *Plug-in* has its own implementation of the SPL grammar defined in the *Framework*. This parallel implementation was necessary for the highlighting and the content assist support for the declaration editors.

The grammar implementation uses **Xtext**[1] and is located in the **src** folder in the package **cz.cuni.mff.spl.eclipseplugin.xtext** and it is

The Xtext SPL grammar is split into following files:

SPL_common.xtext
Contains SPL grammar rules for generator and method alias declarations. Does not contain the **Model** rule.

[1]Xtext web page `http://www.eclipse.org/Xtext/`

SPL__formula.xtext
Contains SPL grammar part for formula declarations. Uses the **SPL__common.xtext**. Defines **Model** rule which is entry point for the formula declaration editor (its content assist, highlighting, etc.).

SPL__generatorAlias.xtext
This grammar file just defines the **Model** rule for the generator declaration editor. Uses rules in the **SPL__common.xtext**.

SPL__methodAlias.xtext
This grammar file just defines the **Model** rule for the method declaration editor. Uses rules in the **SPL__common.xtext**.

The Xtext Java source code files generation process is described in the section [Compilation on page 41]. The generated files are placed to the following two folders:

src-xtext/src
Files generated to this folder are meant for the added implementation details.

src-xtext/src-gen
Files generated to this folder are not meant for any modifications

9.9.1 Xtext embedded editors

The *Plug-in* uses the Xtext embedded editors which were introduced in the Xtext version 2.2. The embedded editors can be placed inside another graphical user interface components such as dialogues.

The creation of the Xtext embedded editors is implemented in the class **cz.cuni.mff.spl-.eclipseplugin.xtext.SPL__EmbeddedEditorFactory** which contains three static methods for creating embedded editors. Those methods take two arguments – the first specifies the SWT **Composite** instance specifying the embedded editor parent component, the second is instance of the class **cz.cuni.mff.spl.eclipseplugin.xtext.ui.contentassist-.InteractionProvider**. which .

The instance of the class **InteractionProvider** is provided to the embedded editor content assist and highlighting implementation through the static method **getInteractionProviderForEmbeddedEditor()**. This way was chosen because this is the most straightforward way to allow content assist and highlighting implementations to interact with the current SPL parser context and the declaration editor. The method **getInteractionProviderForEmbeddedEditor()** can be called only when call to one of the **create...** methods is in progress, otherwise an **IllegalStateException** is thrown. Synchronization on the class is used to achieve this behaviour.

9.9.2 Content assist support

The content assist support allows user to easily refer to the defined SPL projects, generator and method aliases and Java types inside the Xtext editors. The context assist implementation is located in the folder **src-xtext/src** inside the package **cz.cuni.mff.spl.eclipseplugin.xtext.ui.contentassist**.

The current parser context obtained from the instance of the **InteractionProvider** is used to create proposals regarding project, revision, generator and method aliases.

The Java type proposals are created in the instance of the class **ClassDeclarationHelper** using the Java model of the Eclipse workspace. The proposal creation can't create proposals for all Java types visible in the Eclipse workspace as it would take too much time and the proposal would not be usable, so the proposals are created only for Java types which match the already typed string. Proposals are created for all matching packages and all classes in the defined packages (the already typed string and its prefix to the last dot).

9.9.3 Syntax coloring (highlighting) support

Syntax highlighting serves for better orientation when using Xtext editors for writing method and generator aliases and formulas. Every part of the text can have its own color and style which depends on the grammar element represented by the text. The highlighting implementation is located in the folder **src-xtext/src** inside the package **cz.cuni.mff.spl-.eclipseplugin.xtext.ui.highlighting**.

The current parser context obtained from the instance of the **InteractionProvider** is used for searching for projects, revisions and aliases. It allows to distinguish between known elements and elements that has not been declared so it can be marked by different color.

Class **DefaultHighlightingConfiguration** provides the set of highlighting styles and their default values. Class **HighlightingHelper** uses the parse tree created by Xtext parser to calculate proper highlighting.

9.10 SPL Annotations Overview

The **SPL Annotations Overview** shows details of the annotations in the active Eclipse Java source code editor.

Binding to the active editor

This view uses binding to the Eclipse user interface part selection change events (which is briefly described in the section [Binding to the Eclipse IDE on page 44]) using class **cz-.cuni.mff.spl.eclipseplugin.tools.listeners.PagePartChangedListener** to listen to the change of the active editor.

The currently active editor is stored in the instance of the class **cz.cuni.mff.spl.eclipse-plugin.tools.code.EditorPluginContext** which is shared between the Eclipse page listener and the annotations overview.

The Eclipse page part is considered editor when its part ID is either **org.eclipse.jdt.ui-.CompilationUnitEditor** or when it contains “editor” as a substring.

When the active editor is changed, than the document changed listeners registered in the **EditorPluginContext** are moved from the previous active editor to the new one.

Refreshing the shown annotations

The annotations shown in the **SPL Annotations Overview** are refreshed automatically when the active editor changes and when the document in the active editor changes. The refresh operation on document change is not performed immediately but with a small delay.

The delay for the refresh operation is used because the refresh operation uses the Abstract Syntax Tree for the document and it takes some time to build it. The second reason is that we wait for the user to stop making changes to the document before we perform the refresh operation as flashing view might disturb the user in his work.

9.11 Assisted annotation editing

The **SPL Annotation Editor** allows user to make modifications to the SPL annotation in the source code with the content assist, highlighting and validation support.

9.11.1 Annotation editor dialog

The editor is implemented as a dialog in the class **cz.cuni.mff.spl.eclipseplugin.dialogs-.SPLAnnotationEditorDialog**. This class is used for showing the instance of the class **SPLAnnotationEditor** from the package **cz.cuni.mff.spl.eclipseplugin.guiparts.editors.annotation** to the user.

The dialog has two buttons – ***Save*** and ***Cancel***. The save button is enabled only when the annotation editor part is *valid* (it is currently valid when no its inner declaration is edited).

The dialog is created and show in the method **showSplAnnotationEditor** of the class **cz.cuni.mff.spl.eclipseplugin.tools.StandaloneOperations**.

9.11.2 Annotation editor component

The class **cz.cuni.mff.spl.eclipseplugin.guiparts.editors.annotation.SPLAnnotationEditor** provides the editing capabilities for the SPL annotation represented by the instance of the class **cz.cuni.mff.spl.eclipseplugin.ast.AnnotationDeclaration**.

This component uses all other annotation editor components located in the package **cz.cuni-.mff.spl.eclipseplugin.guiparts.editors.annotation**. either directly or indirectly.

The hierarchy of the composition is shown in the following list:

- **SPLAnnotationEditor**
    - **SPLFormulasEditor**
        - **SPLFormulaEditor**
    - **SPLGeneratorAliasesEditor**
        - **SPLGeneratorAliasEditor**
    - **SPLMethodAliasesEditor**

- **SPLMethodAliasEditor**

The second level editors allow to edit multiple declarations and those declarations are edited in the third level editors.

The formula editor requires that the instance of the **cz.cuni.mff.spl.formula.context-.ParserContext** is provided to the basic SPL parser with the declared aliases. The information about the annotation declarations are maintained in the instance of the class **cz.cuni.mff.spl.eclipseplugin.tools.AnnotationDeclarationEditContext**.

The method and generator declarations editors are used to edit declarations obtained from the **AnnotationDeclarationEditContext**. which then provides the **ParserContext** instance to the formulas editor. There are few issues which need to be taken in account when the user edits the annotation:

- The alias declaration have parser errors when parsed with the basic SPL parser. Such declaration should be corrected by the user as it will not be used in any formula by the *Framework*.

  The solution for this issue is not to pass such declaration to the formula editor in the **ParserContext** instance.
- The alias declaration name may be duplicate. There is no guarantee which one of the duplicate aliases would be used by the *Framework* and it is considered as an integrity error. This may happen when the project configuration already has an alias with specified name.

  The used solution for this issue is not to pass declarations with duplicate aliases to the formula editor in the **ParserContext** instance. Note that when the global alias with the duplicate name is defined, than it is passed to the **ParserContext** instance.

The multiple declaration editors (**SPLFormulasEditor**, **SPLGeneratorAliasesEditor**, **SPLMethodAliasesEditor**) show the list of current declarations with the validity state and problem message (if any) for each declaration. They allow to add new declaration and edit or delete the selected one.

The single declaration editors (**SPLFormulaEditor**, **SPLGeneratorAliasEditor**, **SPL-MethodAliasEditor**) serve only for the assisted editing and validation of the declaration. They use the **Xtext** embedded editors which utilize the **Xtext** grammar for SPL (see the section 47). The embedded editors use the Xtext grammar for the content assist and highlighting support, but the declaration editor uses the basic SPL parser for determining if the declaration text is valid or not. This behaviour was chosen, because the *Plug-in* development may respond to the potential changes made to the SPL grammar in the core project with a delay.

9.12 Invoking execution

The *Framework* execution can be invoked directly from within the *Plug-in* using the **SPL Execution View**. The invocation uses the invocation methods described in the chapter [Using SPL as a library on page 37].

9.12.1 SPL Execution View

The **SPL Execution View** (**cz.cuni.mff.spl.eclipseplugin.views.execution.SPLExecutionView**) contains instance of the class **cz.cuni.mff.spl.eclipseplugin.guiparts.execution.StartExecutionControl** which allows user to set execution parameters, start the execution, view its progress, cancel it and view results when execution finishes.

The view contains three tabs. The first one is for execution parameters and allows user to save the configuration for further usage under specified name. The execution configuration is saved using the class **cz.cuni.mff.spl.eclipseplugin.SPLToolsEclipsePluginConfiguration**. The run button starts the *Framework* execution which is described further.

The second tab allows user to observe progress of the running execution. There are two buttons – ***Cancel*** button to stop the running execution (available only while the execution is running) and ***Show results*** button which loads the execution results to the results presenter on the third tab.

The third tab contains the SPL results presenter. This presenter is the same component which is used in the **SPL Results Overview** which is described in the next section [Viewing results on page 53].

9.12.2 Invocation implementation details

The *Framework* execution invocation implementation is located in the package **cz.cuni.mff.spl.eclipseplugin.tools.execution**. The class **ExecutionStarter** starts the execution for provided instance of the class **ExecutionConfiguration** with the configuration details.

The *Framework* is loaded by its own instance of the **URLClassLoader**. This class loader load all JAR files from the **lib** folder of the *Plug-in*. The class loader is used to obtain the methods necessary for the invocation and its cancellation and to get the class instance representing the class **cz.cuni.mff.spl.utils.interactive.InteractiveSwingGui** which is used for user interaction with the *Framework*.

The execution process is encapsulated in the **ExecutionStarter** inner class **RunFrameworkEclipseJob**. This class is an Eclipse job (**org.eclipse.core.runtime.jobs.Job**) which is shown in the Eclipse **Progress view** and it can be even cancelled from there. This class creates the thread for the execution, starts it and waits for it to finish. The newly created execution thread is necessary for the cancel ability to work correctly. The execution thread just invokes the execution with the parameters provided in the instance of the **ExecutionConfiguration** class.

The execution progress is passed from the *Framework* to the *Plug-in* GUI through the **ExecutionStarter** inner class **FrameworkLoggerAppender** which implements the interface **java.lang.Appendable**. This class buffers lines produced by the execution and allows the **RunFrameworkEclipseJob** instance to apply them to the GUI. This two step update is necessary to prevent the GUI from flashing when too many updates occur in a very short time (the default update interval is 250 milliseconds).

9.13 Viewing results

The *Plug-in* contains **SPL Results Overview** which allows user to browse the SPL evaluation results. The results can be located either on local file system, or accessed over HTTP protocol.

The SPL evaluation results can be also browsed in the third tab of the **SPL Execution View**.

9.13.1 SPL Results Overview

The implementation of the **SPL Results Overview** is the class **cz.cuni.mff.spl.eclipseplugin.views.results.SPLResultsOverview**. The main visual component is in the class **cz.cuni.mff.spl.eclipseplugin.guiparts.results.ResultsComposition**.

The main package for the results visual components is **cz.cuni.mff.spl.eclipseplugin.guiparts.results**.

The results are loaded from the XML result description file generated by the XML output of the evaluation (see [XML output on page 32] for more details). The path to the file can be either local or

The **ResultsComposition** component contains the area for specification of the XML result description file and a **org.eclipse.swt.widgets.TabFolder** instance for showing the results.

When the user fills the path to the results and presses the ***Refresh*** button or ***Enter*** key, than attempt to load the XML description file is made. The data load is implemented using Eclipse Job functionality to allow easy way to cancel the operation.

The first thing which is done is to acquire the read only abstraction of the file access implemented in the interface **cz.cuni.mff.spl.deploy.store.IStoreReadonly**. Following items are needed for access to information about presented results:

XML results description file (IStoreReadonlyFile)
The file with the XML description of the results. Access to this file has to be available in order to show the results. The file contains XML representation of the class **cz.cuni.mff.spl.evaluator.output.results.ResultData**.

Evaluation directory (IStoreReadonlyDirectory)
The abstraction of the folder with the evaluation results. This is used to access the graph images generated during the evaluation. Optional.

The IStoreReadonly instance for the results location
The abstraction of the whole store for accessing measured data for dynamic graph creation. Optional.

All those items are acquired by calling the static method **tryToFindStoreForPath** on the class **cz.cuni.mff.spl.eclipseplugin.tools.execution.StoreProvider**. This method checks if the location starts with *"http://"* or *"https://"* string and if so, than it uses the **cz.cuni.mff.spl.deploy.store.HttpStore** to access remote files, otherwise the **cz.cuni.mff.spl.deploy.store.LocalStore** is used.

The entire results processing from this point works only on top of the **IStoreReadonly**, **IStoreReadonlyDirectory** and **IStoreReadonlyFile** interfaces.

When the **XML results description file** is not available, than error message is shown. It is loaded otherwise and preprocessed – each instance of the class **cz.cuni.mff.spl.evaluator-.statistics.MeasurementSample** in it is assigned the store data provider for accessing the measurement sample data. The **IStoreReadonly** instance acquired for the results location path is used when available, otherwise a dummy *nothing-providing* implementation is created (it just throws exception that no data are available).

9.13.2 Navigation in the results

The navigation in the results shown in the **TabFolder** is processed over the interface **IResultsNavigator** and its implementation **ResultsNavigator** which is inner class of the **ResultsComposition**. This interface provides access to the currently shown results data, evaluation configuration, evaluation directory and methods to show specific levels of detail of the results.

Each call to one of the **show..** methods manipulates the **TabFolder** instance. All tabs from the position for the new tab to the right are removed and new tab for the specified level of detail is created. The tab creation includes creation of the scrolling composite to provided better user experience and better resizing support.

9.13.3 The result details visual components

The result details GUI components are all located in the package **cz.cuni.mff.spl.eclipse-plugin.guiparts.results**

Following list provides description of the result presentation visual components

ResultsComposition
: This is the main component which manages shown results.

EvaluationResultsOverview
: Shows the evaluation results overview summary. Uses the **AnnotationOverview-PresenterTreeView** component.

AnnotationDetail
: Shows details about one annotation location. Uses the **FormulasInAnnotation-PresenterTreeView** component.

FormulaDetail
: Shows details about one SPL formula inside the SPL annotation. Uses the **Formula-DetailEvaluationPresenterTreeView** component.

ComparisonDetail
: Shows details about one comparison in the SPL formula with its graphs.

MeasurementDetail
: Shows details about one SPL measurement with its graphs.

ParsedDeclarationsPresenterTreeView
: This generic component is used for presenting SPL aliases and formulas in the .

SimpleIniPresenter
: This component is used to present the configuration values which were used for the evaluation and configured through the INI file.

9.13.4 Graph presentation

The graph presentation is done by the **GraphPresenter** component. This component tries to obtain the graph for the specified graph definition.

The graph presenter tries to create dynamic graph first using the measured values and provided definition using the class **cz.cuni.mff.spl.evaluator.graphs.GraphProvider** which uses the **JFreeChart** library. This fails when the measurement sample data are not available. When the data are available, than the **org.jfree.chart.JFreeChart** instance is obtained and enclosed into the instance of the class **org.jfree.chart.ChartPanel**. **ChartPanel** is a **Swing**[2] component and it is integrated to the Eclipse SWT user interface through the method **new_Frame** of the class **org.eclipse.swt.awt.SWT_AWT**. Graph created through this method is dynamic and acts as an active component allowing user for example to zoom in and out.

When the graph generation fails, than the graph presenter tries to load the graph image file for the specified graph definition which was created during the evaluation. When this file is found, than the image is loaded and presented as a static graph which has no more abilities compared to the graphs generated by previously described way.

When the graph generation failed and no graph image was generated during evaluation, than the message saying that there is no graph data is show.

The graph presentation uses the following components:

GraphPresenter
: This component serves for the presentation of one graph and is used in the **ComparisonDetail** and **MeasurementDetail** components.

GraphConfigurationProvider
: This component is placed before **GraphPresenter** instances in the **ComparisonDetail** and **MeasurementDetail** components. It contains button for showing the graph configuration dialog and is also used to update any existing instance of the **GraphPresenter** component when the height or width in the configuration is changed.

GraphConfigurationEditor
: This component is used to configure graph presentation options. It allows to set minimum width and height for the generated graphs, path to the R project executable (which is used for the probability density estimation calculation, see the section [Graph generation support on page 32] and types of graphs that will be shown in the comparison result detail and measurement detail.

The set of graph types that are shown in the comparison result detail and measurement detail is determined by the union of graphs configured for showing in the results and graphs configured for generation during the evaluation.

[2]Swing is an alternative to SWT for creating graphical user interface

9.14 SPL configuration editing

The **SPL configuration editor** allows user to make modifications in the XML configuration file with validation support.

The editor is implemented by class **cz.cuni.mff.spl.eclipseplugin.views.configuration-.SplConfigurationEditor**. It provides operations specific for editors and shows **cz.cuni-.mff.spl.eclipseplugin.guiparts.editors.configuration.ConfigurationEditor** as its content which is composed of other sub editors.

There are two kinds of sub editors. First kind contains **SPLMethodAliasesEditor** and **SPLGeneratorAliasesEditor** from package **cz.cuni.mff.spl.eclipseplugin.guiparts.editors.annotation** for editing global aliases. They are described in section [Assisted annotation editing on page 50]. The second kind contains editors **ProjectsEditor** (for projects configuration) and **StringPairTableEditor** (for parameters configuration) from package **cz.cuni.mff.spl.eclipseplugin.guiparts.editors.configuration**.

9.14.1 Table editors

The **StringPairTableEditor** is one of universal table editors specified for editing pairs of string values. It extends abstract **TableEditor<T>** which is the base for all table editors used in the *Plug-in* parametrized by the object where value of single item is stored. Every table editor use own table item dialog for editing single table item.

The type hierarchy of the table editors with its dialogs and edited items is shown in the following list:

- **TableEditor<T>** use **TableItemDialog<T>** (abstract ancestor)
    - **StringPairTableEditor** use **StringPairDialog** (edit pairs of strings)
        - **IniSectionEditor** use **IniEntryDialog** (edit INI entries)
    - **StringTableEditor** use **SingleStringDialog** (edit single strings)

Where **TableItemDialog<T>** is immediate abstract ancestor of every concrete table item dialog in the list. The concrete table editors and dialogs are designed to be extended further for specific purposes.

9.14.2 Projects configuration editors

The **ProjectsEditor** uses other configuration editor components located in the package **cz.cuni.mff.spl.eclipseplugin.guiparts.editors.configuration**.

The hierarchy of the composition is shown in the following list:

ProjectsEditor

- **ProjectEditor**
    - **StringTableEditor**
    - **BuildEditor**
    - **RepositoryEditor**

- **RevisionsEditor**
  - **RevisionEditor**

In the **ProjectEditor** is used extended **StringTableEditor** two times. First is configured for editing class paths and second for editing scan patterns.

These editors use classes from package **cz.cuni.mff.spl.eclipseplugin.guiparts.binding-.validators** for validating input values. The validity is reported via classes in **cz.cuni-.mff.spl.eclipseplugin.guiparts.binding** package. Editors values are stored in model objects with property change support from package **cz.cuni.mff.spl.eclipseplugin.guiparts-.model**.

9.15 INI configuration editing

The **INI configuration editor** allows user to make modifications in the INI configuration file with validation and entry description support.

The editor is implemented by class **cz.cuni.mff.spl.eclipseplugin.views.configuration-.IniConfigurationEditor**. It provides operations specific for editors and shows **cz.cuni-.mff.spl.eclipseplugin.guiparts.editors.configuration.IniEditor** as its content which is composed of **cz.cuni.mff.spl.eclipseplugin.guiparts.editors.configuration.IniSection-Editor** sub editors. The **IniSectionEditor** is concrete table editor as is described in previous section. It is extended **StringPairTableEditor** configured for editing INI entries.

10. Hudson Plug-in

SPL Tools Hudson Plug-in is integration of the *SPL Tools Framework* for **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 the *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.

The *Plug-in* allows to run SPL Tools Framework evaluation on 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).

10.1 Compilation from source

This section describes steps to compile the plug-in package for Hudson from the source code. The compiled *Plug-in* package can be downloaded from the SPL Tools web page:

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

10.1.1 Source code repository

The *Plug-in* source code is provided in the form of Git repository on one of the following URLs:

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

Example git command to clone repository:

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

10.1.2 Directory layout of the project

The following directory and file tree describes the project directory layout.

```
(d) .git             (GIT folder)
(d) lib              (folder for SPL library JAR files)
(d) src              (folder for the Hudson plug-in source code files)
(d)  |- main         (production source code files)
(d)     |- java      (Java source code files)
(d)     |- resources (HTML and Jelly files for the Hudson user interface)
(d)     |- webapp    (icons for the Hudson user interface)
(d) target           (folder for compiled binaries, created dynamically)
(-) .gitignore       (GIT ignore file)
(-) license.txt      (Hudson plug-in license file)
```

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

```
(-) README.txt         (Readme file with current development notes)
(-) pom.xml            (Apache Maven Project Object Model file)
```

More files and directories may be created during the compilation with Apache Maven.

10.1.3 Dependencies

Plug-in uses **Apache Maven 3**[2] for external dependency management, compilation, building and packaging, so you will need to have this tool.

You need to copy *SPL Tools Framework* distribution JAR files to folder **lib** which is placed in the plug-in project root directory[3]. This step is necessary as Maven does not resolve SPL dependency. The SPL binary package can be downloaded from the following URL:

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

10.1.4 Compilation

Compilation is done by running command **mvn** in repository clone root directory.

Apache Maven downloads all required libraries and dependencies to compile (except *Framework* JAR files mentioned above), runs compilation and produces Hudson Plug-in package in folder **target** with name **spl-tools-hudson-plugin.hpi**.

Note that running the compilation for the first time can take a long time as Apache Maven downloads many dependencies for the Hudson runtime.

10.1.5 Importing *Plug-in* to the Eclipse IDE

The *Plug-in* can be imported to the Eclipse IDE as an Eclipse project. The Eclipse project needs few configuration files, the most important ones are **.project** and **.classpath**. Those files are not placed in the source code repository as **.classpath** requires references to various JAR files which are obtained using the Apache Maven and are then computer specific.

You can generate the necessary files with the following commands procedure:

1. Compile the *Plug-in* according to the previous sections
   (a) Place *Framework* JAR files to the **lib** directory.
   (b) Run `mvn` command in the *Plug-in* project root.
2. Run command `mvn eclipse:eclipse` to generate files for the Eclipse IDE.

Now the *Plug-in* project can be imported to the Eclipse IDE[4].

[2]Apache Maven web page `http://maven.apache.org/`

[3]All JAR files in build/dist/ in compiled *SPL Tools Framework*

[4]Importing project to Eclipse tutorial `http://help.eclipse.org/helios/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Ftasks%2Ftasks-importproject.htm`

10.2 Implementation details

This section contains the implementation details for the *Plug-in*. It contains brief overview about Hudson plug-in development and terminology, brief description of all classes, details about running SPL execution and providing access to the SPL evaluation results over HTTP protocol from within Hudson instance.

10.2.1 Basic information on Hudson plug-in development

Hudson uses its own JAR-like format for the plug-in distribution. The plug-in files have the extension **hpi** and are similar to the JAR files. The following sections contain basic information necessary to understand how Hudson uses its plug-ins.

Hudson extension point usage

When the plug-in is installed into Hudson, than it is extracted from the HPI file into its own folder and loaded into the Hudson instance with a separate class loader. Hudson instance loads the plug-in classes and finds all classes with the extension implementation.

A class is recognized by Hudson as its extension implementation when it contains an inner class marked with the Java annotation **hudson.Extension** which extends Hudson plug-in extension descriptor (for example **hudson.tasks.BuildStepDescriptor**). This inner class is called as the **plug-in descriptor**.

Hudson examines the plug-in descriptor inheritance hierarchy and assigns it to the proper extension group (such as build step or presenter extension). The plug-in descriptor instance is used for its configuration in the user interface.

The extension implementation class (which contains the plug-in descriptor class as its inner class) is instantiated only when the extension is used (for example, build step extension is instantiated only when a build is in progress).

Contributing to the user interface

Hudson uses the Jelly[5] language to create the user interface components. The Jelly allows to combine HTML fragments with its own constructs and Java code invocation.

The Hudson plug-ins can create instances of **hudson.model.Action** interface implementation which are used to create and process actions in the Hudson web user interface. Those actions are represented with links in the user interface and they process requests made to their URL.

Each **hudson.model.Action** instance should have its Jelly description file or files. Further description of the Jelly syntax and usage in Hudson is out side of the scope of this documentation and you should refer to the official Hudson documentation which is available on the following URL:

`http://wiki.eclipse.org/Hudson-ci`

[5]Basic guide to Jelly `http://wiki.hudson-ci.org/display/HUDSON/Basic+guide+to+Jelly+usage+in+Hudson`

10.2.2 Description of classes

This section contains description of all classes used in the *Plug-in*, because there are very few classes for its implementation compared to the **SPL Tools Eclipse Plug-in**. The classes are described in the order of their usage during the *Plug-in* runtime inside a Hudson instance. All classes with the *Plug-in* implementation are placed to the package **cz.cuni-.mff.spl.eclipseplugin**.

SPLToolsDescriptor

This class contains basic implementation for the *Plug-in* descriptor.

Note that as this class implementation is standalone it does not act as a Hudson plug-in descriptor itself – this implementation can be used with multiple Hudson plug-in extension points. This design decision allows easier extension of the *Plug-in* with reusing existing implementation for more Hudson extension points.

SPLToolsHudsonPlugin

This is the main class of the *Plug-in* implementation. It processes build requests.

Note that as the **SPLToolsDescriptor** this class is not Hudson extension itself and it also contains only the execution implementation.

SPLBuilder

This class is the implementation of the *Plug-in* extension for Hudson. It extends the **hudson.tasks.Builder** class to create a Hudson builder extension.

This class uses an instance of the class **SPLToolsHudsonPlugin** for processing the build requests. It contains the public static inner class **SPLDescriptorImpl** which extends the **SPLToolsDescriptor** and which is annotated with the Java annotation **hudson.Extension** to act as Hudson plug-in descriptor.

ExecutionConfiguration

This class represents execution configuration details. It is serializable and it is used to store the values configured in the Hudson user interface and to provide them to the SPL execution processing.

ExecutionStarter

This class contains the *Framework* execution invocation implementation.

The *Framework* invocation relies on fact, that the *Plug-in* plug-in classes are loaded by their own separate class loader of the type **java.net.URLClassLoader**.

This class loader is used to locate the plug-in directory where are the *Framework* JAR files located. New **URLClassLoader** is created for loading the JAR files and it is then used for the *Framework* invocation[6].

SPLBuildPublisher

This class is responsible for publishing of the SPL evaluation results after successful *Framework* execution. It is used by the **SPLBuilder**.

It copies the evaluation results file into directory for the executed Hudson build. It creates the directory **spl-tools-reports** in it. This directory is used for all *Plug-in* instances which are performed during the Hudson build.

[6]See [Using SPL as a library on page 37] for further details on the *Framework* invocation.

Then it creates a subdirectory with name **SPL_<identification>** which is specific to the **SPLBuilder** instance. The identification in the directory name is just call to **System.nanoTime()** encoded to the hexadecimal string.

Then a directory **evaluation** is created under **SPL_<identification>** and the evaluation result files are copied there.

This procedure ensures that the SPL evaluation results is available as long as the build results are maintained in the Hudson job.

SPLBuildAbstractAction
This abstract class contains basic shared implementation for the other **hudson.model-.Action** implementation. It provides access to the Hudson instance URL and *Plug-in* images and icons.

SPLBuildFailedAction
This action is used to present results of the *Framework* executions that were either cancelled or that failed due to fatal error or exception. It contains a message with the failure description.

SPLBuildReportAction
This action is used to present execution results summary for *Framework* executions that finished correctly (not cancelled, no fatal errors). Its presentation shows links to HTML report, XML result description file (and URL to it for usage in the **SPL Tools Eclipse Plug-in**) and execution full log file. Note that only links to really created files are shown.

The other purpose of this action is to serve content of the stored evaluation results from the build folder over HTTP protocol. The details are described further in this chapter.

SPLBuildReportPresentAction
This action is used to show the **SPL Tools HTML Report** link in the build menu. It shows the HTML report inside the Hudson web user interface. The HTML report is shown inside an `<iframe>` tag.

The Hudson web user interface Jelly files for the classes described above are located in the following directories as the Hudson plug-in development suggests:

```
src/main/resources/cz/cuni/mff/spl/hudson/<class name>/
```

The class **SPLBuilder** has Jelly files for its configuration and help for configuration parameters.

The classes **SPLBuildFailedAction** and **SPLBuildReportAction** have only Jelly template for the summary information which is shown on the build status page.

The class **SPLBuildReportPresentAction** has Jelly template for the whole Hudson web interface page and it contains an `<iframe>` tag with link to the HTML report.

10.2.3 HTTP access to the evaluation results

The *Plug-in* provides access to the SPL evaluation results over HTTP protocol from within Hudson instance. The implementation is located inside the class **SPLBuildReportAction**. This class contains all file paths necessary for accessing the SPL evaluation files stored inside build results directory and the *Framework* working directory configured for the processed build step.

The HTTP request processing is done in the method **doDynamic** which handles all requests for the URL specified inside the instance **SPLBuildReportAction**.

This method checks if the request URL matches path to the evaluation results (i. e. if it leads to the **evaluation** directory) and if so, than an instance of the class **hudson-.model.DirectoryBrowserSupport** with root in the build result subdirectory specific to the **SPLBuildReportAction** instance is used to handle the request.

Otherwise, the **DirectoryBrowserSupport** with the root in the *Framework* working directory is used to handle the request. This allows the *Plug-in* to serve the evaluation results from the persistent location (build result folder) and measured data from the *Framework* working directory configured in the build step configuration which may be deleted over time.

11. Development

This chapter briefly describes some improvements that came to our minds during work on the project but could not be implemented due combination of lack of time and complexity. Also libraries and tools used in the project are listed here together with work timeline and progress.

11.1 What to improve

Add more version control systems

The *Framework* is prepared for adding support for more more version control systems such as **Bazaar**[1] or **Mercurial**[2].

The necessary implementation steps are described in the section [Extension on page 16].

Add platform specific time measurement

The current implementation of the time measurement relies on usage of the method **System.nanotime()**. This implementation is portable between platforms and the resolution of the **nanotime** method is believed sufficient in Java 7 implementation. However various platforms offer optimized platform specific ways to measure time. Further development should add support for platform specific time measurement to improve measurement results on the various platforms. This extension has to include adding ability to select proper implementation for the machine where the measurement is to be performed as the measurement sampling code has to reflect usage of the platform specific time measurement.

Logging

The current logging implementation does not support concurrent *Framework* execution, because the log outputs would get mixed together.

HTML report

The HTML report offers many ways for improvements. It may utilize JavaScript for enhanced interactivity or it may use frames like the JUnit does.

Eclipse plug-in improvements

The Eclipse plug-in allows to make numerous improvements. Following list contains some of them:

[1]Bazaar web page `http://bazaar.canonical.com/`
[2]Mercurial web page `http://mercurial.selenic.com/`

- Improve content assist support for Java types.
- Add support for viewing all SPL annotations in an Eclipse project. Annotation could be obtained either by using the Abstract Syntax Tree for Java types, or using the SPL scanner on build class files.
- Add support for formula evaluation on demand. The measurement data could be acquired from the cache or measured on demand.
- Add support to run evaluation for specified formula only.
- Add support for assisted editing of SPL annotations inside Java editor. This feature was discussed during the design phase of the project and we decided not to do it this way, because it may become obsolete with the new Eclipse release as the Java editor is marked as *internal* which means that it can change in any way without maintaining backwards compatibility.
- Add support for mapping projects form SPL configuration to the Eclipse workspace projects. This could be used in content assist for better Java type proposals when only Java types visible from the specified project would be proposed.

Hudson plug-in improvements

The Hudson plug-in does not offer many opportunities for extension compared to the Eclipse plug-in.

The main improvement would be to add ability to specify the INI configuration file content in the build step configuration. This would allow user to just copy the INI values to the configuration instead of uploading the INI configuration file to the machine running Hudson instance.

Improve integration API

The current API for SPL invocation is created for purposes of the Eclipse and Hudson plug-ins. Those plug-ins need to run the whole execution only. This public API has to be extended to allow easier integration of the *Framework* library into third party applications and in order to implement some of the new features for the Eclipse plug-in.

11.2 List of used libraries and tools

List of external libraries and tools used in this project follows.

SVNKit
Library for accessing Subversion repositories.
Version: 1.7.5
`http://svnkit.com/`

JGit
Library for accessing Git repositories.
Version: 2.1.0.201209190230
`http://www.eclipse.org/jgit/`

JSch
Library for accessing machines via SSH.
Version: 0.1.49
`http://www.jcraft.com/jsch/`

Apache Velocity
Library for code templates used for code generation.
Version: 1.7
`http://velocity.apache.org/`

Commons Math
Library for statistical computing used in evaluator and integrated generators.
Version: 3.1.1
`http://commons.apache.org/proper/commons-math/`

Apache log4j
Library for logging.
Version: 1.2.17
`http://logging.apache.org/log4j/`

Castor
Library for XML serialization and deserialization.
Version: 1.3.2
`http://castor.codehaus.org/xml-framework.html`

Commons Lang
Library required by the Castor.
Version: 3.1
`http://commons.apache.org/proper/commons-lang/`

Commons Logging
Library required by the Castor.
Version: 1.1.1
`http://commons.apache.org/proper/commons-logging/`

ini4j
Library for INI files serialization and deserialization.
Version: 0.5.2
`http://ini4j.sourceforge.net/`

JFreeChart
Library for graph generation.
Version: 1.0.14
`http://www.jfree.org/jfreechart/`

JCommon
Library required by the JFreeChart.
Version: 1.0.18
`http://www.jfree.org/jcommon/`

Saxon
XML processing library required by the Castor.
Version: 9.4
`http://saxon.sourceforge.net/`

Xerxes
XML processing library required by the Castor.
Version: 2.11.0
`http://xerces.apache.org/xerces2-j/`

JavaCC
Tool for grammar parser generation.
This tool is used during compilation only.
Version: 5.0.0
`http://javacc.java.net/`

JUnit
Tool and library for unit testing.
This library is used during compilation only.
`http://junit.org/`

Rscript
Tool for statistical computing and graph generation.
This tool has to be provided by the user and is not included in the project.
`http://www.r-project.org/`

11.3 Development timeline and responsibilities

This section contains development timeline and responsibility information for each project included in the SPL Tools.

11.3.1 Core project

All members of the development team participated on the Core project implementation.

Jiří Daniel

Work timeline for Jiří Daniel who left the project in the January 2013. He was responsible for the Annotation Scanner and mapping of Java classes to XML in the Core project

Month	Functionality	Additional notes
2012-05	Initial layout of the repository Java source code formatter for Eclipse Annotation representation XML mapping	 initial implementation added support, standalone application
2012-06	Annotation scanner XML mapping Unit tests for XML mapping	initial implementation fixes, improvements fixes, improvements
2012-07	Annotation scanner	fixes, improvements
2012-08	Annotation scanner	added support for usage from code
2012-09	Annotation scanner Annotation representation XML mapping	fixes, improvements fixes, improvements fixes, improvements
2012-10	Annotation scanner XML mapping Logging	fixes, improvements fixes, improvements added support
2012-11	Logging Annotation scanner XML mapping	fixes, improvements fixes, improvements fixes, improvements

František Haas

Work timeline for František Haas who was responsible for access to remote repositories, measurement execution and data storage.

Month	Functionality	Additional notes
2012-05	Prototyping	
2012-06	XML representation Execution support, SSH	design design
2012-07	Execution support, SSH	basic implementation

Month	Functionality	Additional notes
2012-08	Git, Subversion access	unauthenticated access
2012-09	Sampler creation	design
2012-10	Sampler creation Execution support, SSH Data store	basic implementation adapted to sampler basic implementation
2012-11	Execution support, SSH Sampler creation Data store Git, Subversion access File system and file utilites	rewritten to batch execution compilation, output handling refactored key authentication fixes, improvements
2012-12	Git, Subversion access Sampler creation SSH Data Store Command line interface File system and file utilites	key authentication, host verify improvements fixes, improvements fixes, improvements improved fixes, improvements
2013-01	Javadoc, documentation Integrated generators Fixes, improvements, tests INI configuration Git access Execution support, SSH	fixes, improvements added fixes, improvements fixes, improvements fallback to system git reconnect support
2013-02	Javadoc, documentation Fixes, tests, improvements	
2013-03	Javadoc, documentation Fixes, tests, improvements	

Jaroslav Kotrč

Work timeline for Jaroslav Kotrč who was responsible for SPL grammar parser and annotation representation.

Month	Functionality	Additional notes
2012-05	Parser Expander	initial implementation initial implementation
2012-06	Parser context Parser Expander	improvements improvements fixes
2012-07	Parser	improvements, fixes
2012-09	Parser	improvements
2012-10	Parser Expander	extended grammar, improvements, fixes extended grammar, improvements
2012-11	Parser context Parser	improvements improvements, fixes, updated grammar

Month	Functionality	Additional notes
	Scanner	fixes
2012-12	Parser	improvements
2013-01	Parser Expander	improvement, fixes fixes
2013-02	no work done (focused on the case study project)	
2013-03	Documentation	

Martin Lacina

Work timeline for Martin Lacina who was responsible for annotation evaluation and HTML report creation in the Core project. Most of the improvements were made for better usage from the Eclipse plug-in.

Month	Functionality	Additional notes
2012-06	Parser	improvements
2012-07	XML representation Parser	improvements improvements
2012-08	Parser Global defined aliases	improvements added support
2012-09	Annotation evaluation HTML report Graph creation Project R usage	design and prototype added support added support design and initial implementation
2012-10	Measurement unique identification XML output HTML output Scanner Graph creation Annotation information structure Annotation evaluation Execution support Sampler creation Parser	added support added support improvements improvements improvements improvements (annotation location) integrated to basic code rewritten to use data storage ab- straction added support to skip already measured measurements improvements in code templates improvements
2012-11	Annotation evaluation INI configuration support Project R usage Parser Global defined aliases Graph creation HTML output	improvements, added configuration design and initial implementation improvements improvements refactored fixes fixes
2012-12	Parser Sampler creation HTML output	improvements improvements fixes

Month	Functionality	Additional notes
	Data storage Logging Integrated generators Invoked execution	improvements improvements improvements added support
2013-01	Invoked execution Logging Graph creation Annotation evaluation HTML output Parser	improvements improvements fixes fixes, improvements rewritten with XSLT support improvements
2013-02	HTML output Documentation	improvements
2013-03	Documentation	

11.3.2 Case study project

The Case study project was lead by Jaroslav Kotrč. Following table contains its timeline. František Haas provided runtime for measurements and fixed discovered bugs in the *Framework* with Martin Lacina.

Jaroslav Kotrč

Month	Functionality	Additional notes
2012-10	Beginning	chosen project JDOM, repository briefly inspected
2012-11	Configuration Generators	created created
2012-12	Annotations	added
2013-01	Generators Annotations	improved, fixed added, commented, refactored, improved
2013-02	Annotations Documentation	measured, improved
2013-03	Annotations Documentation	measured, improved

11.3.3 Eclipse plug-in project

The Eclipse plug-in project was lead by Martin Lacina. Jaroslav Kotrč participated in the development.

Martin Lacina

Month	Functionality	Additional notes
2012-05	Prototyping and design	
2012-06	Annotations overview Source code manipulation	initial implementation initial implementation
2012-07	Project configuraiton dialog Annotations overview Annotation editor	added support improvements added support
2012-08	Xtext grammar and editors Content assist Project configuraiton dialog Annotations overview Annotation editor	added support added support fixes, improvements fixes, improvements fixes, improvements
2012-09	no work done (focused on formula evaluation in the core project)	
2012-10	Content assist Core integration fixes	fixes, improvements
2012-11	Content assist Core integration fixes	fixes, improvements
2012-12	Annotation editor Source code manipulation Project configuraiton dialog Evaluation results presentation Graph presentation Evaluation execution	fixes, improvements fixes, improvements fixes, improvements initial implementation added support added support
2013-01	Project configuraiton editor Evaluation results presentation Evaluation execution Graph presentation Annotation editor Xtext grammar and editors Integrated generators New file wizards	fixes, improvements fixes, improvements fixes, improvements added configuration dialog fixes, improvements fixes, improvements added support to content assist added support
2013-02	Evaluation results presentation Evaluation execution	fixes, improvements fixes, improvements
2013-03	Documentation Annotation editor	 fixes, improvements

Jaroslav Kotrč

Month	Functionality	Additional notes
2012-07	Project configuration dialog	added validation support, improvements, fixes
2012-08	Xtext editors Content assist	fixes refactored
2012-09	Syntax highlighting	added support
2012-10	Syntax highlighting Project configuration dialog Xtext editors	completed fixes updated grammar

Month	Functionality	Additional notes
2012-11	Xtext editors Project configuration dialog	updated grammar, refactored, improvements fixes
2012-12	Xtext editors Annotations overview Project configuration editor	fixes improvements created from Project configuration dialog, improvements, fixes
2013-01	INI configuration editor Xtext editors Project configuration editor	added support updated grammar imrovements, fixes
2013-02	no work done (focused on the case study project)	
2013-03	INI configuration editor Project configuration editor Documentation	fixes fixes

11.3.4 Hudson plug-in project

The Hudson plug-in project was lead by Jiří Daniel who created the initial implementation. Martin Lacina took over this project in the half of December 2012 due to serious illness of Jiří Daniel. Martin Lacina is the responsible person for this project.

Jiří Daniel

Month	Functionality	Additional notes
2012-11	Initial prototype	SPL execution through the command line

Martin Lacina

Month	Functionality	Additional notes
2012-12	Took over the project Rewritten SPL execution	 Java reflection
2013-01	Build step configuration SPL execution Results presentation	improvements fixes, improvements added support
2013-03	Documentation	