Appendix D

Technical About the Tools

As a part of the research method, we implemented the algorithms that we made in order to evaluate them in realistic settings. The Software Product Line Covering Array Tool (SPLCA-Tool) implements various algorithms for working with feature models and covering arrays. It includes various libraries from the community; thus, it integrates with formats and algorithms in the research community and from industry.

D.1 Version History

Throughout the PhD project, the tool was updated to include our latest developed and published results. Each release of the tools was named after the conference its implemented results were published at. Here is the release history of the tools:

- **SPLCATool v0.1 (EuroPLoP 2011)** - Released 2011-06-14 as resource material for a paper presented at EuroPLoP 2011 (Paper 3).
- **SPLCATool v0.2 (MODELS 2011)** - Released 2011-05-03 as resource material for paper submitted to MODELS 2011 (Paper 1).
- **SPLCATool v0.3 (SPLC 2012)** - Released 2011-10-28 as resource material for a paper published at SPLC 2012 (Paper 2).
- **SPLCATool v0.4 (MODELS 2012)** - Released 2012-04-03 as resource material for paper submitted to MODELS 2012 (Paper 4).
- **Automatic CIT Tool Collection (ICTSS 2012)** - Released 2012-06-18 as resource material for a paper submitted to ICTSS 2012 (Paper 5).

The tool provided us with a framework onto which new functionality could be added onto and experimented with. It also was used to produce the empirical evaluations in the papers published.

D.2 Architecture

The tool works with three kinds of artifacts given to it by the user along with some instructions of what to do with them.

- **Feature Models:** In essence, any feature model that can be converted to a propositional formula can be loaded by the tool. We have, however, only implemented some of the major formats for feature modeling used in academia and industry. Whenever possible, the libraries associated with a particular format have been used. When this was not possible, custom loaders were developed. When the libraries contained bugs that stopped us, those bugs were fixed.

- **Covering Arrays:** A covering array is a set (or an array) of configurations of a feature model selected such that each t-sets, for some t, are covered (thereby the name *covering array*). Each configuration is a set of assignments such that all features are given an assignment and such that the configuration is valid according to the feature model.

- **Weighted Sub-Product Line Models:** Similarly as covering arrays, sub-product line models give assignments to features, but assignments can be left unassigned. The unassigned features thus maintain some variability, and the configurations of this partial feature model are a sub-set of all possible configurations. Therefore, they are called sub-product line models. The models are called weighted because each sub-product line model is given a positive integer as a weight.

The first set of functionalities provided by the tool is verification of the artifacts.

- **Feature models** have two basic requirements: (1) They must have valid syntax, and (2) they must have at least one valid configuration. Both of these are checked by running the `is_satisfiable` task.

- **Covering arrays** are valid and invalid relative to a feature model. Because a covering array is a set of configurations of the feature model, they must, to be valid, be valid configurations of the feature model. This is verified using the `verify_solutions` task.

- **Weighted sub-product line models** are also valid or invalid relative to a feature model. Because they are not configurations but sub-product lines, they have the same requirement as feature models: They must give at least one valid configuration. This is checked by running the `verify_weighted_solutions` task. In addition, the weights must be positive integers.

The second set of functionalities is the various algorithms:

- **Generating covering arrays:** A covering array is produced from a feature model and a strength t for the strength of the coverage. Various algorithms can be invoked for generating covering arrays. The algorithms all have different characteristics of (1) speed of generation, (2) size of resulting covering array, (3) variance in results between invocations and (4) supported additional features.

- **Calculating the coverage of a set of products:** Given a feature model and a set of configuration and a strength t, the t-wise coverage of the products given the feature model can be calculated. The coverage can be given either as a t-set coverage in percent of t-sets

covered by the configurations or as weight coverage in percent of the weight covered by the configurations. To calculate weight coverage, a weighted sub-product line model is also required.

- **Generate a list of suggestions for improving the weight coverage of a set of products:** Given a feature model, a set of products and a weighted sub-product line model, the set of products might be improved by changing it slightly. Suggesting such small changes to improve coverage is the role of this class of functionality.

D.2.1 Feature Model Formats

The following five formats are supported.

- **GUI DSL Models (.m)** This is a feature model format used by the Feature IDE tools suite [155]. It is as a format by the AHEAD and GenVoca tools by Automated Software Design Research Group at the University of Texas at Austin.

- **Simple XML Feature Model (SXFM) (.xml)** SXFM is a feature model format used to store feature models for the Software Product Line Online Tools (SPLOT)[1] developed by the Generative Software Development Lab / Computer Systems Group, University Of Waterloo, Canada, 2010.

- **DIMACS CNF format (.dimacs)** This is the format used by the SAT4J library. It has become a standard format for Boolean formulas in CNF. It was initially proposed as a format for CNF formulas by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) [48].

- **CVL 1.x (.xmi or .cvl)** CVL is a variability-modeling format proposed in 2008 for adding variability mechanisms to DSLs [69]. It is currently being considered as a standard for variability-modeling by the OMG [71]. The version 2 of CVL is currently being developed. However, only the first version of the format is supported by SPLCATool. CVL models can be created using the CVL Tool[2].

  Due to the richness of CVL models, a feature model must be extracted from it before it can be loaded. Download the CVL Tool chain from the ICTSS 2012-paper resource page[3], unzip `CVLUtils.jar` and execute `java -cp CVLUtils;CVLUtils/lib/* CVL2GDSL model.cvl` to produce a Feature IDE GUI DSL file `model.cvl.m`. It can be loaded with the normal loading procedure.

D.2.2 Configuration Set Formats

Covering Arrays are a set of configurations of a feature model. Thus, except for their special property, they are no different from a set of configurations.

[1] `http://gdansk.uwaterloo.ca:8088/SPLOT/sxfm.html`, retrieved 2013-01-05

[2] `http://www.omgwiki.org/variability/doku.php/doku.php?id=cvl_tool_from_sintef`

[3] `http://heim.ifi.uio.no/martifag/ictss2012/`

- **Comma Separated Values (CSV):** Due to the use of the comma as a decimal mark in Norway, Norwegian CSV files are semi-colon separated instead of comma-separated.

  Configurations are stored in the following way, textually: The first entry in the file must be "Feature\Product;". This means that the features will be listed in the rows, and the products will be listed in the columns. This is the most practical arrangement for editing and viewing. Following this entry, the names of the products are listed. This can be as simple as 1, 2, 3 etc., written as "1;2;3;". On each line following the first line is a feature name followed by either an "X" or a "-" when the feature is included or excluded, respectively, for the product in that column. Editing these files in Excel or similar is of course recommended.

- **Excel Office Open XML (XSLX):** These files are edited using Excel or similar. They are loaded by the tool using the "Apache POI - Java API To Access Microsoft Format Files"-library. There are some additional requirements when using this format in order to delimit the product configurations and allow for additional comments and entries in the file: (1) the entry following the lower-most feature must be "#end" and (2) the entry following the last product name must also be "#end".

D.2.3 Weighted Sub-Product Line Formats

Weighted Sub-Product Line models are a set of configurations of a feature model with two additional things added. Thus, except for their two additions, they are stored similarly to covering arrays.

The two additions are: (1) Assignments can be left unassigned using a question mark. (2) The feature on the first row is named "#Weight". On this row, the weight for each sub-product line is given. It must be a positive, non-zero integer.

D.2.4 Versions

There are several versions and editions of the tool. Each version is named after the conference at which the results it implements are published. For one version, there are several editions because of licensing concerns.

- **v0.1 (EuroPLoP 2011):** This version introduced basic covering array generation for $1 \leq t \leq 3$ and implements the Bow-Tie reduction algorithm. The covering arrays were stored in the CSV format.

- **v0.2 (MODELS 2011):** This version introduced 4-wise covering array generation, SAT-timing, t-wise coverage calculation and the verification of the solutions in a covering array.

- **v0.3 (SPLC 2012):** This version of the tool introduced an implementation of the ICPL algorithm for $1 \leq t \leq 3$. In addition, it implements support for running the IPOG, CASA and MoSo-PoLiTe algorithms; however, these are only available in the Full edition of the tool. It cannot be freely distributed because of licensing concerns. However, an edition with the adaptors only (The Adaptors Only Edition) is available into which the three tools

can be added to produce the full edition. The IPOG algorithms supports covering array generation for $1 \leq t \leq 6$, CASA for $1 \leq t \leq 6$ and MoSo-PoLiTe for $2 \leq t \leq 3$.

- **v0.4 (MODELS 2012):** This version of the tool introduced implementations of algorithms related to weighted sub-product line models. In addition, a GUI was introduced to make working with the models easier; the output is still written to the command line however. The command-line interface is still available by extracting `SPLCATool-v0.4-MODELS2012.jar` and calling `java -cp SPLCATool-v0.4-MODELS2012; SPLCATool-v0.4-MODELS2012 no.sintef.ict.splcatool.SPLCATool`.

  The tool introduced (1) the generation of covering arrays prioritized using the weighted models, (2) the verification of weighted sub-product line models, and (3) the improvement of existing covering arrays using the weighted sub-product line models. This version also introduced the loading of covering arrays in the XLSX format.

- **Automatic CIT Tool Collection (ICTSS 2012):** The Automatic CIT Tool introduced scripts, formats and the implementation of the algorithms for automatic CIT both for Eclipse- and CVL-based product lines. It also introduced a support for extracting GUI DSL models from CVL-models, thereby enabling the generation of covering arrays from feature models within CVL models.

D.2.5 Overview of Algorithms and Object Structures

Inside the tool, the different formats are imported and converted into other formats in order to be applicable for various algorithms. Figure D.1 shows an overview of the transformations, what formats are fed to the different algorithms and what these algorithms produce. The figure is of no particular modeling notation. First of all, feature models can be given to the tool in the four different formats. They are converted in one direction. After being converted, some of the formats have exporting to file implemented. Covering arrays are read in from CSV and XLSX files. Weighed sub-product line models can also be read in from CSV and XLSX files. The rectangular boxes symbolize internal object-structures; the arrows symbolize algorithms that generate some other data.

The various algorithms available are further described in the basic user manual, Section D.3.

D.3 User Manual

This basic user manual explains how to use the tool using the Eclipse case study as a running example. The four files for the Eclipse case study are available on the resource page of the MODELS 2012-paper[4].

- Products — `eclipse-red.m.actual.csv`
- Feature model — `Eclipse.m`
- Weights — `eclipse-red.m.actual-weighted.csv`
- Ordering — `eclipse.m.order.csv`

[4] `http://heim.ifi.uio.no/martifag/models2012/`

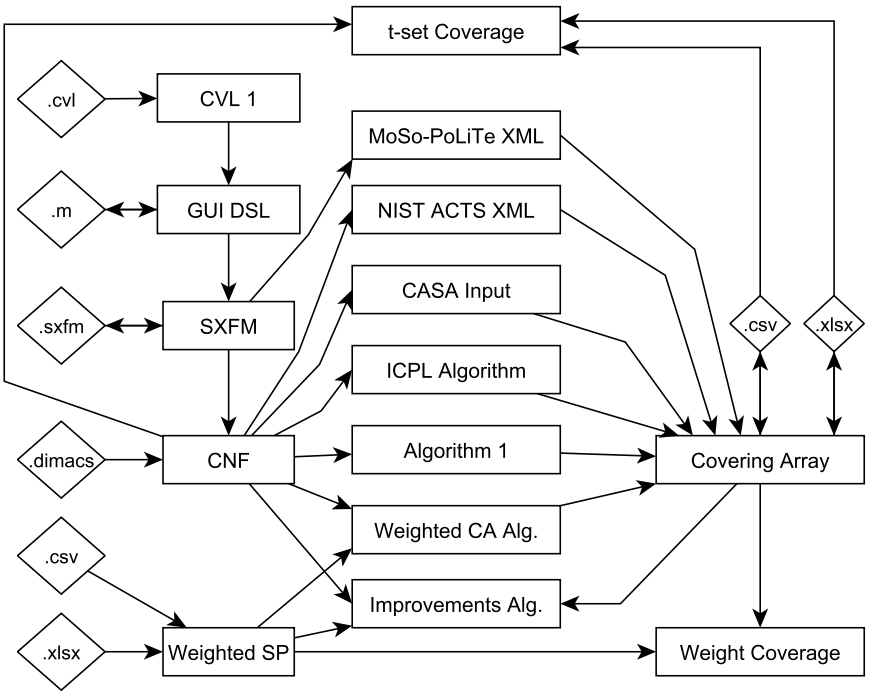

Figure D.1: Transformations in the Tool

The products are the configurations of the 12 products that were available for download on the Eclipse v3.7.0 (Indigo) web site. They are valid configurations of the feature model that captures the relations between the different features of the Eclipse product line. The weights-file contains the current market-situation—it was captured by adding the number of downloads to each product offered. This is a basic approximation because we have no clue as to how the users proceeded to configure their products. The ordering files contain the order the features should be listed in the generated files. This is necessary because there is no intrinsic order to the features in the other files.

Throughout the manual, the command-line will be used. Only the arguments of the command-line invocations will be shown.

D.3.1 Analysis

First, let us check the validity of the files and find out some basic information about them. Let us first check the feature model. The `sat_time` task (v0.4) will attempt to load the feature model; if successful, it will print some basic information about it and try to find a single valid configuration. It will then report whether one exists and how long time it took to find it, its satisfiability-time. Execute `-t sat_time -fm Eclipse.m` (v0.4) to run the task. This information is returned:

- Valid feature model?: yes
- Number of Features: 26
- Number of CNF-clauses of the composed constraint: 6
- Satisfiable?: yes
- Satisfying time: 0.446 ms

Thus, we know several important things: The feature model is valid and it has a very fast satisfiability time. This means it is fully usable for all other tasks.

Let us try to find one additional piece of information, the number of valid configurations. Run `-t count_solutions -fm Eclipse.m` (v0.4). This task does not scale very well; up to about 200 features should work on modern hardware.

- Number of possible configurations: 1,900,544.

This is an enormous number; $log_2(1,900,544) \approx 21$, which is relatively close to 26, the number of features of the feature model. If each feature is a yes-or-no choice, and each choice was independent, then the number of configurations would be 2^f, where f is the number of features. Features normally are not independent, but 2^f still gives us an idea of the magnitude of the number of configurations.

Let us now analyze the products. Run `-t verify_solutions -fm Eclipse.m -check eclipse-red.m.actual.csv` (v0.4). This task will check whether the configurations are valid according to the feature model. It tells us that it is. Had it been invalid, the tool would output the CNF-clauses that were not satisfied. This gives us a good guide as to where to look for the problem.

Let us analyze the weighted sub-product line model. Run `-t verify_weighted_solutions -fm Eclipse.m -check eclipse-red.m.actual-weighted.csv` (v0.4). This task will check whether the partial configurations are valid. It tells us that it is. Again, had it been invalid, it would tell us which CNF-clauses were not satisfied.

- Configurations valid?: yes
- Weighted sub-product line models valid?: yes

Finally, let us find the t-set and weight coverage of the products. Run `-t calc_cov -fm Eclipse.m -s <t> -ca eclipse-red.m.actual.csv` (v0.4) to find the t-set coverage and `-t calc_cov_weighted -fm Eclipse.m -s <t> -ca eclipse-red.m.actual.csv -weights eclipse-red.m.actual-weighted.csv` (v0.4) to find the weight coverage, both for $1 \leq t \leq 3$. That gives the following coverages:

- 1-set coverage: $49/50 \approx 98.0\%$
- 2-set coverage: $929/1,188 \approx 78.2\%$
- 3-set coverage: $10,034/17,877 \approx 56.1\%$
- 1-wise weight coverage: $52,949,676/52,949,676 = 100\%$
- 2-wise weight coverage: $661,870,950/661,870,950 = 100\%$
- 3-wise weight coverage: $2,623,612,957/2,623,612,957 = 100\%$

It is not a surprise that the weight coverage is 100%. It is because the weighted sub-product line model contains only the products that are offered. It would be better to have actual information about the market, but such information is unavailable, unfortunately. The TOMRA models are unavailable due to sensitivity concerns.

The denominators of the t-set coverage are 50, 1,188 and 17,877. These are the number of valid 1, 2 and 3-sets, respectively. When all are covered, we have a complete t-wise covering array.

The denominators of the weight coverage are 52,949,676, 661,870,950 and 2,623,612,957. These are the total weight for all valid 1, 2 and 3-sets, respectively. When all the weight is covered, the weighted covering array has a coverage of 100%.

D.3.2 Generating Covering Arrays

Now that we have analyzed all our models and checked their validity, let us generate some new things and then verify them.

Generate a t-wise covering array for $1 \leq t \leq 3$ using ICPL by running `-t t_wise -a ICPL -fm Eclipse.m -s <t>` (v0.3 (ICPL Edition)). This results in the following files containing the covering arrays.

- Eclipse.m.ca1.csv, products: 2, generation time: 26 ms
- Eclipse.m.ca2.csv, products: 12, generation time: 136 ms
- Eclipse.m.ca3.csv, products: 39, generation time: 663 ms

Calculating the coverage of each using the same method as discussed above yields 100% 1, 2 and 3-wise coverage, respectively, as expected. Notice that the 12 products offered by Eclipse have a 2-wise coverage of 78.2% while 12 products are also needed to yield 100% 2-wise coverage!

It is also possible to calculate covering arrays using other algorithms. Because of licensing issues, we cannot distribute the full edition of the tool. However, once these tools have been acquired, the following commands can be used to calculate covering arrays using other algorithms: `-t t_wise -a <a> -fm Eclipse.m -s <t>` (v0.3 (Full Edition)) where `a` is `IPOG`, `CASA` or `MoSoPoLiTe`. These three support $1 \leq t \leq 6$, $1 \leq t \leq 6$ and $2 \leq t \leq 3$, respectively.

If 12 products, a complete 2-wise covering array, are too costly for testing in your context, and you would like to get a smaller set that is still related to the market situation, you can use weighted covering array generation. Let us say we decide to cover 95% of the weight. Run `-t t_wise_weighted -a ChvatalWeighted -fm Eclipse.m -s <t> -weights eclipse-red.m.actual-weighted.csv -limit 95%` (v0.4) for $2 \leq t \leq 3$ to produce the following results:

- Eclipse.m.ca2.csv, products: 4, generation time: 341 ms
- Eclipse.m.ca3.csv, products: 3, generation time: 260 ms

Calculating the weighted coverage using the methods discussed above yields 97.05% and 95.08% 2 and 3-wise weight coverage, respectively. Both are, as required, equal to or above 95%.

D.3.3 Extending Towards a Covering Array

A realistic situation when doing product line testing is that certain popular and important products must be tested. For example, for Eclipse, 3 products account for 88% of the downloaded products. Instead of just adding these 3 products to the already complete covering arrays, we could start from them to probably make them be a part of the covering array.

Extract the "Java", "Java EE" and "Classic" products into a file called `eclipse-red.m.important.csv`. To include them in a covering array run `-t t_wise -a ICPL -fm Eclipse.m -s <t> -startFrom eclipse-red.m.important.csv` (v0.3 (ICPL Edition)). The extended versions are:

- Eclipse.m.ca1.csv, products: 5, time: 27 milliseconds
- Eclipse.m.ca2.csv, products: 13, time: 120 milliseconds
- Eclipse.m.ca3.csv, products: 40, time: 507 milliseconds

For 1-wise testing, this does not reduce the effort because there were three important products, but only two were required to achieve 1-wise coverage. For 2-wise testing, the new covering array has 13, only one more than a fresh 2-wise covering array. For 3-wise testing, the new covering array has 40 products, also only one more than the 39 required when building a fresh covering array.

We can also extend covering arrays based on weight. If we start with the three important products again, let us see what is needed to extend with one more product. We could also extend based on percentages, but this opportunity is used to demo the size limits.

First of all, the weight coverages of the 3 important products are 98.68%, 97.05% and 95.08%, respectively. Run the following command to extend the set of products with a maximum of two new product based on maximizing the weight coverage starting from three important products for $1 \leq t \leq 3$: `-t t_wise_weighted -a ChvatalWeighted -fm Eclipse.m -s <t> -weights eclipse-red.m.actual-weighted.csv -startfrom eclipse-red.m.important.csv -sizelimit 5` (v0.4). The new set of products of sizes 4, 5 and 5 has weight coverages of 100%, 99.5% and 98.5%, respectively.

D.3.4 Improving a Partial Covering Array

In settings where a complete test system has been set up; for example, if the products of a hardware product line has been physically set up, it is not good to have drastic changes to the product configurations.

The tool supports suggesting small changes to an existing set of products to improve its weight coverage.

An especially important usage of this is in the following situation: A product line gets a new feature added to it. The company already has a test lab of, say, 12 products set up. The question now is: To which product is it best to add this feature to maximize the coverage with respect to the market situation (the weight coverage)?

First add a new feature to `Eclipse.m` named `X` as an optional feature of `RCP_Platform`, then add a new row to the set of product offered by Eclipse with X set to all excluded ('-'), then finally, add a new row to the weighted sub-product line model with all unassigned ('?'). This row of unassigneds tells the tool, in effect, that we do not know who will be using the new feature, so we had better ensure that it is exercised during testing and that it is tested out together with the other features.

To have the tool figure this out automatically, run `-t improve_weighted -fm Eclipse.m -s 2 -ca eclipse-red.m.actual.csv -weights`

`eclipse-red.m.actual-weighted.csv -search 1 > list.txt` (v0.4). Here, we assume that the offered products are also the products that the Eclipse Projects tests, a realistic assumption. We want to maximize our 2-wise interaction coverage, which means that we want our new feature to be exercised against any other feature. Finally, we want to maximize the weight coverage to ensure market relevance. Because we only want to activate a single feature, we only search for single changes, `-search 1`. Running the task results in the following results:

- Original weight coverage: 96.3%
- Suggested change: Set feature X to included for the product named "1" (the second product).
- New weight coverage: 99.3%.
- Time taken: about 2 seconds.

D.3.5 Automatic CIT: Eclipse-Based Product Lines

This part of the manual explains how to apply Automatic CIT to Eclipse-based product lines. The first thing that is needed is to set up the directory structure where all files will go. First, decide on a root directory in which all files related to the testing will be stored. In this directory, create these directories:

- `models` — contains all input to the testing tools.
- `packages` — contains all packages to be used to construct the software to be tested.
- `products` — will store all built and tested products and their associated workspaces.
- `results` — will store all generated results.
- `scripts` — contains all scripts used for testing.
- `tools` — contains all tools used for testing.
- `workspace` — workspace for the Eclipse tool used to build the package repository.

The next step is to acquire the required tools. Links are available from the ICTSS 2012-paper resource page[5]. Note that all downloads mentioned in this part of the manual is found on that resource page. Download and extract the following free tools in the "tools" directory.

- 7-Zip Command Line Version (9.20)
- Eclipse Platform (3.7.0)
- UnxUtils (of 2007-03-01), ports of the most important GNU utilities to Windows
- An implementation of Algorithm 3 from the paper

The next step is to build the package repository. Download the mirroring-script, and place it in the scripts directory. Download and edit the environment set up script in which the path to the java.exe, cmd.exe and the root directory must be specified. Executing the mirroring script will construct mirrors for the Eclipse Indigo repositories in "%basedir%/packages/repos/". This will take a while and might require rerunning the script several times to ensure that all packages

[5] Appendix A or `http://heim.ifi.uio.no/martifag/ictss2012/`

were downloaded correctly. The size of these repositories during the experiments presented in the paper was 3.6 GiB.

Place a copy of the package containing the Eclipse Platform in "packages". Download the Eclipse Automated Tests for Eclipse 3.7.0. Inside it there is a zip-file called `eclipse-testing/eclipse-junit-tests-I20110613-1736.zip`. Extract it in the "packages" directory.

The next step is setting up the required models. The technique requires three files to be built: (1) The feature model, for example such as the one for the Eclipse IDEs. (2) The mapping between features and code artifacts, as for example the one used in the experiment in the paper. (3) The mapping between features and tests, as for example the one used in the experiment in the paper. Examples of these files for the Eclipse 3.7.0 system are available on the resource page.

The next step is execution. The following command will execute Automatic CIT: `java no.sintef.ict.splcatool.lasplt.LTSPLT <basedir> <t> <timeout>`.

`basedir` is the root directory of the testing system, t is the strength of the combinatorial interaction testing and timeout is the time in seconds after which a test suite execution will be aborted.

When the feature model, the features themselves or the test suites are changed, there is no additionally required manual work to redo the testing other than re-executing the above command.

Output such as the following will be produced on the command line:

```
Converting Package Mapping File to CSV [done]
Converting Feature-Test Mapping File to CSV [done]
Loading Mapping Files [done]
Generating 2-wise Covering Array [done]
Loading Covering Array [done] Products: 14
Test Product 0:
  Installing base product: [done]
  Installing features:
    Installing feature EclipseIDE [nothing to install]
    Installing feature RCP_Platform [nothing to install]
  Installing tests:
    Installing tests for EclipseIDE
    Installing tests for RCP_Platform
      Installing test org.eclipse.test.feature.group [done]
      ...
      Installing test org.eclipse.search.tests [done]
  Running tests:
    Running tests for EclipseIDE
    Running tests for RCP_Platform
      Running test org.eclipse.ant.tests.core.AutomatedSuite [done] 100%
          (85/85)
      Running test org.eclipse.compare.tests.AllTests [done] 89% (101/113)
      Running test org.eclipse.core.internal.expressions.tests.AllTests [
          done] 100% (108/108)
      ...
      Running test org.eclipse.search.tests.AllSearchTests [already exists]
           78% (29/37)
```

```
Test Product 1:
  Installing base product: [already exists]
  Installing features:
    Installing feature WindowBuilder [done]
    Installing feature SVN15 [done]
    Installing feature EclipseIDE [nothing to install]
    ...
    Installing feature RCP_Platform [nothing to install]
    Installing feature Datatools [done]
    Installing feature SVN [nothing to install]
...
Test Product 13:
...
```

As a final stage, it is possible to generate a web-view for the results. This is good for using Automatic CIT in conjunction with continuous integration system such as Hudson[6] or Jenkins[7].

The command in the previous step will produce logs containing the contents of *stdout* and *stderr* during test execution called `failure-<product nr>-<test class name>.log` and test result details in files called `product<product nr>-<test class name>-test-result.xml`. The results produced by the experiment reported in the paper are available on the resource page of Paper 5.

These results can be compiled into a report by executing the following command: `java no.sintef.ict.splcatool.lasplt.GenerateWebView <basedir>` This will produce an HTML-document. An example for the Eclipse IDE experiment is found on the resource page of Paper 5.

D.3.6 Automatic CIT: CVL Based Product Lines

Automatic CIT can be applied to CVL-based product lines in a similar way to Eclipse-based product lines. The set up required is found in `CVLLASPLTInput.java` a template found in CVL tool chain available from the resource page website. The guidelines for Eclipse-based product lines can be used to understand how to run Automatic CIT for CVL-based product lines.

[6] `http://hudson-ci.org/`
[7] `http://jenkins-ci.org/`