****

Project name: 15010 REVaMP²

REVaMP²

**R**ound-trip **E**ngineering and **Va**riability **M**anagement **P**latform and **P**rocess

D7.5 REVaMP² tool-chain Users Guide

|  |  |  |  |
| --- | --- | --- | --- |
| Deliverable | D7.5 REVaMP² tool-chain Users Guide | | |
| Confidentiality | Confidential | Type | Software & Video |
| Project | REVaMP² | Date | 2019/11/14 |
| Status | Final | Version | 02 |
| File identifier | D7.5.docx | | |

|  |  |
| --- | --- |
| Editors, contact persons | Michael Benkel, ScopeSET |
| E-mail | benkel@scopeset.de |

AUTHORS

|  |  |
| --- | --- |
| **Name** | **Affiliation** |
| Luis Pérez | The Reuse Company, Spain |
| Michael Benkel | ScopeSET, Germany |
| Felix Suda | ScopeSET, Germany |
| Sebastian Reiter | Forschungszentrum Informatik (FZI), Germany |
| Anton Paule | Forschungszentrum Informatik (FZI), Germany |
| Jan-Philipp Steghöfer | University of Gothenburg, Sweden |
| Christian Kröher | University of Hildesheim, Germany |
| Klaus Schmid | University of Hildesheim, Germany |
| Sascha El-Sharkawy | University of Hildesheim, Germany |
| Ana C. Marcén | University San Jorge, Spain |
| Jaime Font | University San Jorge, Spain |
| Tewfik Ziadi | Sorbonne University, France |
| Anas Shatnawi | Sorbonne University, France |
| Xhevahire Tërnava | Sorbonne University, France |
| Saura Jyoti Dhar | Robert Bosch GmbH, Germany |
| Slawomir Duszynski | Robert Bosch GmbH, Germany |
| Andreas Burger | ABB AG, Germany |
| Alessandro Cornaglia | Forschungszentrum Informatik (FZI), Germany |

CHANGE HISTORY

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Reason for change | Sections / Pages Affected |
| 01 | 14 November 2019 |  | First version |
| 02 | 12.December 2019 |  | Final version |

Executive summary

This document is a brief Users Guide of the available tool-chains and the supported workflows.

For each tool the following information is described

* Brief description of the tool
* where to get it
* license information (where to get it)
* short guidelines on installation
* short guidelines on how to use it (or link to an external documentation)
* list of commands that can be used from the command line (if any)
* example data
* any relevant links

Table of Contents

[Executive summary 3](#_Toc27065609)

[Table of Contents 4](#_Toc27065610)

[Acronyms 5](#_Toc27065611)

[1. Overview 6](#_Toc27065612)

[2. Tool Guides 7](#_Toc27065613)

[2.1. Feature Dependency Visualization (FeDeV) 7](#_Toc27065614)

[2.2. KernelHaven 9](#_Toc27065615)

[2.3. Variability Extraction and Analysis (VEXA) 12](#_Toc27065616)

[2.4. Feature Location in Models by an Evolutionary Algorithm (FLiMEA) 16](#_Toc27065617)

[2.5. Commit Analysis Infrastructure (ComAnI) 18](#_Toc27065618)

[2.6. Problem-Solution-Space Co-Evolution Support (PSS-CE) 20](#_Toc27065619)

[2.7. Product-Line-Product-Variant Co-Evolution Support (PLPV-CE) 22](#_Toc27065620)

[2.8. FINALIsT² 26](#_Toc27065621)

[2.9. pure::variants 27](#_Toc27065622)

[2.10. Jittac 28](#_Toc27065623)

[2.11. Systems Engineering Suite (SES) 30](#_Toc27065624)

[2.12. SIMULTime 30](#_Toc27065625)

[2.13. DragonflyME 35](#_Toc27065626)

[3. Predefined Workflows 38](#_Toc27065627)

[3.1. Feature Annotation Extraction and Visualization Workflow 38](#_Toc27065628)

[3.2. Extraction and Variability Management Workflow 39](#_Toc27065629)

[3.3. Constraint Extraction Workflow 40](#_Toc27065630)

[3.4. Feature Dependency Visualization and Traceability Workflow 42](#_Toc27065631)

[3.5. Identify and Inspect Feature Locations Workflow 45](#_Toc27065632)

[3.6. Analyse Models and Extract Features Workflow 46](#_Toc27065633)

[4. Toolsets 48](#_Toc27065634)

[4.1. PL Co-Evolution Toolsets 48](#_Toc27065635)

[4.2. PL Verification Toolsets 52](#_Toc27065636)

Acronyms

|  |  |
| --- | --- |
| ADL | Architecture Description Language |
| API | Application Program Interface |
| AST | Abstract Syntax Tree |
| BUT4Reuse | Bottom-Up Technologies for Reuse |
| CSV | Comma-Separated Values, file format |
| DSL | Domain Specific Language |
| EAST-ADL | An Architecture Description Language |
| FeDeV | Feature Dependency Visualization |
| FLiMEA | Feature Location in Models by an Evolutionary Algorithm |
| FM | Feature Model |
| Jittac | Just-In-Time Tool for Architecture Consistency |
| LoC | Lines of Code |
| MOF | Meta-Object Facility |
| PL | Product Line |
| PSS Mapper | Problem-Solution-Space Mapper |
| RQS | Requirements Quality Suite |
| SES | Systems Engineering Suite |
| SIS | Software Intensive Systems |
| SPL | Software Product Line |
| SPLE | Software Product Line Engineering |
| SysML | Systems Modeling Language |
| UML | Unified Modeling Language |
| VEL | Variability Exchange Language |
| VEXA | Variability Extraction and Analysis |

1. Overview

Based on the individual tool development that is done in work packages 4, 5 and 6, work package 7 provides integrated tool-chains to support the workflows requested by the use cases.

This document provides brief Users Guide of the available tool-chains and the supported workflows. Chapter 2 provides a list of all REVaMP tools including a brief description, some installation guidelines and settings, which can be used for the tool.

Chapter 3 gives an overview of predefined workflows and the settings needed for the involved tools.

1. Tool Guides

This section describes the proof-of-concept tools of the REVaMP2 tool-chains. For each tool, a brief description of the baseline functionality is provided as well as information on how to get it and how to install and how to set it up.

* 1. Feature Dependency Visualization (FeDeV)
     1. Tool Description

The Feature Dependency Visualization (FeDeV) tool is a visualization application for analysis results based on the TomSawyer Visualization framework. The tool covers tabular lists for all data elements, some tree structures and interactive graphs to inspect the analysis results. The main focus of the tool is the visualization of dependencies of features, feature annotations, and variation points. FeDeV is able to import, inspect and stepwise explore the analysis results.

Each analysis source is mapped to an internal meta-model and instantiated during an import process. Several views like trees, tables, and graphs are provided to visualize the imported data. Features and feature dependencies are listed in tables while trees are used to represent variation points within files, or showing a directory hierarchy. Dependencies of features are explored step-by-step. Starting with an empty graph, identified features can be added via table selection or a search function and composed to a feature-dependency-graph.

* + 1. Source of Supply

The FeDeV tool can be downloaded from the ScopeSET website: <https://nextcloud.scopeforge.de:444/s/m4RWSPB55pXKdPT>

* + 1. Installation and License Information

**Prerequisite**

* FeDeV requires Java 1.8
* for the the VEXA integration, a running database Neo4j (Neo4j 3.5 or other version compatible with VEXA) is needed
* For the visualization an internet connection is required, to obtain a license for the Tom Sawyer framework
* FeDeV uses a port to communicate with other tools (e.g. Eclipse Capra). The default port is 4500.

**License information**

There is a deployment license required from Tom Sawyer. This license is provided by ScopeSET and comes with the FeDeV build. The license from Tom Sawyer is obtained at the first start of the FeDeV application and has a one week grace period, meaning, that the license is valid for one week, then license must be revalidated. For license revalidation, the internet connection is required, again.

**Installation**

Download the latest FeDeV build from scopeforge (link see below) and extract the zip archive. To launch FeDeV, navigate into the extracted directory and run *eclipse.exe*.

* + 1. Getting started

FeDeV is able to process analysis results produced by KernelHaven and VEXA. A KernelHaven analysis is focused on software features and their dependencies. VEXA in contrast offers tools to explore a code base, its variation points and calculate various metrics on the analyzed data.

To inspect a KernelHaven analysis, first run KernelHaven with the configuration, which produces a sqlite file containing the analysis results. Then start FeDeV and choose in the *File* menu the command *Load KernelHaven Analysis*. In the opening file selection dialog, select the sqlite file and enter two excel files to store additional notes and feature selections. It is possible to use empty excel files. Confirm the values by clicking *Ok*. Next, FeDeV will load the data and open the exploration views.

Opening the VEXA views works similar like loading a KernelHaven analysis database. First, the user has to make sure, that Neo4j is running. If Neo4j doesn’t use the default configuration or authentication is enabled, the user has to enter the bolt URL as well as username and password in the FeDeV settings dialog. It is not required, that a VEXA project (or analysis) is performed before opening the VEXA views. FeDeV offers a view, where the user can enter cypher queries and so initialize a VEXA analysis. To open the VEXA views and load any available data, use the *Load VEXA Analysis* from the *File* menu.

* + 1. Settings and configurable properties

Below a list of supported command line arguments and parameters. The parameters can be passed as a key value sequence after an additional -vmargs.

* **-Dfedev.server.graphql.port=<port>** port of the GraphQL endpoint (default is 4500)
* **-Dproxy.connection\_check.url=<url>** URL to check the proxy connection settings (default is https://www.scopeset.de
* **-Dproxy.[url|user|port|pw]=<value>** proxy connection settings (no default values, overwrites values in Settings Dialog)

Example, start FeDeV and set the GraphQL port to 4242

<FeDeV-Install-Path>\eclipse.exe –vmargs -Dfedev.server.graphql.port=4242

* + 1. Example data

FeDeV processes data created by KernelHaven and VEXA. Therefore, the example data are provided by the analysis tools. The Linux kernel source code has been used as example data for KernelHaven and the ERIKA3 code base as example for the VEXA tool.

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | <https://nextcloud.scopeforge.de:444/s/m4RWSPB55pXKdPT> |
| Download Link | <https://nextcloud.scopeforge.de:444/s/m4RWSPB55pXKdPT> |
| Source Code Repository | N/A |
| Video Tutorial | <https://youtu.be/En4QTdDE35w> (FeDeV and KernelHaven)  <https://youtu.be/gprB-bBy3IM> (FeDeV and VEXA) |
| Tutorial Slides | <http://www.revamp2-project.eu/images/Tool-chain/tutorial-slides/D4.4-Tool-Presentation-FeDeV.pdf> |
| Manual | https://nextcloud.scopeforge.de:444/s/GfgRXaKjpLzkyWY |
| Contact | Michael Benkel <[benkel@scopeset.de](mailto:benkel@scopeset.de)>  Felix Suda <[felix.suda@scopeset.de](mailto:felix.suda@scopeset.de)> |

* 1. KernelHaven
     1. Tool Description

**KernelHaven** is a flexible framework that allows you to conduct various types of analyses on software product line assets. For this purpose, KernelHaven can extract variability information from various asset types. The core components of KernelHaven are three extraction pipelines, the data processing, and a pipeline configurator as illustrated in Figure 1.

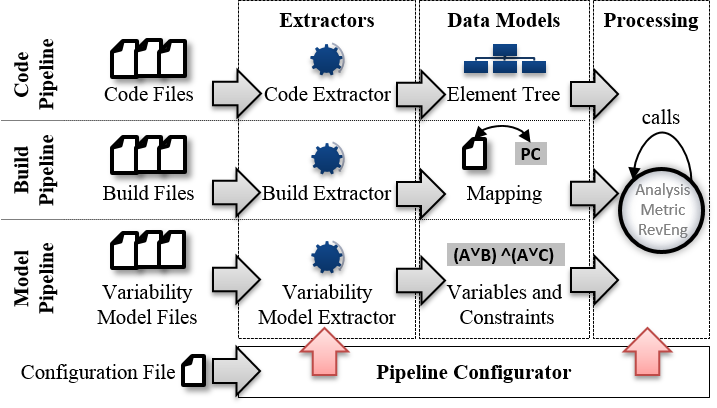


Figure 1: KernelHaven architecture.

The following information can be extracted using the tool:

*Variability Models* (variability model pipeline, lower part in Figure 1).

* The logical conditions from variability models, *e.g.,* Kconfig.
* Structural information from variability models, *e.g.,* nesting structures, type information, information about features that are linked together via constraints, …
* The range definitions of non-Boolean variables from CSV/Excel files.

*Variability on Code* (code pipeline, upper part in Figure 1)

* The variability conditions from C and C++ code, *i.e.,* #ifdef statements, from \*.c, \*.cpp, \*.S, and \*.h files.
* A variability-aware abstract syntax tree from C-Code. This may also include a macro expansion, if this is required by subsequent analyses.
* The conditions from CSV/Excel files. This serves mainly as an import interface to other systems.

*Build Variability* (build model pipeline, middle part in Figure 1)

* The variability information for the conditional inclusion of code files from makefiles.
* The variability information for the conditional inclusion of code files from CSV and Excel files. This serves as an import interface to other systems.

KernelHaven provides capabilities to conduct various analyses. Among others these are:

* The extracted variability information from build and code assets to generate (local) presence conditions and finally form (global) feature effect constraints. These feature effect constraints can be used to build a variability/feature model. While earlier studies developed an approach to extract feature effect constraints on Boolean models, KernelHaven is also able to apply this approach on numeric expressions.
* If information is available to which component a feature belongs to, the aforementioned analysis can be used to check the implemented dependencies among architectural components.
* Features may be traced in code files, variability models, and mailing list archives.
* Feature effect constraints with configurations of legacy products to estimate the amount of configured variables, which had no impact on the overall configuration (impact analysis).
* Variability-aware DeadCode analysis and configuration mismatch analysis to detect code that looks configurable, but is not configurable if code and variability model are analyzed in conjunction.
* Traditional and variability-aware code metrics with more than 42,000 different metric variations to analyze the complexity of the product line’s implementation.
  + 1. Source of Supply

The source code, but also compiled and prepacked bundles can be obtained from KernelHaven’s Github page: <https://github.com/KernelHaven/KernelHaven>

* + 1. Installation and License Information

**Prerequisite**

KernelHaven is platform independent and requires Java 8. However, some of its plug-ins have certain requirements. For instance, for the analysis of Linux, KernelHaven needs to be executed on a Linux machine with build tools installed. Mailing list analysis requires GIT to be installed.

**License information**

KernelHaven is published under Apache v2 license. However, some of its plug-ins contain 3rd party components, for which reason they are published under GPLv3 license.

**Installation**

1. Install Java, preferable Java 8. JRE should be sufficient.
2. Download KernelHaven bundle and unzip it. The GPLv3 bundle contains most plug-ins. Therefore, we recommend this bundle for the beginning.
3. To run a Linux analysis, please install the following packages on a Linux machine (Ubuntu command): sudo apt install openjdk-8-jre build-essential libelf-dev
4. Edit one of the configuration files (\*.properties), which fits best to your needs.
5. Start KernelHaven via the command: java -jar KernelHaven.jar <configuration file>
   * 1. Getting started

This video demonstrates how to install, configure and run KernelHaven to run three different analyses: <https://youtu.be/xKde6tPY_jA>

This video demonstrates how to use KernelHaven to run more than 23,000 code metric variations on the Linux case study containing more than 20,000 files in 3.5 hours, which are less 40ms per code function or to put it differently half a second per metric: <https://youtu.be/vPEmD5Sr6gM>

This video demonstrates how to extract implemented constraints from code artifacts and how to visualize them with FeDeV (cf. Section 2.1): <https://youtu.be/nsWNKIR5Lyc>

* + 1. Settings and configurable properties

KernelHaven can be configured through a configuration file (\*.properties). The provided configuration parameters vary based on the available plug-ins and the selected analysis. Now, KernelHaven offers more than 130 configuration parameters, which may still be revised and, thus, may become outdated. Instead, the KernelHaven bundles are shipped with a config\_template .properties, which contain a description of currently supported parameters.

* + 1. Example data

Please look at the video tutorials on how to use KernelHaven. For testing KernelHaven, we recommend to analyze the sources of the Linux Kernel up to version 4.14, which can be obtained here: <https://www.kernel.org/>

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | <https://github.com/KernelHaven/KernelHaven> |
| Download Link | <https://github.com/KernelHaven/KernelHaven> |
| Source Code Repository | <https://github.com/KernelHaven/KernelHaven> |
| Video Tutorial | KernelHaven manual: <https://youtu.be/xKde6tPY_jA>  MetricHaven manual: <https://youtu.be/vPEmD5Sr6gM>  Short introduction: <https://youtu.be/IbNc-H1NoZU>  KernelHaven + FeDeV manual: <https://youtu.be/nsWNKIR5Lyc> |
| Tutorial Slides | N/A |
| Manual | <https://github.com/KernelHaven/KernelHaven/wiki> |
| Contact | Klaus Schmid <[schmid@sse.uni-hildesheim.de](mailto:schmid@sse.uni-hildesheim.de)>  Sascha El-Sharkawy <[elscha@sse.uni-hildesheim.de](mailto:elscha@sse.uni-hildesheim.de)>  Christian Kröher <[kroeher@sse.uni-hildesheim.de](mailto:kroeher@sse.uni-hildesheim.de)> |

* 1. Variability Extraction and Analysis (VEXA)
     1. Tool Description

The **Variability Extraction and Analysis (VEXA)** toolkit is a versatile collection of complementary procedures to help with many different tasks of variability extraction, feature analysis, visualization and the calculation of user-defined metrics. Implemented as a plug-in for the "world's leading Graph Database" Neo4j[[1]](#footnote-1), the VEXA toolkit leverages the powerful graph storage and processing capabilities of Neo4j to enable detailed dependency analyses of source code artifacts (*e.g.,* #ifdef variability in C/C++ code) and reveal intricate feature connections across project artefacts along with graph visualization possibilities.

The input to VEXA is a collection of software project artefacts that are first processed; then, disassembled into their constituent parts and relations; finally, transformed into a property graph, which contains all relevant information for subsequent analysis steps. The methodology behind VEXA pursues an iterative workflow, in which the user of the VEXA toolkit - be it a software developer or domain expert - specifies all analysis steps in a flexible way - tailored to his specific needs - using Cypher queries. User-defined analysis steps can create new connections in the property graph, *e.g.,* the connection between an *#ifdef* statement and a feature annotation; add intermediate results and metrics, such as the distribution of embedded feature annotations across source files; thus, advancing the overall analysis iteratively.

Context-specific Cypher queries can be employed to support the user in visualizing relevant dependencies between analyzed artefacts - *e.g.,* feature locations and their relation to source files and code elements - in the form of graphs and tables.

* + 1. Source of Supply

The VEXA Toolkit is bundled as a JAR file and can be obtained from the FZI for demonstration purposes.

* + 1. Installation and License Information

**Prerequisite**

The VEXA Toolkit is platform independent and can run on Windows or Linux operating systems requiring the Java 8 Runtime Environment (JRE) or higher. VEXA is a Plug-In for the Neo4j Graph database and can run inside the Enterprise Server, Community Server or Neo4j Desktop editions (versions higher than 3.5).

For extraction and analyses, VEXA relies on third party applications:

* srcML (srcml.org): A scalable parsing tool to convert source code into srcML XML format.
* Z3Prover ([github.com/Z3Prover/z3](https://github.com/Z3Prover/z3)): A theorem prover from Microsoft Research.
* APOC ([github.com/neo4j-contrib/neo4j-apoc-procedures](https://github.com/neo4j-contrib/neo4j-apoc-procedures)): Cypher procedures for Neo4j.

**License information**

The license for the VEXA Toolkit is currently not finalized. You can obtain the status and further information from FZI.

Licenses of third party tools used by VEXA:

* Neo4j CE: GPLv3
* srcML: GPL
* APOC: Apache License 2.0
* Z3Prover: MIT

**Installation**

The VEXA Toolkit is deployed as a JAR bundle that can be copied to the Neo4j “*plugins*” folder. The configuration of VEXA is done in Neo4j global configuration file “*neo4j.conf*”. A default configuration is shipped with the JAR bundle.

* + 1. Getting started

An analysis using the VEXA toolkit is setup of multiple analysis steps executed in order. We show how a user can develop their own source code analysis utilizing the VEXA toolkit. We cover two analysis goals for the ERIKA3 use case. ERIKA3 is a real-time operation system that implements the AUTOSAR OS and OSEK/VDX API specification. As our first analysis goal, we want to gain insight into the code variability of the ERIKA3 operating system. Our second goal is to calculate Feature Effects of preprocessor (#ifdef) definitions in the source code. Our custom analysis is implemented as an analysis script filled with Cypher queries. The Cypher queries can also be run from the Neo4j Browser or the FeDeV tool iteratively.

1. We start by configuring the Neo4j database and setting data constraints as well as indices. The initial setup is done by calling the VEXA query “**vexa.init.database**”. The VEXA toolkit will create the necessary database indices and constraints.
2. The next step in our analysis is to create the VEXA project and add file system resources by calling the procedure “**vexa.project.add.resource**”. We add the whole source code folder of the ERIKA3 operation system. Before we can run the extraction phase, we need to initialize the generated project, so the VEXA plugin can track the added resources.

//== Create VEXA infrastructure for projects

CALL vexa.project.create("ERIKA3-UseCase3") YIELD node

RETURN node.name AS `Project`;

CALL vexa.project.add.resource("ERIKA3-UseCase3",

"file:///local\_disk/REVaMP/vexa/use\_cases/ERIKA3/erika3");

//== Initialize project

CALL vexa.project.init("ERIKA3-UseCase3");

1. The third step in our analysis is the extraction step. First we match all *(:File)* nodes in the database that are C/C++ code and header files and call the VEXA procedure “**vexa.resource.adapt.iterate**” to execute the VEXA adapters. Based on the resource type, the VEXA plug-in will choose an adapter to process files. In this case the *(:Srcml)* adapter will be chosen that creates the abstract source code representation based on the VEXA data model.

//== Run SrcmlAdapter on all C, C Header files in batches

MATCH (f:File) WHERE f.extension =~ '^(c|h|cc|hh)$'

WITH collect(f) as files

CALL vexa.resource.adapt.iterate(files, {batchSize:32, parallel:true, concurrency:8})

RETURN size(files) AS `Processed source files`

1. To get an overview of the variability in the source code, we call the procedure “**vexa.metrics.cpp**” which will analyze the extracted data and generate the code metrics based on the work of Liebig et al.[[2]](#footnote-2)

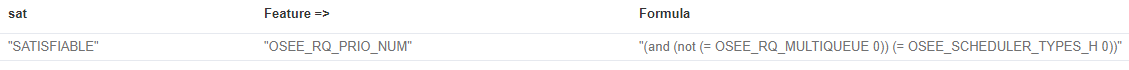
//== Output Code Metrics

CALL vexa.metrics.cpp({headerGuardFilter:'.\*\_[H|H\_|h|h\_]'});



1. To calculate “Feature Effects”[[3]](#footnote-3) we first need to create the presence conditions for each *(:CodeBlock)* node. We call the procedure “**vexa.analysis.createPC**”. Feature effects describe the constraint for selecting a feature. In our analysis, we take all preprocessor symbols as individual features and calculate the feature effect for one symbol. The VEXA plug-in will call the Z3 SMT solver in the background to calculate and simplify the logic formula and present the result in tabular form. For the feature “OSEE\_RQ\_PRIO\_NUM” we get the constraint:

**OSEE\_RQ\_PRIO\_NUM => (!(OSEE\_RQ\_MULTIQUEUE == 0)) && (OSEE\_SCHEDULER\_TYPES\_H == 0)**



//== Calculate presence conditions for :CodeBlock nodes

MATCH (cb:CodeBlock)

WITH collect(cb) AS cbs

  CALL vexa.analysis.createPC(cbs, {batchSize:5000, parallel:true, concurrency:16})

  RETURN size(cbs) AS `Processed (:CodeBlock) nodes`;

//== Feature Effect Analysis

// 1) Merge (:Symbol) nodes

MATCH (s:Symbol)

WITH DISTINCT s.name AS symbol, collect(s) AS symbols

  CALL apoc.refactor.mergeNodes(symbols, {properties:"discard", mergeRels:true}) YIELD node

  RETURN count(node);

// 2) Calculate Feature Effects stemming from source code for all (:Symbol) nodes

MATCH (s:Symbol)

CALL vexa.analysis.calcFE(s.name) YIELD f, sat, simplifiedFormula

RETURN sat, f AS `Feature => `,  simplifiedFormula AS `Formula` LIMIT 1;

In our example, we calculated just one feature effect. We can instruct the VEXA procedure to calculate all feature effects by removing the limit at the end of the Cypher query.

A step-by-step video tutorial for the lwIP use case is presented at REVaMP2 Youtube channel (<https://www.youtube.com/watch?v=Znk2Cul51RQ&t=1s>).

* + 1. Settings and configurable properties

VEXA Toolkit configuration is done in the Neoj4 “neo4j.con” file, where the paths for external plug-ins such as srcML or Z3 need to be set.

* + 1. Example data

A detailed explanation of VEXA’s capabilities is decribed on the REVaMP2 toolchain website (<http://www.revamp2-project.eu/tool-chain>). There you can find links to video tutorials, presentation slides and in depth use case description demonstrating the VEXA workflow.

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | <http://www.revamp2-project.eu/tool-chain> |
| Download Link | N/A |
| Source Code Repository | N/A |
| Video Tutorial | <https://www.youtube.com/watch?v=E1fpI15mZak&t=1s>  <https://www.youtube.com/watch?v=Znk2Cul51RQ&t=1s> |
| Tutorial Slides | <http://www.revamp2-project.eu/images/Tool-chain/tutorial-slides/D4.6-Tool-Presentation-VEXA.pdf> |
| Manual | <http://www.revamp2-project.eu/images/Tool-chain/tutorial-slides/D4.4-Tool-Presentation-VEXA-Case_Study.pdf> |
| Contact | Anton Paule <[anton.paule@fzi.de](mailto:anton.paule@fzi.de)>  Sebastian Reiter <[sebastian.reiter@fzi.de](mailto:sebastian.reiter@fzi.de)> |

* 1. Feature Location in Models by an Evolutionary Algorithm (FLiMEA)
     1. Tool Description

FLiMEA is a tool implemented as a generic and extensible plugin for feature location in models. FLiMEA capitalizes on experts’ domain knowledge to boost the feature location process and produces model fragments that properly capture the reusable assets of the domain. The input to FLiMEA is a collection of product models created with a DSL conforming to MOF, where FLiMEA searches features in the form of model fragments basing on a textual similarity between a textual query (feature description) and the model fragment. Implemented with interchangeable components, FLiMEA support different techniques to preprocess the natural language, extract model fragments, and evaluate the relevance of these model fragments, in order to provide a flexible setup that leverages the powerful experts’ knowledge under different industrial scenarios.

* + 1. Source of Supply

The plug-ins constituting the tool can be requested through the website of SVIT research group: <https://svit.usj.es>

* + 1. Installation and License Information

**Prerequisite**

The tool runs under the Eclipse Modeling Framework which is used for managing the models. The tool also needs the Eclipse Graphical Modelling Framework for the visualization of model fragments.

Moreover, the tool uses as input

product models or families of product models, and

a textual description of a feature that will be used as query.

The models used represent a product that is used to generate code. The models should conform to any DSL conforming to MOF and should be verbose in order to boost the results.

**License information**

The license for the FLiMEA tool is currently not finalized, and the tool is only available for researching purposes yet.

**Installation**

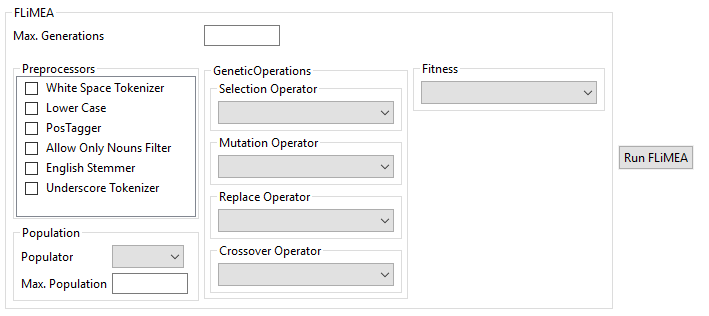
The tool is based on Eclipse respectively Java. In addition, Eclipse Modeling Framework and Eclipse Graphical Framework have to be installed to manage and visualize the models.

* + 1. Getting started

This video demonstrates a simple example of the capabilities of FLiMEA, giving an overview of the main views to analyse the results of the tool: <https://www.youtube.com/watch?v=0ZuTWR5rQPg>

* + 1. Settings and configurable properties

FLiMEA have not to be configured through a configuration file. However, the tool contains a view to specifically configure the executions or experiments performed. Through this view, the users can configure the preprocessors to manage natural language and the setup of the evolutionary algorithm such as the population, the genetic operations, or the fitness function.



* + 1. Example data

In order to obtain more information about how to use FLiMEA, please look at the video tutorial and the presentation slides, which are available on the REVaMP2 website (<http://www.revamp2-project.eu/tool-chain>). Moreover, in the website of the SVIT research group, there are also descriptions and videos of the cases studies and publications about the tool.

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | <https://svit.usj.es/> |
| Download Link |  |
| Source Code Repository | On order, requested through the website of SVIT research group |
| Video Tutorial | <https://www.youtube.com/watch?v=0ZuTWR5rQPg> |
| Tutorial Slides | <http://www.revamp2-project.eu/images/Tool-chain/tutorial-slides/D4.4-Tool-Presentation-FLiMEA.pdf> |
| Manual |  |
| Contact | Ana C. Marcén <[acmarcen@usj.es](mailto:acmarcen@usj.es)>  Jaime Font <[jfont@usj.es](mailto:jfont@usj.es)> |

* 1. Commit Analysis Infrastructure (ComAnI)
     1. Tool Description

The Commit Analysis Infrastructure (ComAnI) enables the extraction and the analysis of commits and the information about their changes to individual artifacts from software repositories. Different plug-ins provide these core capabilities (extraction and analysis) to support a variety of different version control systems, like Git or SVN, and analyses, like determining the intensity of variability changes or the effect changes have on variable code blocks. This configurability is one of the main features of ComAnI and enables different setups of that infrastructure. Besides the specific combination of plug-ins, configurations may differ in the way commits are extracted: a full repository extraction forces the extraction of all commits of a software repository; a partial repository extraction allows the extraction of a predefined set of commits; the single commit extraction offers an interactive mode, in which the content of a single commit can be passed on the command line as an input. A configuration file defines such a particular setup of the infrastructure. It consists of a set of configuration parameters, which the infrastructure reads to configure ComAnI prior to its actual execution.

The second main feature of ComAnI is its extensibility. Besides the available plug-ins for the commit extraction and analysis, the infrastructure enables the development and execution of third-party plug-ins. This enables the application of the infrastructure to different types of repositories and version control systems as well as the processing of currently unsupported artifact types and their content. A developer therefore only has to subclass the respective abstract class for extraction or analysis and then implements the core algorithms of the new plug-in. This new plug-ins is integrated into a new setup by simply defining the location and the main class name of that plug-in in the configuration file. ComAnI manages all other requirements for its execution transparently.

* + 1. Source of Supply

The Commit Analysis Infrastructure can be downloaded from github: <https://github.com/CommitAnalysisInfrastructure/ComAnI>

* + 1. Installation and License Information

**Prerequisite**

The core infrastructure of ComAnI requires Java 8 or higher (or equivalents, like OpenJDK). The individual plug-ins may require additional components, like Git or SVN for the respective extraction plug-ins or R for calculating statistics of change intensities (analysis). These individual requirements can be found as part of the descriptions of the plug-ins in their dedicated github repositories (<https://github.com/CommitAnalysisInfrastructure>).

**License information**

The core infrastructure as well as all plug-ins provided by the Stiftung University of Hildesheim are available under Apache License 2.0.

**Installation**

Download the latest release from <https://github.com/CommitAnalysisInfrastructure/ComAnI/tree/master/release> and extract the zip archive to your favorite location.

* + 1. Getting started

ComAnI is a command line tool, which is executed by starting the ComAnI.jar file as a typical Java archive file, e.g. *java –jar ComAnI.jar [...]*. As the purpose of this tool is to extract and analyze commits, it is necessary to have a local copy of the target software repository in place. The path to that repository is a mandatory configuration parameter, which the infrastructure receives as part of a ComAnI configuration file. We describe these configuration file and some of the options in more detail in Section 2.5.5. Assuming that we have a configuration file, called *MyConfig.properties*, the full command to execute ComAnI is: *java –jar ComAnI.jar MyConfig.properties*.

* + 1. Settings and configurable properties

A ComAnI configuration file defines the setup of the infrastructure, the plug-ins to use, the target software repository, the result directory, etc. Below we briefly describe the configuration options of the core infrastructure. However, some plug-ins may introduce their individual configuration options. Hence, we recommend reading their description to avoid missing options.

ComAnI core infrastructure configuration options:

* *core.plugins\_dir = <Path>*: The path to the directory, which contains the ComAnI plug-ins, like the available extractors and analyzers.
* *core.version\_control\_system = <VCS\_Id>*: The identifyer of the version control system (VCS), which the repository as the input for commit extraction relies on. Commit extractors and analyzers need to support the VCS. See the respective documentations of the desired plug-ins.
* *core.log\_level = <0|1|2>*: The number defining a particular log-level and, hence, the amount of information the infrastructure as well as the plug-ins provide at runtime. Valid values are:
  + 0 - SILENT: No information is provided and, hence, there will be no message at all except for initial setup errors
  + 1 - STANDARD: Basic information, warnings, and errors are provided
  + 2 - DEBUG: Similar to STANDARD, but additional debug information is provided
* *core.commit\_queue.max\_elements = <1|2|3|...|100000>*: The maximum number of commits, which the commit queue manages simultaneously for their transfer from the extractor to the analyzer. If this number is reached, the queue blocks the extractor until the analyzer removes the next commit from the queue.
* *extraction.extractor = <Extractor>*: The fully qualified main class name of the commit extractor to use in the particular ComAnI instance.
* *extraction.input = <Path>*: The path to the directory denoting the root of a software repository from which the commit extractor will extract the commits.
* *extraction.commit\_list = <Path>*: The path to and name of the file containing a list of commit numbers. Extractors will try to extract the corresponding commits from the specified repository exclusively. This is an optional parameter.
* *extraction.cache = <Path>*: The path to the directory for saving extracted commits. Defining this parameter enables the caching feature for the extraction, which allows saving extracted commits as individual files and reuse them in future analyses, while the current analysis processes the extracted commits as usual. This avoids repeating the extraction of the same commits for future analyses. This is an optional parameter.
* *extraction.reuse = <Path>*: The path to the directory containing cached commits. Defining this parameter enables the caching feature for the extraction, which leads to a reuse of previously extracted commits instead of executing the defined extractor. This avoids repeating the extraction of the same commits for future analyses. This is an optional parameter.
* *analysis.analyzer = <Analyzer>*: The fully qualified main class name of the commit analyzer to use in the particular ComAnI instance.
* *analysis.output = <Path>*: The path to the directory for saving the analyis results. Each analysis will create its own sub-directory in this directory named by the name of the analyzer and a timestamp to avoid unintended overriding of previous results.
  + 1. Example data

ComAnI successfully processes the history of the Git-based Linux kernel and Coreboot firmware repository as well as the SVN-based axtls repository. Hence, local copies of these repositories have been used as example data. However, the tool is not limited to these repositories.

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | https://github.com/CommitAnalysisInfrastructure |
| Download Link | https://github.com/CommitAnalysisInfrastructure/ComAnI/tree/master/release |
| Source Code Repository | https://github.com/CommitAnalysisInfrastructure |
| Video Tutorial | <https://www.youtube.com/watch?v=k4ZJvlG-yis> |
| Tutorial Slides | <http://www.revamp2-project.eu/images/Tool-chain/tutorial-slides/D4.4-Tool-Presentation-ComAnI.pdf> |
| Manual | <https://github.com/CommitAnalysisInfrastructure/ComAnI/blob/master/guide/ComAnI_Guide.pdf> |
| Contact | Christian Kröher <[kroeher@sse.uni-hildesheim.de](mailto:kroeher@sse.uni-hildesheim.de)> |

* 1. Problem-Solution-Space Co-Evolution Support (PSS-CE)
     1. Tool Description

The Problem-Solution-Space Co-Evolution (PSS-CE) support tool enables the detection of unintended divergences between artifacts in the problem space (variability model) and the artifacts in the solution space (build and code) during evolution. For example, the tool detects configuration options defined in the variability model, which are not used in build or code artifacts, and vice-versa. The tool consists of three individual plug-ins developed for KernelHaven (cf. Section 2.2), which realize the main steps of the support process. The first step is to create a mapping between problem and solution-space artifacts based on the usage of variability information, like the definition of and references to configuration options. The second step identifies divergences between those artifacts based on the mapping of the previous step, while the third step provides proposals for corrections of the identified divergences. Hence, the tool provides information about potential problems or smells along with guidance for their correction.

* + 1. Source of Supply

The individual plug-ins constituting the tool can be downloaded as part of the KernelHaven distribution (cf. Section 2.2).

* + 1. Installation and License Information

**Prerequisite**

The PSS-CE support tool consists of three individual KernelHaven plug-ins. Hence, it requires KernelHaven as well as its extraction capabilities, which in turn impose further prerequisites (cf. Section 2.2).

**License information**

The PSS-CE support tool (the KernelHaven plug-ins) is available under Apache License 2.0.

**Installation**

Download the latest KernelHaven release bundle from <https://github.com/KernelHaven/KernelHaven> and extract the zip archive to your favorite location.

* + 1. Getting started

The execution of a particular KernelHaven instance is described in Section 2.2. In order to execute the PSS-CE support tool, it is only necessary to define a corresponding KernelHaven configuration, which declares the execution of the respective PSS-CE plug-ins. We describe these configuration options in the next section. This does not change the general execution as described in Section 2.2.

* + 1. Settings and configurable properties

KernelHaven provides a highly configurable infrastructure with various parameters to customize a particular instance. Hence, in this section we focus on those configuration options, which are specific for the PSS-CE support plug-ins. However, a full configuration file can be found in the respective plug-in’s repositories, e.g. at <https://github.com/KernelHaven/ProblemSolutionSpaceDivergenceCorrectorAnalysis>.

PSS-CE support configuration options for KernelHaven:

* In order to combine the three single plug-ins into a comprehensive analysis pipeline, we have to define the main analysis class as a *ConfiguredPipelineAnalysis* as follows: *analysis.class = net.ssehub.kernel\_haven.analysis.ConfiguredPipelineAnalysis*
* The actual definition of the analysis pipeline, which combines the individual PSS-CE plug-ins is stated as follows: *analysis.pipeline = net.ssehub.kernel\_haven.pss\_divergence\_corrector.ProblemSolutionSpaceDivergenceCorrector(net.ssehub.kernel\_haven.pss\_divergence\_detector.ProblemSolutionSpaceDivergenceDetector(net.ssehub.kernel\_haven.pss\_mapper.ProblemSolutionSpaceMapper(cmComponent(), bmComponent(), vmComponent())))*
* Optionally, we can define that intermediate results produced by the mapping and divergence detection step shall be saved as well. This is done as follows: *sanalysis.output.intermediate\_results = ProblemSolutionSpaceDivergenceDetector, ProblemSolutionSpaceMapper*
* As KernelHaven supports multiple result formats, we need to define the desired one for the PSS-CE support tool, which is typically an Excel workbook: *analysis.output.type = xlsx*
* The PSS-CE plug-in for creating the initial problem-solution-space mapping requires the definition of a regular expression to identify the usage of variables in build and code artifacts, which represent configuration options defined in the variability model. This is done as follows: *analysis.pss\_mapper.variable\_regex = CONFIG\_.\**
  + 1. Example data

The PSS-CE support tool successfully processes Kbuild-based software product lines, like the Linux kernel. Hence, parts of that software have been used as example data.

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | Each PSS-CE plug-in has its own homepage:  <https://github.com/KernelHaven/ProblemSolutionSpaceMapperAnalysis>  <https://github.com/KernelHaven/ProblemSolutionSpaceDivergenceDetectorAnalysis>  https://github.com/KernelHaven/ProblemSolutionSpaceDivergenceCorrectorAnalysis |
| Download Link | Either as KernelHaven bundle from <https://github.com/KernelHaven/KernelHaven> or from the individual plug-in homepages above. |
| Source Code Repository | <https://github.com/KernelHaven/ProblemSolutionSpaceMapperAnalysis>  <https://github.com/KernelHaven/ProblemSolutionSpaceDivergenceDetectorAnalysis>  <https://github.com/KernelHaven/ProblemSolutionSpaceDivergenceCorrectorAnalysis> |
| Video Tutorial | <https://www.youtube.com/watch?v=gpBT9wiDRhE> |
| Tutorial Slides | N/A |
| Manual | N/A |
| Contact | Christian Kröher <[kroeher@sse.uni-hildesheim.de](mailto:kroeher@sse.uni-hildesheim.de)> |

* 1. Product-Line-Product-Variant Co-Evolution Support (PLPV-CE)
     1. Tool Description

The Product-Line-Product-Variant Co-Evolution (PLPV-CE) support tool, called Jess, enables the automated migration of semantically related lines of code, like functions or features, from a C preprocessor-based software product line to a product variant and vice-versa. The semantically related lines of code, called *semantic units* hereafter, are a reduced slice of the software, which contains the set of all semantically related lines of code representing a particular function, bugfix, or feature. In order to produce these semantic units, the software is parsed into a code property graph (a graph database) that contains information about the code structure (AST), properties (like statement type, file name, etc.) data-, and control-flow. The nodes of the graph represent source code fragments and the edges represent relations among these fragments. Based on a user-given entry point, which represents the point-of-interest for the user, like a particular function to merge, the tool slices the graph accordingly to list all semantically related nodes of the code property graph. In the next step, the tool translates this list into a lines-of-code-based representation that is syntactically correct. It is based on the same directories, *\*.h*-, and *\*.c*-files as the original software, but contains only the elements that are relevant for the given entry point. This result can be used to merge it into the target software.

* + 1. Source of Supply

The Jess tool can be downloaded from <https://github.com/LPhD/Jess>.

* + 1. Installation and License Information

**Prerequisite**

Jess is a Java application and we have tested it on Debian and Ubuntu. If you plan to work with large code bases such as the Linux Kernel, you should have at least 30GB of free disk space to store the database and 8GB of RAM to experience acceptable performance. In addition, the following software must be installed:

* A Java Virtual Machine 1.8. Jess is written in Java 8 and does not build with Java 6 or 7. It has been tested with OpenJDK-8 but should also work fine with Oracle’s JVM.
* Python 3. Jess implements a client/server architecture where client scripts are written in Python 3. Please note that these scripts are not compatible with Python2.
* Python3-setuptools and python3-dev. Client scripts are installed using *setuptools*. Moreover, some of the Python libraries, on which the client tools depend, are written in C and require header files from *python3-dev* to be present.
* Graphviz-dev. Plotting tools require *Graphviz* and its development files to be installed.
* Gradle 2.x. Jess uses the Gradle build tool, and some features specific to Gradle 2.0 and above.
* Git Jess is hosted on Github, to contribute (and for an easy installation) you need Git.

If you are on a Linux-based system, try the following to download the necessary dependencies:

1. *sudo apt-get update*
2. *sudo apt-get install openjdk-8-jdk gradle python3 python3-setuptools python3-dev graphviz graphviz-dev git*

**License information**

The Jess support tool is available under GNU Lesser General Public License v3.0.

**Installation**

Please make sure Gradle 2.x is installed. Then clone the repository and invoke the build script, which will automatically download and install the dependencies, as follows:

1. *git clone https://github.com/LPhD/Jess.git*
2. *cd Jess*
3. *./build.sh*

In order to test the successful installation, first, invoke the script for testing the Jess server by navigating into the Jess directory and executing *./jess-server.sh*. This will start the server, which can be suspended again by pressing *Ctrl + c*. The next test concerns the Jess client scripts, which are installed into the user script directory typically at *~/.local/bin*. To use these scripts from every location, please make sure this directory is in your path, e.g., by adding the line via *export PATH="$PATH:~/.local/bin"* to your *~/.bashrc*, and restarting the shell. During this process, the server should not be running. You can execute the *jess-import* command without parameters to verify that scripts are installed correctly.

* + 1. Getting started

Once Jess has been installed, the Jess server can be started to import code into the database by executing the *jess-import* command. For this purpose, open a terminal and execute the Jess server first (where $JESS is the Jess root directory):

1. *cd $JESS*
2. *./jess-server.sh*

In a second terminal, we import a project, which we use for creating semantic units and generating patches that can be applied to related projects. Therefore, the source code of the project must be first packed as a tarball. Then we invoke jess-import to import the project:

1. *cd $JESS*
2. *tar -cvzf ProjectName path/to/your/projectCode*
3. *jess-import ProjectName*

This will upload the tarball to the server, unpack it, parse the code, and create a new project named *ProjectName* as well as the corresponding graph database. The next step is to start the *SUI* Python script in the *customScripts* directory, which guides the user through the core process of creating a semantic unit. Hence, we use the previous (second) terminal from the steps above as follows:

1. *cd $JESS/customScripts*
2. *python3 SUI.py*

The script will ask to provide some information (project name and entry point) as input for the slicing algorithm. The project name is the same name you used for the *jess-import* command above. The next step is to define the entry point for the slicing algorithm, which can be either a feature/configuration option or a code statement. A feature/configuration option refers to the name of the symbol that is used as part of an *#if/#ifdef* conditional statement to include or exclude specific feature code. Such a name has to be identified in the code manually. If you would like to set a code statement as entry point, the script requires the ID of that code statement as provided by the AST nodes. Therefore, the user has to define either the unique file name or the fully qualified file name (in case that that name is not unique in the project) in which the statement is located. Then, the script asks for the line number of the desired statement in that file. Finally, the script provides the list of all AST nodes and their IDs at the specified location from which the user has to select one as the entry point. Based on that input, the *SUI* script iteratively gather all graph nodes semantically related to the given entry point. The output of this process is a *Graphviz* *\*.dot*-file and a *\*.png*-file in the *$JESS/customScripts/SemanticUnit* directory.

The result of the slicing algorithm described above is input to the patch generation. This process creates the code representation of the slice by translating the graph nodes to their original code statements. Therefore, we have to execute the *patchCreator* Python script as follows:

1. *python3 patchCreator.py*

The output of that script is the final code slice, which has the same structure as the original project (directory and file names, their nesting as well as line numbers of the code statements). However, it only contains the lines of code that are part of the semantic unit. Further, empty directories or files, as well as non *\*.c*- or *\*.h*-files are not contained either. This result can be used as basis for patch generation or code inspection.

* + 1. Settings and configurable properties

Jess is a tool for customizable slicing of C preprocessor-based software product line. Hence, it provides a number of configuration options for the slicing process. These options can be edited in the *SUI* Python script located in the *customScripts* directory of Jess:

* *includeEnclosedCode*: Whenever a syntax structure is selected that encloses code, this code is included in the semantic unit.
* *followDataFlows*: Follow data flow relations (uses/defines relations).
* *connectIfWithElse*: Always connect an existing *else*-statement, whenever an *if*-statement is selected.
* *searchDirectoriesRecursively*: When a directory node is analyzed, all contained directories are added to the semantic unit and then recursively analyzed.
* *includeOtherFeatures*: When we search for the semantically related lines for a specific feature, we only expand for the occurrence of this feature name. When we reach an implementation that is connected to another feature (via incoming variability edges), we do not search for all other implementations that are annotated with this other feature. We do include the implementations that were reached through all (except variability) edges.
* *lookForAllFunctionCalls*: Whenever a *functionDef* statement is analyzed, additionally look for all calls to this function. If deactivated, the process will only analyze the content of the function.
* *includeVariabilityInformation*: After the analysis is finished, look for variability implementations that affect the semantic unit. This is helpful if you would like to know the variability information of all statements in the semantic unit. Activation does not trigger further analyses.
* *includeComments*: After the analysis is finished, look for comments for the included code and add them to the result. Activation does not trigger further analyses.
* *generateOnlyAST*: The resulting slice contains only AST elements to clarify the illustration. This has no effect on the semantic unit identification process.
* *generateOnlyVisibleCode*: The resulting slice contains only top level AST statements (the statements that contain the lines of code as you see them when you are programming). This has no effect on the semantic unit identification process. This option is mandatory, if you would like to use the *patchCreator* script.
* *DEBUG*: Activate to get more outputs on the console, e.g., in which order the statements are added to the semantic unit. This has no effect on the semantic unit identification process.
  + 1. Example data

Jess contains a directory (*testProjects*) with example projects and results. These projects were used in the video tutorial (<https://www.youtube.com/watch?v=Le2X-Fq5-r8>). Further, there exists a paper[[4]](#footnote-4) describing the results and how to obtain them.

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | <https://github.com/LPhD/Jess> |
| Download Link | <https://github.com/LPhD/Jess> |
| Source Code Repository | <https://github.com/LPhD/Jess> |
| Video Tutorial | <https://www.youtube.com/watch?v=Le2X-Fq5-r8> |
| Tutorial Slides | N/A |
| Manual | <https://jess.readthedocs.io/> |
| Contact | Christian Kröher <[kroeher@sse.uni-hildesheim.de](mailto:kroeher@sse.uni-hildesheim.de)>  Lea Gerling <[gerling@sse.uni-hildesheim.de](mailto:gerling@sse.uni-hildesheim.de)> |

* 1. FINALIsT²
     1. Tool Description

FINALIsT² supports developers and domain experts by identifying, locating and documenting features. FINALIsT2 provides a semi-automatic iterative approach for feature mining. It guides the user through the code base by using a combination of IR technique and static code analysis.

The benefits of this approach are:

* it scales even in huge source code bases (> 1M LoC),
* not so experienced developers/domain experts can identify the entire feature quickly,
* the semi-automatic approach gives feedback (graphically and via metrics) in every step of the feature identification and localization process, and
* arguing on the decision which method, global variable or source file should be included in the feature is easier.
  + 1. Source of Supply

Currently FINALIsT² is not publicly available. In case you have interest to test please write an email to [sten.gruener@de.abb.com](mailto:sten.gruener@de.abb.com) or [andreas.burger@de.abb.com](mailto:andreas.burger@de.abb.com).

Additionally, FINALIsT² needs a running Understand installation from [scitools](https://scitools.com/features/).

* + 1. Installation and License Information

FINALIsT² is written in pure Python and can be run on every OS supporting Python 3.7.

* + 1. Getting started

Once the FINALIsT² zip-file is extracted, you will find the folder and file structure shown in Figure 2 in the extracted folder.

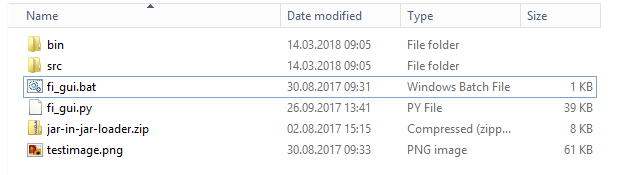


Figure 2 FINALIsT² Folder and File Structure

On a Windows System, you can simply execute the fi\_gui.bat file and the FINALIsT² will start. On Linux or MacOS X systems you need to start FINALIsT² from the command line by executing the command: python3 fi\_gui.py .

For additional information how to use FINALIsT² please be referred to the following video tutorial available on YouTube: <https://www.youtube.com/watch?v=Icv_UwKYj3A> .

* 1. pure::variants
     1. Tool Description

pure::variants is a tool for efficient variant management supporting the whole life cycle of product lines starting from the requirements phase through design, architecture, and realization phase up to the testing phase. Due to its independence for example of programming languages or modeling tools it integrates nearly seamlessly into existing development processes.

The basic idea of pure::variants is the separation of concerns and as such the distinction between a problem domain that considers the variability of the system from an abstract point of view and a solution domain that deals with implementation artefacts like requirements, architecture models, source code, etc. pure::variants captures the problem space within feature models and the solutions with family models separately and independently in a structured way. Additionally, those models allow to define inter-relations and restrictions on the building elements that cannot be expressed by just structuring the information in the right manner. From the management perspective the models allow a uniform representation of all variabilities and commonalities of the products of a product line and the therein captured knowledge provides the basis to perform informed, comprehensible and valid decisions during the definition of variants.

* + 1. Source of Supply

Evaluation versions of pure::variants can be downloaded from pure-systems’ web site: <https://www.pure-systems.com/>

* + 1. Installation and License Information

**Prerequisite**

pure::variants needs Java 1.6+ and Eclipse 3.6+ (Eclipse 3.8 is part of the installation package)

**License information**

There is an evaluation version for non-productive use. For productive use, a license has to be bought.

**Installation**

pure::variants provides an installation package for installation. Extensions, which are created for pure::variants in this project are not part of the standard installation package. They can be requested from pure-systems. Contact [uwe.ryssel@pure-systems.com](mailto:uwe.ryssel@pure-systems.com) for that.

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | <https://www.pure-systems.com/products/pure-variants-9.html> |
| Download Link | https://www.pure-systems.com/downloads-6.html |
| Source Code Repository | N/A |
| Video Tutorial | <https://www.pure-systems.com/Tutorials.101.0.html> |
| Tutorial Slides | <https://www.pure-systems.com/Tutorials.101.0.html> |
| Manual | Accessible within pure::variants |
| Contact | Uwe Ryssel <[uwe.ryssel@pure-systems.com](mailto:uwe.ryssel@pure-systems.com)> |

* 1. Jittac
     1. Tool Description

Jittac, the “Just-In-Time Tool for Architecture Consistency” is an open source architecture analysis and recovery tool originally developed by Lero – The Irish Software Research Centre and extended by Karlstad University, Sweden. It implements an interactive technique to architecture recovery and consistency checking called reflexion modelling. In this approach, the software architecture of a system is modelled in terms of components/modules representing logical building blocks of the system’s code base and desired/expected dependencies between these building blocks. After modelling the architecture, the software architect maps those building blocks to source code entities to express that these entities of the implementation are part of the more coarse-grained architectural building block. After that, source code analysis is applied to detect the dependencies between source code entities, compare them to the desired/expected dependencies in the architectural model, and visualize divergences in the model.

Jittac provides additional capabilities that refines the conceptual reflexion modelling approach such as visualization of divergences such as undesired dependencies in the source code and just-in-time functionality, which means that the tool gives immediate feedback when editing the source code has architectural consequences. If, for example, the software engineer introduces a new undesired dependency by calling a method that should not be accessible according to the architecture model, the corresponding source code fragment is highlighted immediately. Furthermore, we extended Jittac towards feature and feature dependency analysis in (see D4.3 and D4.5).

* + 1. Source of Supply

Jittac can be retrieved by contacting Sebastian Herold ([sebastian.herold@kau.se](mailto:sebastian.herold@kau.se)).

**Prerequisite**

Jittac is implemented as Eclipse plugin and hence requires an installation of the IDE including the Java Development Tools (JDT).

**Installation**

Jittac can be installed by copying the obtained jar files into the plugins folder of your local Eclipse installation.

* + 1. Getting started

Please refer to the following videos and tutorials:

* <http://actool.sourceforge.net/> (original version by Lero)
* <https://www.youtube.com/watch?v=1Hm2OSoJF6s>
  + 1. Settings and configurable properties

If your tool supports any command line flags/parameters or can be configured by a properties file, describe the parameters here.

* + 1. Example data

Please contact Sebastian Herold if you are interested in example datasets

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | N/A |
| Download Link | N/A |
| Source Code Repository | Please contact Sebastian Herold for access |
| Video Tutorial | <https://www.youtube.com/watch?v=1Hm2OSoJF6s> |
| Tutorial Slides | N/A |
| Manual | N/A |
| Contact | Sebastian Herold <[sebastian.herold@kau.se](mailto:sebastian.herold@kau.se)> |

* 1. Systems Engineering Suite (SES)
     1. Tool Description

Systems Engineering Suite is the natural step forward from the Requirements Quality Suite (RQS) to the Model Based Systems Engineering approach. Keeping the concept of requirements quality, SES leverages other System Engineering activities providing traceability, retrieval and reuse, and quality for different types of engineering items.

* + 1. Source of Supply

**Prerequisite**

The SES Architectural, Hardware and Software prerequisites can be found at:

<https://www.reusecompany.com/public-downloads/Public-downloads/Systems-Engineering-Suite/1.-Architecture-and-Deployment-(including-HW-and-SW-requirements)/>

**License information**

All licensing information is described at the chapter three of the following document:

<https://www.reusecompany.com/public-downloads/Public-downloads/Systems-Engineering-Suite/5.-SES-Server-Instalation-and-Licensing-Guide/>

For evaluation and academic purposes please contact us at: [contact@reusecompany.com](mailto:contact@reusecompany.com)

**Installation**

* + 1. Installation and License Information

Apart of the information supplied previously each tool of the suite has his own installation guide.

The installation guides can be found at:

<https://www.reusecompany.com/public-downloads/Public-downloads/Systems-Engineering-Suite/>

* + 1. Relevant links

|  |  |
| --- | --- |
| Project Homepage | <https://www.reusecompany.com/> |
| Download Link | <https://www.reusecompany.com/public-downloads/Public-downloads/Systems-Engineering-Suite/> |
| Video Tutorial | <https://www.youtube.com/watch?v=-H_3aN17d9o> |
| Contact | [contact@reusecompany.com](mailto:contact@reusecompany.com) |

* 1. SIMULTime
     1. Tool Description

SIMULTime is a measurement-based timing analysis tool that allows evaluating the execution time of programs executed on embedded hardware platforms. In particular, SIMULTime produces fast and accurate timing estimations that consider multiple input data sets and different hardware platforms. These estimations are produced executing context-sensitive timing simulations. Multiple configurations can be analysed in parallel executing only one simulation. In addition, SIMULTime can be integrated in MATLAB/Simulink for visualizing the hardware related timing effects executing native Simulink simulations.

* + 1. Source of Supply

**Prerequisite**

At least one hardware platform including a tracing port is required as well as the non-intrusive tracer Lauterbach Trace32.

**License information**

The license for SIMULTime is currently not finalized. You can obtain the status and further information from FZI.

SIMULTime requires the following tools and libraries licensed under:

* LLVM Compiler Infrastructure - [License information](https://llvm.org/foundation/relicensing/)
* Radare2 - LGPLv3
* RapidJSON – MIT license

The MATLAB/Simulink integration requires:

* Embedded Coder Toolbox

**Installation**

SIMULTime is a collection of sub-tools, libraries and scripts. No installation guide is available. Please contact us for further information.

* + 1. Getting started

The SIMULTime’s workflow is composed of five consecutive phases as shown in Figure 3.

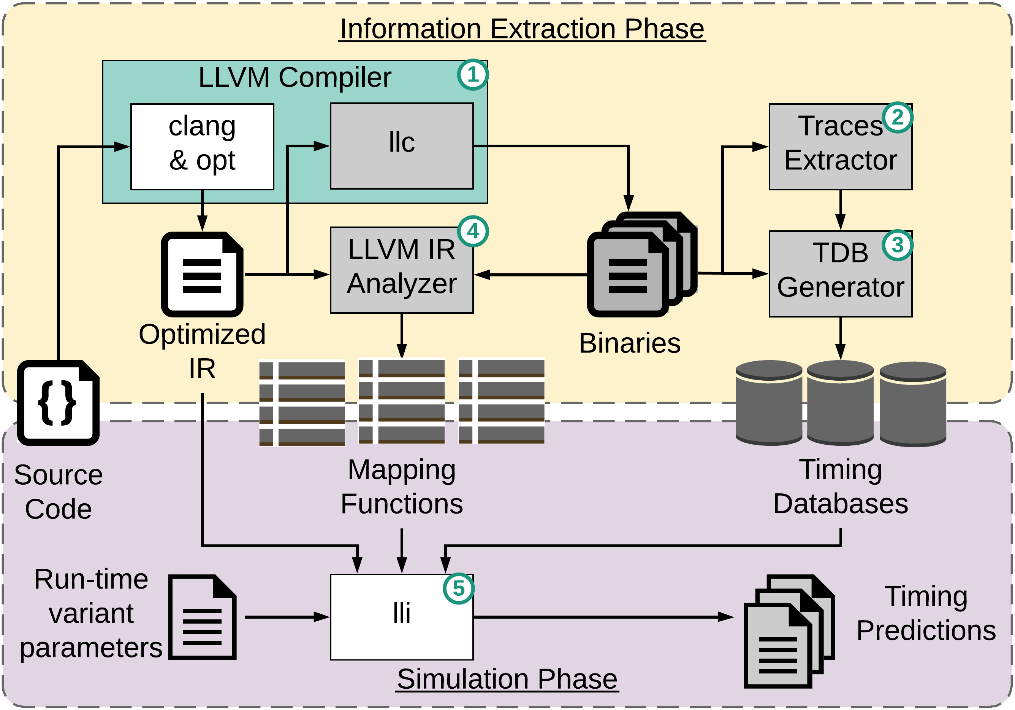


Figure 3 SIMULTime Workflow

A simple example has been created, inspired from one of the real use-cases, in order to show in a practically way the full methodology workflow. The example consists in a simple monitoring system of a production plant that computes the checksums for all the values read by the connected sensors. The execution time of the SW program needs to be evaluated on two different HW variants that are utilized in the PL. The SW program contains run-time variability that is expressed via configuration files, as shown in Figure 4.

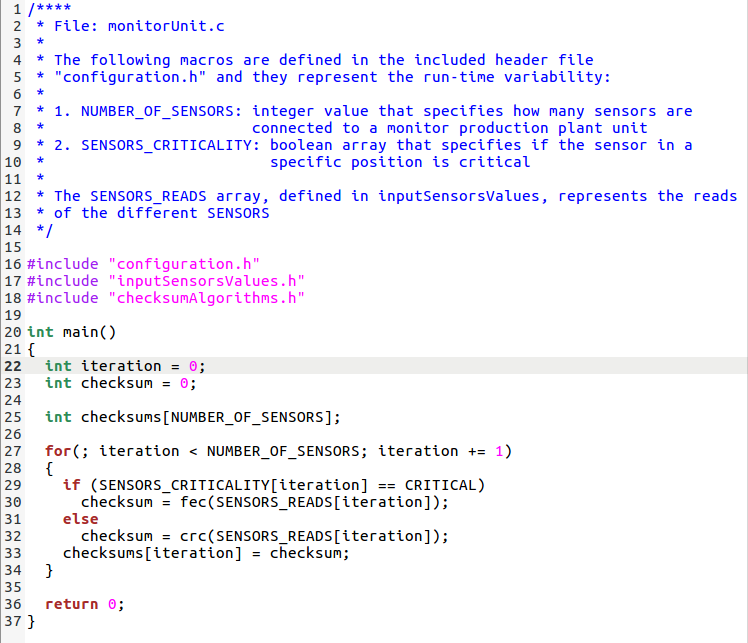


Figure 4 Source code example that contains run-time variability

The main outputs of the compilation phase are the binary executable and the program IR representation. The IR representation of the main function is shown in Figure 5**Fehler! Verweisquelle konnte nicht gefunden werden.**, the code is HW independent and a different label identifies every basic block.

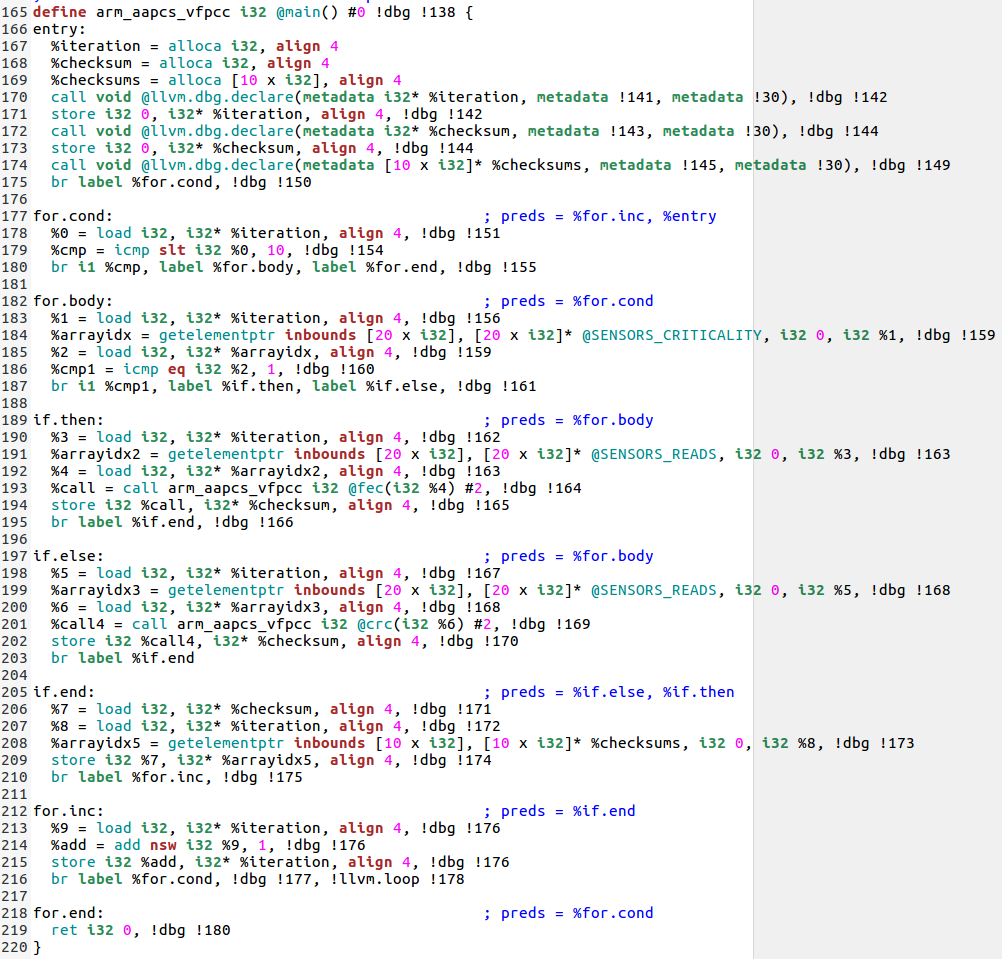


Figure 5 IR representation of the previous C source code.

Timing traces are extracted via the non-intrusive Lauterbach Trace32 tool. The tool allows easily benchmarking the run-time variability contained in the software program. The timing traces are utilized for generating the necessary timing databases.

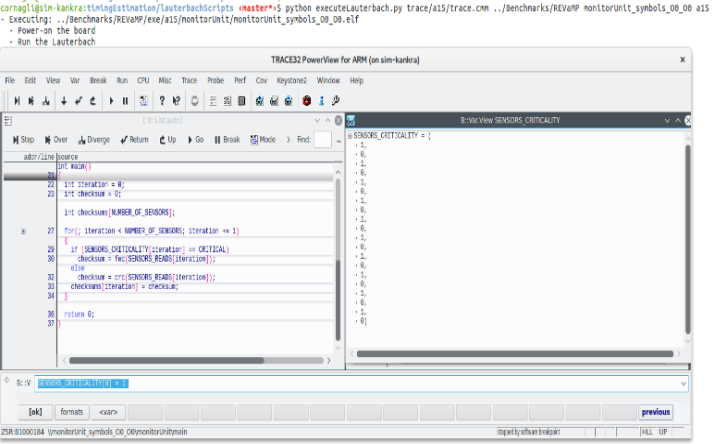


Figure 6 Traces Extractor and run-time variability stimulation

A mapping function is necessary for associating a binary timing database to the IR code. As shown inFigure 7, the binary and IR control flow graphs may differ due to compiler optimizations.

The different timing predictions can be produced by running a modified version of *lli*, the LLVM IR execution engine. The tool is run on the host machine, not in any of the target HW platforms, and it directly executes the bitcode representation of the SW program. The predictions for the execution time of a single SW variant for two different HW platforms can be obtained as shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**.

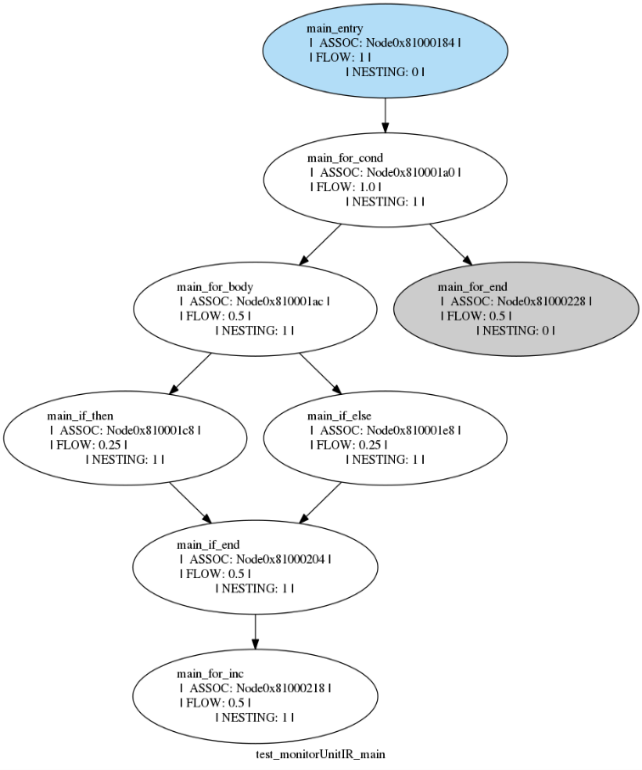
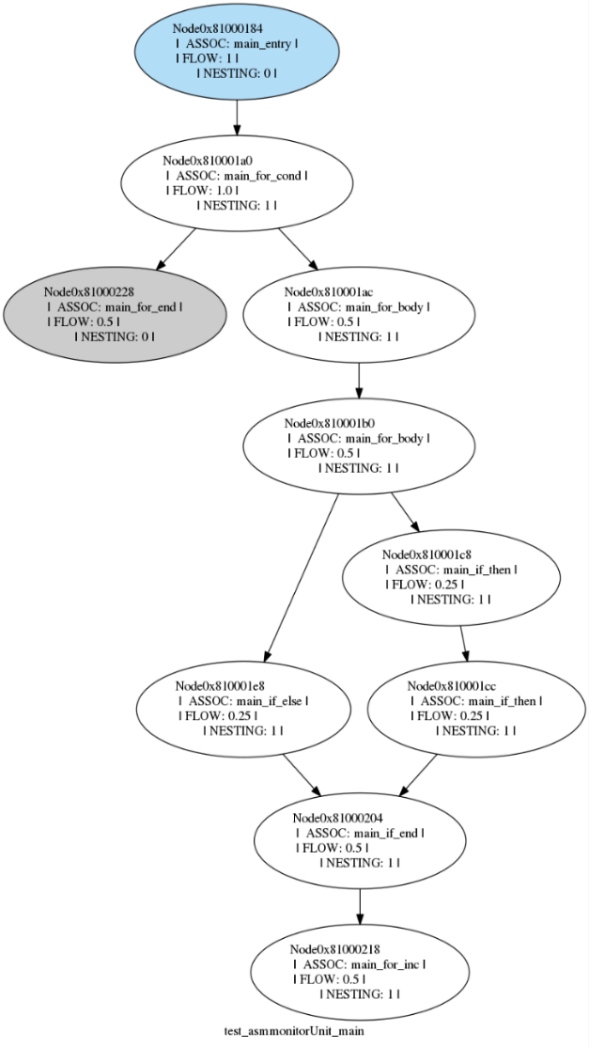


Figure 7 Enriched graphs derived from the CFGs that support the mapping procedure.

The different timing predictions can be produced by running a modified version of *lli*, the LLVM IR execution engine. The tool is run on the host machine, not in any of the target HW platforms, and it directly executes the bitcode representation of the SW program. The predictions for the execution time of a single SW variant for two different HW platforms can be obtained as shown in Figure 8**Fehler! Verweisquelle konnte nicht gefunden werden.**.

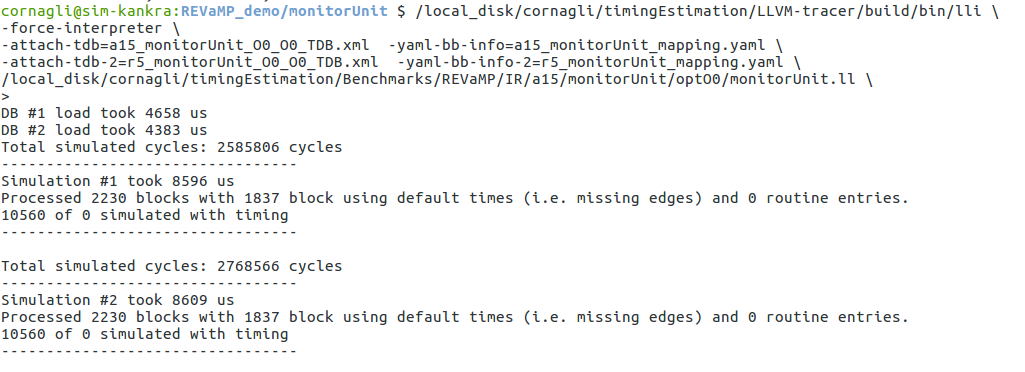


Figure 8 Generation of the timing estimations for one run-time variant of the example program considering two different HW architecture.

A step-by-step video tutorial for the use case is presented on the REVaMP2 Youtube channel (<https://www.youtube.com/watch?v=kuSK739infA>).

**Evaluation**

We provide demos on request.

* + 1. Relevant links

|  |  |
| --- | --- |
| Video Tutorial | <https://www.youtube.com/watch?v=kuSK739infA> |
| Contact | Alessandro Cornaglia – [cornaglia@fzi.de](mailto:cornaglia@fzi.de)  Sebastian Reiter – [sreiter@fzi.de](mailto:sreiter@fzi.de) |

* 1. DragonflyME
     1. Tool Description

The Dragonfly modelling environment (DragonflyME) supports users with testing existing software intensive systems (SIS) that contains variability. We assume that an executable simulation model exists, that abstracts the functional behaviour of the system and contains all the variability of the SIS product line (PL). The overall goal is to specify the system under test (SUT) in the DragonflyME with all its variation points and to generate test instances that are verified with the executable simulation model.

The objective of the developed plug-ins is to provide an iterative algorithm to select feasible instances from the input and PL variability space for testing. With the help of simulations, a quality metric or more precise, an objective function, such as the execution delay of a SIS, is determined. The mapping of the objective function over the search space is called fitness landscape. Based on the sampled fitness landscape, the tool selects new test vectors. By taking both the input space and the PL space into account, we assume that the overall test effort for the complete SIS PL can be reduced, by iteratively narrowing down the representative test cases.

The tooling consists of an editor to specify the simulation model of the SIS PL, code generators, a simulation framework, to support the execution of multiple simulations as well as the exploration algorithm for simulation instance selection.

* + 1. Source of Supply

**Prerequisite**

As mentioned the approach assumes that an executable simulation model exists, that abstracts the functional behaviour of the system and contains all the variability of the SIS PL. The automatic generation of such a simulation model is not focus of the tool. DragonflyME only supports the user with the generation of the structural aspects as well as the dynamic configuration of the simulation model.

The modelling environment is based on the Eclipse Modeling Tools and requires the Papyrus UML plug-in as well as different supporting plug-ins that can be obtained via the Eclipse marketplace. The framework of for the simulation models is based on C++/SystemC as well as supporting public libraries such as Boost or Xerces-C++.

**License information**

The license for the Dragonfly Modeling Environment is currently not finalized. You can obtain the status and further information from the FZI.

**Installation**

The modeling environment is based on Eclipse respectively Java. Beside the Java Runtime Environment no further installations are required.

* + 1. Getting started

The main prerequisite is the simulation model of the SIS PL. The modeling environment supports simulation models based on SystemC/C++. To support the user and reduce the effort for executing a single simulation, the tool provides mechanism such as a runtime configuration approach and code generation steps. Nevertheless, the simulation models are application specific and have to be provided for each SIS PL. The framework supports the user in the creation and managing of the simulation models.

The second prerequisite is the variability specification of the SIS PL, more precise the variability of the simulation model. The variability specification covers the variation points, the possible values of configuration parameters as well as dependencies between variation points. The DragonflyME utilizes a VEL based representation, to link the variation point constraints to the design entities. The import to DragonflyME extracts the constraints and annotates them to the corresponding entities in the VP design specification. This specification constraints are used to describe the valid search space used for iterative test case generation. The approach uses the same format to specify the variability of the SIS PL as well as the variability of the input space, which could be specified via the modeling environment.

With the simulation model as well as the variability specification available the generation of the test instances can be triggered. The tool will generate an IP-XACT-based configuration file for each simulation run. With both the simulation configuration files as well as the simulation model the simulation can be executed. The execution of the simulation is not part of the modeling environment and the user is responsible to provide the required runtime environment. After the execution of the generated simulation instances, the user is responsible to evaluate the monitored behavior and calculated the fitness landscape, meaning associating each generated configuration with a numerical value, specifying the system behavior based on the objective function, e.g., the delay of a calculation. This numerical values can be imported to the modelling environment and a new set of test configurations can be generated. With this iterative approach the test space can be sampled, enabling the guided testing of the variant space.

**Evaluation**

The following section depicts how the approach implemented in the DragonflyME would work. The shown use case is a SUT, with two variation points that could be configured with values up to 400. Figure 9 shows the fitness landscape, if all possible variations would have been evaluated. Figure 10 shows the test instance generated by the iterative approach. It could be seen, that fluctuating regions with a high behavioural variance are tested more often, than regions with a stable system behaviour.

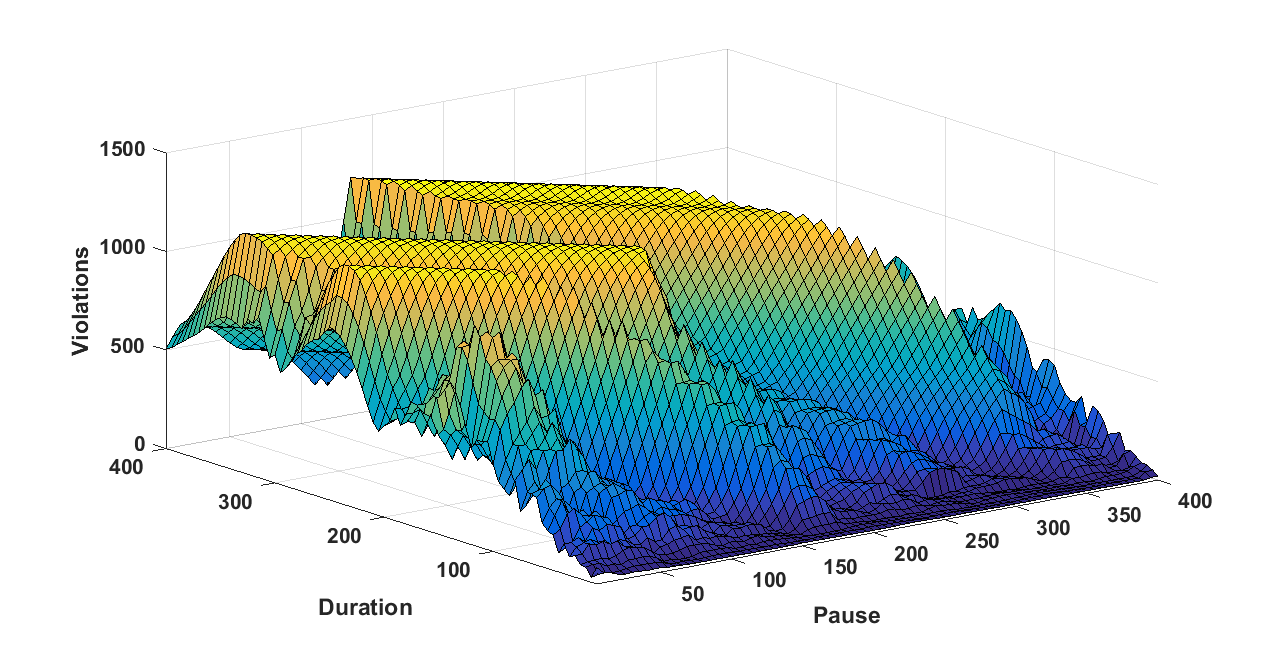


Figure 9 The fitness landscape over two variability parameters

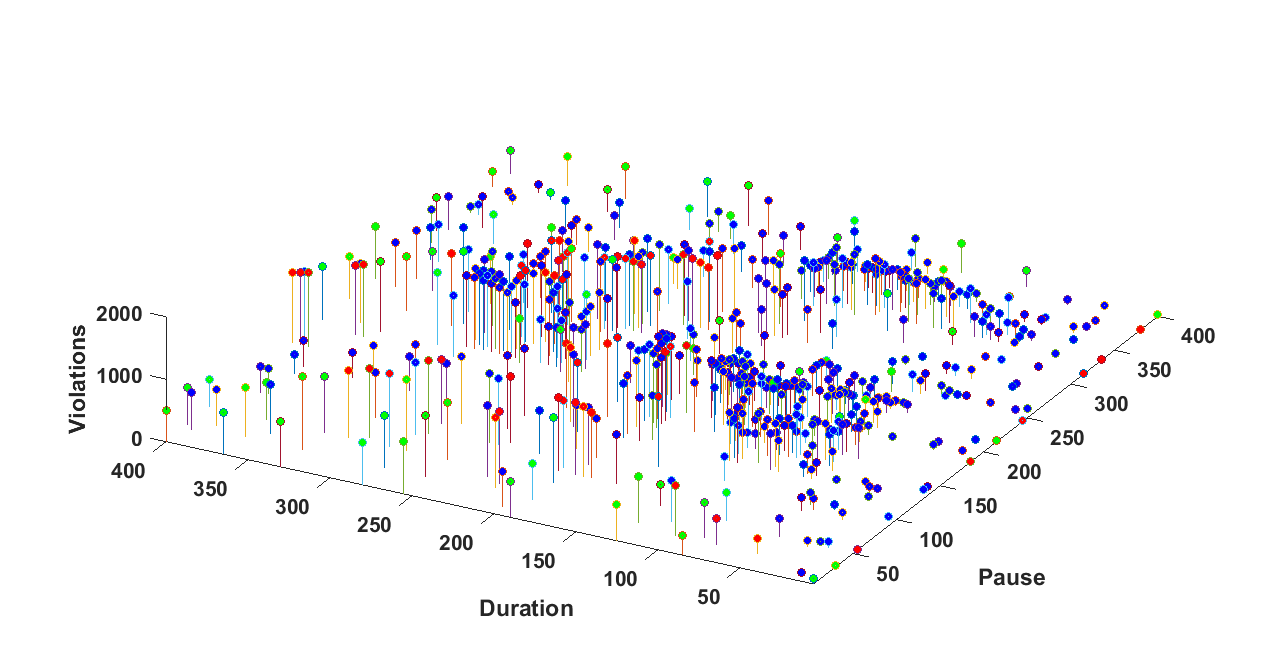


Figure 10 The test instances generated by the iterative approach

* + 1. Relevant links

|  |  |
| --- | --- |
| Contact | Paolo Care – [pcare@fzi.de](mailto:pcare@fzi.de)  Sebastian Reiter – [sreiter@fzi.de](mailto:sreiter@fzi.de) |

1. Predefined Workflows

This section gives an overview of some predefined workflows and the corresponding REVaMP2 tool-chains. Each description consists of a short scenario for use, a brief workflow description, settings forf the tools to be used in the workflow and benefits of using the tool-chain.

* 1. Feature Annotation Extraction and Visualization Workflow

Involved tools are FINALIsT², BUT4Reuse, VEXA and FeDeV.

* + 1. Scenario for use

The integration of ABB’s FINALIsT², BUT4Reuse, FZI’s VEXA and ScopeSET’s FeDeV, is aimed at providing integrated extraction and visualization of feature annotations and variation points. This allows developers to identify, locate, document and isolate features and make them visible as well as to define a feature model. In this step, the differences between product variants are identified.

* + 1. Workflow

Involved tools and actions of this workflow are shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**.

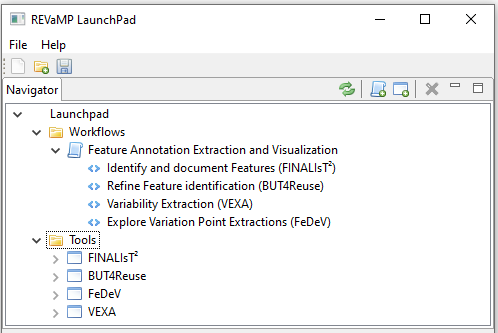


Figure 11 Feature Annotation Extraction and Visualization Workflow

The involved tools for this workflow are called in succession.

**VEXA**

Prerequisites

* A running Neo4j database (Enterprise, Community Edition, Desktop) with the VEXA plugin installed and configured.
* The source code under analysis needs to be located on the host running the Neo4j database.

Settings

* srcML binary and library paths are configured using the VEXA option: “vexa.srcml.bin=<path>” and <vexa.srcml.lib=<path>”

Execution

* All execution steps are run by either running a Cypher script file or running the Cypher iteratively using the FeDeV Cypher view.
* Create VEXA project.
* Add source code folders.
* Initialize resources and run the VEXA “SrcmlAdapter” on all resources.
* Perform metrics calculation.
* Perform dependency analysis.

Results

* *What are the results of the execution?*
* VEXA will generate all results and store them in the Neo4j graph database.
* Results are retrieved via Cypher queries.
* Metrics calculation results are in tabular form and can be viewed in the FeDeV results view.

**FeDeV**

Prerequisites

* A running Neo4j database
* (optional) VEXA analysis results stored in the Neo4j database (can be done in FeDeV, too)

Settings

* If required, enter following values in the FeDeV settings dialog:
  + Neo4j Bolt URL (if not default URL and port are used)
  + If Neo4j uses authentication, enter username and password of Neo4j database user

Execution

* Connect to VEXA Neo4j and import VEXA data via *File* > *Load VEXA Analysis*
* Explore data in graph views
* Use cypher view to execute VEXA procedures in Neo4j

Results

* Visualization of VEXA analysis results
* Cypher view can be used to interact with Neo4j and VEXA
* Search and explore the VEXA dataset
  + 1. Benefits

Combining these tools allows developers to identify, locate, document and isolate features and make them visible as well as to define a feature model by feeding back the insights won by the visual inspection process. This is the first step to migrate different repositories and product families into a Software Product Line.

* 1. Extraction and Variability Management Workflow

Involved tools are BUT4Reuse and pure::variants.

* + 1. Scenario for use

This sub-tool chain has been integrated by BUT4Reuse’s team. It combines the capabilities on extraction and construction of BUT4Reuse with the variability management of pure::variants. It represents a solution for a round-trip engineering of some related clone-and-own software systems by refactoring them to an SPL, for their methodological engineering. In particular, this integration allows a user to identify and extract the variabilities of software systems from source code artefacts and to refactor them into a software product line (SPL), which represents a kind of 150% model of reusable code assets, including the construction of their feature model (FM). Then, the extracted SPL and FM can be used to derive back the software systems.

* + 1. Workflow

Involved tools and actions of this workflow are shown in Figure 12.

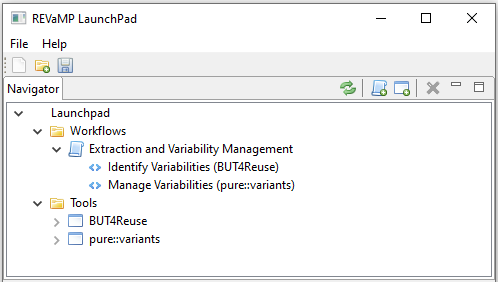


Figure 12 Extraction and Variability Management Workflow

The involved tools for this workflow are called in succession.

* + 1. Benefits

BUT4Reuse is mainly used to identify and extract the variabilities of some related software variants. Its integration with pure::variants makes possible, in the first place, to generate back these software systems. In particular, it makes possible the construction of the extracted artefacts (*i.e.,* of reusable assets, including the feature model), in BUT4Reuse, to a format supported by pure::variants. Its benefits can be several, naming the testing of feature identification and extraction approaches during reverse engineering, or a quick refactoring of related software systems to an SPL for their variability evolution during forward engineering.

* 1. Constraint Extraction Workflow

Involved tools are KernelHaven, Configuration Mining and pure::variants.

* + 1. Scenario for use

The toolchain has been integrated by Robert Bosch GmbH and has the purpose of extracting configuration constraints concerning an already existing product line component from either the source code of that component, using KernelHaven, or from the past configuration (feature selection) data, using Configuration Mining. Subsequently, the constraints can be used to create a new feature model or to enrich an existing one with new constraints. These task need to be performed using a feature modeling tool – in the toolchain, the pure::variants tool is hence extended with a constraint import plugin which reads the constraint file from either KernelHaven or Configuration Mining and stores the constraints in the feature model.

In a practical scenario, the constraints extracted by the toolchain should be first reviewed by a domain expert to evaluate whether they conform to the domain knowledge. The benefit of using the toolchain is the reduction of feature modeling effort. The complete toolchain is schematically depicted in the Figure 13 below.

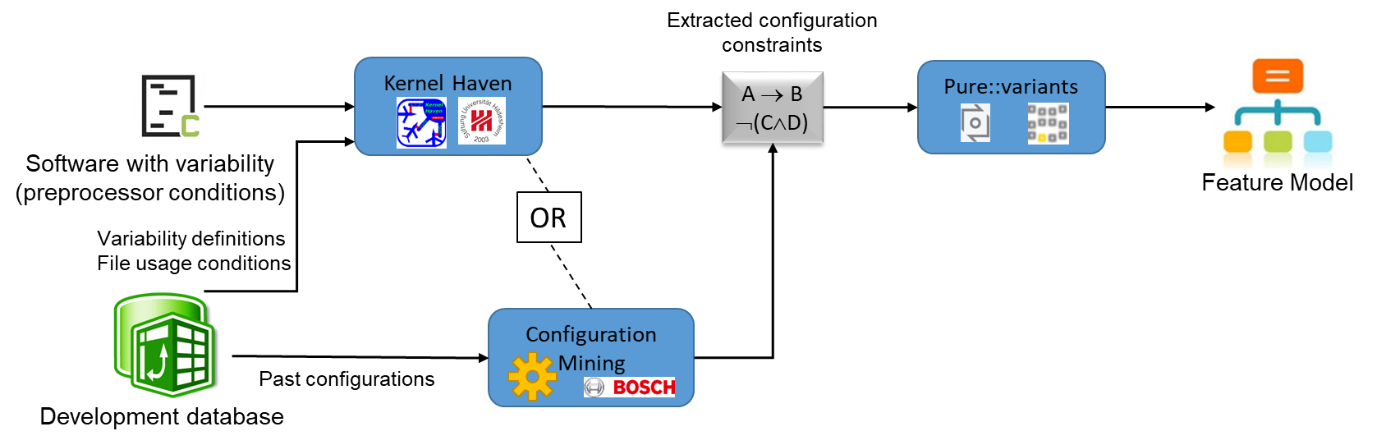


Figure 13 The constraint extraction toolchain using KernelHaven, Configuration Mining, and pure::variants

* + 1. Workflow

Involved tools and actions of this workflow are shown in Figure 14.

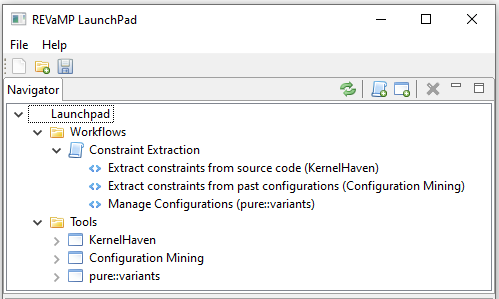


Figure 14 Constraint Extraction Workflow

**KernelHaven**

Prerequisites

* As described in Section 2.2, the general KernelHaven prerequisites have to be satisfied and the required installations have to be available
* Some Bosch-specific plug-ins need to be available, which are not part of the official public releases as they contain company-specific and, hence, confidential information

Settings

* The specific KernelHaven configuration uses some of the publicly available plug-ins: the code block extractor and the feature effect analysis
* The specific KernelHaven configuration also uses some Bosch-specific plug-ins (see above) as well as Bosch-specific configuration parameters
* Due to the previous point and the inherent confidential information, we cannot provide a full setting description here

Execution

* Execute KernelHaven as follows: *java -jar KernelHaven.jar constraint\_extraction.properties*
* The file *constraint\_extraction.properties* contains the settings described above

Results

* An Excel workbook (*\*.xlsx*) containing the presence conditions extracted from code artifacts (e.g., defined by pre-processor statements)
* The detailed content is Bosch-specific and, hence, confidential
  + 1. Benefits

The existing source code and past configurations represent artifacts that were used to create correct variants products. Hence, they were created with the knowledge of the constraints that need to be adhered to by the variant products. The use of the presented toolchain for constraint extraction enables the developer to reduce the effort of creating a feature model for an already existing product line. The existence of the feature model, in turn, makes it easier to configure new products in the future.

* 1. Feature Dependency Visualization and Traceability Workflow

Involved tools are KernelHaven, PSS Mapper, FeDeV and Eclipse Capra.

* + 1. Scenario for use

The integration of SUH’s KernelHaven and ScopeSET’s FeDeV, is used by several REVaMP use cases in order to visualize extraction results. The additional integration of Eclipse Capra and the Problem-Solution-Space (PSS) Mapper, which is part of the PSS-CE support tool described in Section 2.6, provides integrated feature dependency visualisation and feature traceability. This allows developers to review the features defined in the source code, the constraints that are defined on them, and directly navigate to the source code files in which these definitions occur. If additional assets are available (*e.g.,* requirements, test cases, or design models), it is also possible to include them in the traceability analysis. This makes it easier to comprehend the feature dependencies in their context, check for mistakes in the code, and directly edit the underlying definitions if needed.

* + 1. Workflow

Involved tools and actions of this workflow are shown in Figure 15.

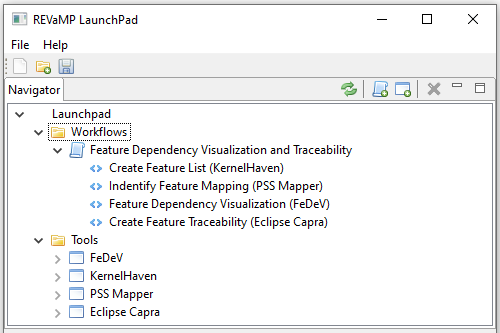


Figure 15 Feature Dependency Visualization and Traceability Workflow

**KernelHaven**

Prerequisites

* As described in Section 2.2, the general KernelHaven prerequisites have to be satisfied and the required installations have to be available
* All required plug-ins are part of the official and publicly available release

Settings

* All settings have to be defined in the KernelHaven *\*.properties* file
* The configuration of information extraction capabilities depends on the technologies used to realize the software product lines; for the Linux kernel we use:
  + The *KconfigReaderExtractor* as *variability.extractor.class*
  + The *KbuildMinerExtraxctor* as *build.extractor.class*
  + The *CodeBlockExtractor* as *code.extractor.class*
* The *analysis.class* property must be set to *ArchitecturalDependencyVisualization* to provide the correct output data
* In order to visualize the analysis results, set the *analysis.output.type* to *sqlite*

Execution

* Execute KernelHaven as follows: *java -jar KernelHaven.jar visualization.properties*
* The file *visualization.properties* contains the settings described above

Results

* An SQLite database containing the identified features and their constraints

**PSS Mapper**

Prerequisites

* The PSS Mapper is part of the PSS-CE tool (cf. Section 2.6) and, hence, a KernelHaven plug-in
* Hence, the same prerequisites as for KernelHaven above apply

Settings

* All settings have to be defined in the KernelHaven *\*.properties* file
* The *analysis.pss\_mapper.variable\_regex* property must be set to a regular expression, which enables the identification of configuration options or features of the variability model in code artifacts, e.g., to *CONFIG\_.\** for the Linux kernel

Execution

* Execute KernelHaven as follows: *java -jar KernelHaven.jar visualization.properties*
* The file *visualization.properties* contains the PSS Mapper settings described above and the KernelHaven settings described in the previous paragraph

Results

* An SQLite database containing the identified features and their constraints as well as their usage in code artifacts

**FeDeV**

Prerequisites

* Configure KernelHaven properties, so a sqlite file is produced with KernelHaven’s analysis execution

Settings

* When loading the KernelHaven analysis database, choose an excel file to store additional remarks and notes

Execution

* Load KernelHaven database via *File* > *Load KernelHaven Analysis*
* Explore Submodules, Features and their Dependencies as well as Constraints set to individual Features.
* Make notes on remarks or change requests in a separate note file (Excel format)

Results

* Interactive exploration of features from a codebase and their relationships among each other
* Overview and visual understanding of features and constraints contained in a project

**Eclipse Capra**

Prerequisites

* *Short description of things, which are required to work with this tool in this workflow*

Settings

* *Short description of required settings*
* Make sure, that FeDeV and Capra communicate on the same port (see section 2.1.5)

Execution

* *Describe steps to be taken in this workflow*

Results

* *What are the results of the execution?*
  + 1. Benefits

Combining these four tools allows developers to analyse features and their constraints in the context of the source code and other relevant artefacts and thus provide value for analysis and comprehension tasks. A typical workflow consists of running KernelHaven on the source files, while the PSS Mapper creates its mapping using KernelHaven’s results. The resulting database is visualised in FeDeV. Whenever the developer selects a feature in one of the FeDeV views, Eclipse Capra would then update its traceability visualisations and present, *e.g.,* the list of files in which a feature is present. Eclipse Capra could also show a traceability matrix that indicates the files in which a certain feature is present. From these views, the user could navigate to the corresponding source code files and view the feature definitions there.

* 1. Identify and Inspect Feature Locations Workflow

Involved tools are Jittac and FLOrIDA.

* + 1. Scenario for use

This sub-tool chain combines BUT4Reuse’s feature location capabilities and Jittac’s architecture visualization and analysis capabilities. The goal is to visualize features and their interdependencies at the architectural level using Jittac based on information about the features’ locations in source code as obtained from BUT4Reuse.

* + 1. Workflow

I First, FLOrIDA is used to identify the locations of a feature in the source code of a system (family) and to annotate location. In a second step, these annotations are interpreted by Jittac. There, the user can utilize the feature filter functionality to inspect locations of dependencies between features at the architectural level. This means that users can analyse which features are implemented in a particular architectural module, how features are scattered across several modules, and how dependencies between features contribute to dependencies between the modules.

**Jittac**

Prerequisites

* See Sec. 2.14
* No further scenario-specific prerequisites

Execution

* First, the user has to annotate, either manually or by using appropriate feature location techniques, the system’s codebase according to the format provided and supported by FLOrIDA.
* Second, the user has to create an architecture model in Jittac and to build the system which initiates parsing of the source code and the feature annotations.

Results

* Jittac provides now a visualization of feature dependencies in form of Sankey diagrams illustrating the dependencies between features that contribute to a previously selected architectural dependency.

**FLOrIDA**

Jittac is just using a specific data format for feature annotations defined by FLOrIDA (see also D7.4) such that FLOrIDA is not even strictly required but a correctly annotated codebase. For more information on FLOrIDA, see D4.1).

* + 1. Benefits

Combining these tools allows software architects and developers to better understand locations of features and dependencies between features without digging into source code immediately. Instead, such information is presented in an architectural representation of the system under investigation, allowing users to focus on features and their dependencies first without having to deal with source code details. This functionality supports users in isolating features into modules, fostering easier migration to a software product line.

* 1. Analyse Models and Extract Features Workflow

Involved tools are FLiMEA and BUT4Reuse.

* + 1. Scenario for use

This integration is aimed at providing the most efficient way to locate feature depending on the nature of the models used as input. Specifically, this integration will be based on a decision maker, which will allow developers to determine the best tool for a specific scenario or if one of the tools should be used to refine the results of the other one.

* + 1. Workflow

Involved tools and actions of this workflow are shown in Figure 16.

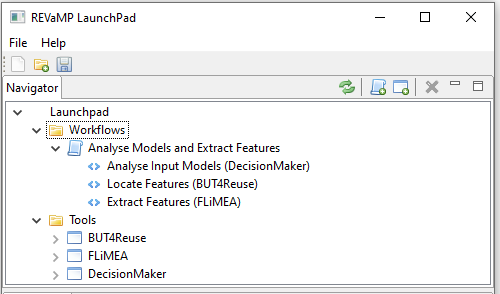


Figure 16 Analyse Models and Extract Features Workflow

**FLiMEA**

Prerequisites

* As described in Section 2.4, both the general and the installation FLiMEA prerequisites have to be satisfied

Settings

* The tool contains a view to configure the executions or experiments performed. Specifically, through this view, the users can configure the preprocessors to manage natural language and the setup of the evolutionary algorithm such as the population, the genetic operations, or the fitness function.

Execution

* After configuring FLiMEA through the configuration view, execute FLiMEA as any other Eclipse plug-in

Results

* A CSV file containing the report with the results using four different metrics: precision, recall, F-measure, and Mathew Correlation Coefficient (MCC).
* A general view showing the best realization (model fragment) located for a feature.
* A detailed view comparing the best realization (model fragment) found against the oracle (correct solution known beforehand) and the other realizations found.
  + 1. Benefits

Combining these tools allows developers to make a decision about which tool should be used in a specific scenario to obtain the best results. This mitigates the difficulty to check each scenario through both tools to obtain the best results and provides an easy way to benefit from both tools equally.

1. Toolsets

For work package 5 and work package 6 toolsets were delivered. Each one focused on verifying a specific asset class, useful at a specific abstraction level and at a specific software engineering process stage. Therefore, the output of each tool in these toolsets does not aim at producing an input directly useful as input to another tool of the toolset. While each tool provides a service that is a complementary to those provided by the other tools in the set, there is no use case for a direct interaction between them. This was one of conclusions resulting from WP2.

* 1. PL Co-Evolution Toolsets
     1. PL Problem Space – Solution Space Co-Evolution (PSS-CE)

Involved Tools: KernelHaven

**Scenario for use**

The Problem-Solution-Space Co-Evolution (PSS-CE) support tool described in Section 2.6 enables the detection of unintended divergences between features defined in the variability model (problem space) and the build rules and code elements (solution space) they (should) control. Control structures, like runtime *if*- or preprocessor *#ifdef*-statements with conditions referencing features of the variability model, typically realize this control over build rules and code elements as shown by an example from the Linux kernel in Figure 10. The source code artifact in the upper left part realize capabilities for performance events for the x86-architecture of the Linux kernel. The preprocessor statement in lines 780, 782, and 784 in *perf\_event.h* references a symbol, which is actually a feature. This is indicated by the prefix *CONFIG\_*, which differentiates general source code symbols from features defined in the variability model. The variability model artifact in the upper right part shows this definition, where the keyword *config* (lines 9 and 1811) indicates new feature definitions. Thus, if a 32-bit kernel for a x86-architecture is configured, the *X86\_32* option has to be selected, which leads to the presence of the code in the *#ifdef*-block of the source code artifact (line 781). The deselection of *X86\_32* in turn leads to the presence of the code in the *#else*-block (line 783). The conditional execution of build rules, like the one in line 195 in the build artifact in Figure 10, works in a rather similar way.



Figure 10: Example of problem-solution-space mapping from the Linux kernel

The example in Figure 10 shows an optimal relation between problem and solution space artifacts: all features defined in the variability model are referenced by at least one artifact of the solution space, while each solution space artifact references a defined feature of the variability model. However, assuming that the *X86\_32* option in the variability model (lines 9ff) is not defined, parts of the source code would never be selected (line 781). This may be an unintended divergence between the problem and the solution space, which the PSS-CE support tool, besides others, automatically detects as shown in Table 1.

Table 1: Example result of the PSS-CE support tool

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Problem Space Symptom | Solution Space Symptom | Correction |
| Undefined Variable Divergence | “CONFIG\_X86\_32” not defined in the variability model | “CONFIG\_X86\_32” used to constrain presence of code element(s) “./arch/x86/events/ perf\_event.h[780-782]” | Define “’CONFIG\_X86\_32” in the variability model  OR  Remove references to “CONFIG\_X86\_32” in the condition controlling the presence or absence of the following code element: “./arch/x86/events/ perf\_event.h[780-782]” |

The result of the PSS-CE support tool provides the necessary information to inspect and, if unintended, correct the identified divergences. Therefore, Table 1 shows the type of the divergence, its symptoms in the problem and solution space causing that divergence, and provides proposals for their correction.

Benefits

The Problem-Solution-Space Co-Evolution (PSS-CE) support tool provides automatic detection and proposals for corrections of unintended divergences as described above. In its recent version, the tool also supports incremental support, which enables the reduction of the analysis effort to changes applied to a software product line during its evolution. Hence, in large-scale software product lines with up to multiple thousands of features and references, the tool supports the developers in ensuring that their changes to variability information in variability model, build, and code artifacts are correct.

* + 1. PL Problem Line – Variant Co-Evolution (PLPV-CE)

Involved Tools: Jess

Scenario for use

The Product-Line-Product-Variant Co-Evolution (PLPV-CE) support tool described in Section 2.7 enables the automated migration of semantically related lines of code, like functions or features, from a product line to a product variant and vice-versa. Figure 11 shows a typical usage scenario for that capability. At a specific point in time (t0 at the left side of Figure 11) a particular product is derived from the product line by instantiating the variable parts of that product line. As a result, the product at t0 only contains the code that realizes feature A. Over time, both, the product line and the product, evolve independently resulting in (cool) new features that are only available in either the product line or the product (t1 at the right side of Figure 11). At that point in time, the developers may decide that transferring these new features from the product line to the product (or vice-versa) may significantly enhance the target software, e.g., as it fixes bugs in the derived product or introduces functionality that also other products of the product line benefit from. However, manually detecting all semantically related lines of code (a semantic unit) that are required to ensure that such a transfer results in a correct state of the product line or the product is tedious and error-prone.



Figure 11: Example of product-line-product-variant co-evolution scenario

The result of the PLPV-CE tool is a patch-ready representation of a semantic unit, which the developers define by using the graph representation of the input software produced by the tool. A developer therefore only needs to apply the tool on the current state of the product line or the product, which then produces a graph with selectable nodes. Each node represents certain code element and, hence, can be used as a starting point for identifying related lines of code by traversing the graph edges. The collection of all related graph nodes is transformed into code files and directories that match the structure and content of the software the tool is applied to. However, this result only contains the necessary parts to ensure a correct transfer of the desired feature or function (the selected start node in the graph). All other code fragments are excluded such that developers can merge only the relevant parts to the target software.

Benefits

The Product-Line-Product-Variant Co-Evolution (PLPV-CE) support tool provides an automated solution for the task described above. It guarantees completeness of the resulting semantic units, which represent all semantically related lines of code for a particular feature or function that shall be transferred from a product line to a product or vice-versa. The developers can rely on the automated semantic unit identification of the tool and receive a patch-ready representation of such a unit for merging. This significantly improves recurring situations during the independent evolution of software product lines and derived products, where improvements or enhancements should be made available for related software as well.

* + 1. PL Hardware – Software Co-Evolution (HWSW-CE)

Involved Tools: SIMULTime

Scenario for use

The Hardware-Software Co-Evolution tool (HWSW-CE) support tool described in Section 2.12 enables producing fast and accurate timing estimations that are essential when evolving software or hardware (or both of them) of embedded systems. For a reference SIS PL, a timing estimation predicts the execution time of a SW program considering its execution on a specific HW platform. The resulting timing estimations are important metrics when a SIS PL is subject to non-functional requirements as timing requirements.

The tool supports the three identified evolution scenarios represented in Figure 18:

1. Evolved HW: This scenario considers the possibility of timing analysing a SW program that has to be executed on multiple HW platforms (e.g. porting the system to a new HW platform). In this case, the tool enables accessing the execution time of all the different variants by running only one context-sensitive timing simulation.
2. Evolved SW: Often, the SW contains customizations that are enabled at run-time via specific configurations. The tool supports this kind of SW evolution by performing the timing analysis process only one and consequently producing all the necessary timing estimations for the run-time configurations to analyse via context-sensitive simulations. Every simulation allows to access only one specific SW run-time configuration.
3. Evolved HW and SW: This scenario considers both the previously described scenarios. In fact, the tool allows accessing the execution time of a SIS PL considering the run-time variability in a software program that can be executed in multiple hardware platforms.

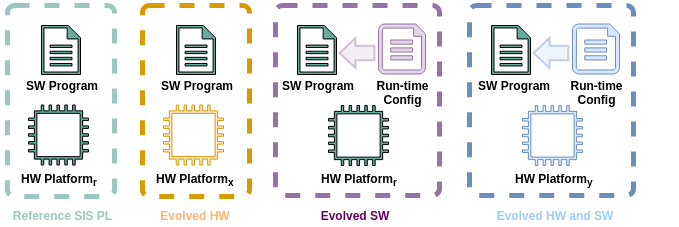


Figure 18: Examples of Hardware-Software Co-Evolution

Additionally, the tool can be integrated in MATLAB/Simulink. The integration allows considering timing effects of an HW platform on SW models developed in Simulink. The timing effects are directly visualized in the native Simulink simulations.

Benefits

The Hardware-Software Co-Evolution tool (HWSW-CE) support tool represents an essential solution for the scenarios described above. It provides fast and accurate timing estimations for the execution time of a SW programs by running context-sensitive timing simulations. The timing estimations are a valid support for accessing the execution time of SIS PLs in case of SW containing run-time variability and evolving HW platforms.

* 1. PL Verification Toolsets
     1. Consistency of Variability Models and Code Artefacts

Involved Tools: KernelHaven

Scenario for use

KernelHaven is a highly configurable analysis infrastructure as described in Section 2.2. A specific setup of this infrastructure enables the analysis of configuration mismatches in a software product line, which aims at consistency checks between the variability model and code artifacts. In particular, it detects consistency problems arising from the combination of the defined configurations in the variability model and their realization in the code as illustrated in Figure 12. The feature diagram (variability model) in the left part of that figure contains two independent features. Both features are optional and are not connected by any constraints. Thus, the configuration space defined by that feature diagram contains four valid configurations: no selection, the single selection of feature *A*, the single selection of feature *B*, and the combined selection of features *A* and *B*. The configuration will be saved as a set of *#define*-statements in a C-header file *fmConfiguration.h*, which is included in the code file in the right part of Figure 12. The variability inside this artifact is managed with preprocessor statements and we assume no variability contained in the build space for this example. Hence, each configuration of the variability model will lead to a syntactically correct product, which will compile successfully. Further, all variable parts of the code can be selected and unselected and, thus, are neither dead nor undead. However, the structure of the variability inside the variability model differs from the structure inside the code, because of the additional dependency between the two *#ifdef*-statements (due to the nesting of *#ifdef B* inside *#ifdef A*). In particular, while four different configurations exist in the variability model, only three different code configurations exist as the single selection of feature *B* results in the same code as if no feature is selected.



Figure 12: Example of configuration mismatch

The example in Figure 12 illustrates situations, which may lead to unforeseen problems, like the derivation of a product without feature B although it is selected. Further, such problems unnecessarily increase the complexity of the configuration process, e.g., by redundant configurations. The configuration mismatch analysis automatically detects exactly such mismatches between the defined configurations in the variability model and the realized configurations in code. The required KernelHaven configuration as well as the plug-ins realizing this analysis are part of the release bundle and, hence, can be applied out of the box. The result is a table containing the identified mismatches by the their origin (file, start and end line number) as well as their conditions and dependencies leading to the mismatch.

Benefits

The detection of configuration mismatches provides the basis for a manual review to fix the underlying problem. For instance, this may be the identification of domain knowledge, which was not covered before. In this case, a new constraint may be added to the variability model. Alternatively, the dependency leading to the derivation of the same product for different configurations is not intended. In this case, the developers may revise the code artefacts to facilitate the correct product derivation.

* + 1. Support for Testing

Involved Tools: VEXA, SIMULTime, DragonflyME

Scenario for use

The goal of the tool chain is to test legacy code of a SIS-PL with the help of virtual prototypes, meaning a system simulation that takes the hardware platform into account. Such simulations can be used to verify future developments of the SIS-PL, e.g., the transfer to a new platform or the extension with new features.

The creation of the virtual prototype is not automated, but the tools support the user in its creation. The tool VEXA can be used to analyses existing legacy code and support the user in the design of a corresponding structure of the virtual prototype. With this the user can specify the general structure within the DragonflyME with the help of a graphical user interface. With this model the structural code of the virtual prototype can be generated and manually linked to the legacy code. Another feature of VEXA is to export the variation points in the legacy code. The variation points have to be provided by the virtual prototype, via the dynamic configuration approach, to test the possible variants. The required code to parse and create the configuration files can be automated with the DragonflyME. Besides the variation points the VEXA tool provides the constraints of each variation point, enabling the DragonflyME to calculate global constraints of the variability and therefore to generate all possible test instances of the SIS-PL. By the combination of this two tools, untimed, functional simulations of the SUT are possible. With the help of the tool SIMULTime it is possible to estimate the resulting software execution time in combination with different hardware platforms. The tool measures the application on different hardware platforms and creates a timing data base (TDB). This TDB can be used to execute the legacy code on a host system, namely in form of a virtual prototype, and calculate the resulting execution time on the target platform. Because the TDB supports the execution time estimation of non-monitored execution orders, the approach can be used to support the flexible configuration approach targeted by the virtual prototype and the DragonflyME. After the creation of the virtual prototype and the TDB the DragonflyME generates test cases of the PL to be executed with the help of a timed, functional simulation of the HW/SW system to analyze different aspects of the system.

Benefits

By combining the tools VEXA, SIMULTime and DragonflyME the user is supported in the creation and execution of timed, simulation in the context of a variable SIS-PL. Based on the intelligent test case generation approach not all variants have to be tested, instead the tool provides the user suggestions of interested test cases. Additionally with tools like SIMULTime the timing behavior of the HW/SW system on the target platform can be considered, with only executing a small set of the possible variants. Both the test case sampling as well as the timing prediction enable the testing of complex SIS-PL with the help of virtual prototypes.

* + 1. Metric-based Smell Identification

Involved Tools: MetricHaven (KernelHaven)

Scenario for use

MetricHaven is a particular extension of KernelHaven (cf. Section 2.2), which supports the implementation and application of software product line metrics. In essence, it is an analysis plug-in for the main infrastructure of KernelHaven, which provides various metrics and their variants. Hence, the general usage scenario of MetricHaven is to measure qualitative aspects of a software product line, like the complexity of variations points in the code or the scattering degree of features to inspect whether they are well modularized or spread across the code base. The results of such measurements provide hints to potential (code) smells, e.g., as a variation point is more complex than it should be and, hence, should be inspected for potential (configuration) errors.

Benefits

MetricHaven provides configuration support for already implemented metrics, resulting in 23,342 metric variations that can be applied out of the box to measure qualitative aspects of software product lines. While many metrics exist, which address code or variability separately, MetricHaven extensively supports combining information from code files and variability models. Further, the tool enables the combination of well-established single system metrics with novel variability-aware metrics, going beyond existing variability-aware metrics. Hence, MetricHaven provides developers as well as engineers with a plethora of qualitative information in terms of most prominent single system and variability-aware code metrics.

* + 1. Incremental Verification Support

Involved Tools: KernelHaven

Scenario for use

The incremental verification support is realized by a specific KernelHaven plug-in, which extends the main infrastructure by capabilities that reduce the analysis effort to those artifacts affected by changes to information relevant to the analysis. In the domain of software product line analysis this information is typically the variability information realizing the configuration and adaptation of artifacts for specific products. For example, the incremental variant of a dead code analysis would only re-analyze those changes, which affect the presence of conditional code blocks. In general, the purpose of this analysis is to detect preprocessor statements, which define conditions that are not satisfiable. Hence, the code blocks enclosed by such statements are never part of any product of the software product line and called dead. These dead code blocks typically arise from inconsistencies between valid configurations defined in the variability model and the conditions controlling the execution of build rules as well as the presence or absence of code blocks. If the combination of the presence condition of a code block (e.g., defined by an enclosing preprocessor *#ifdef*) and all valid configurations is not satisfiable, there exists no valid product, which includes that code block. In the incremental variant, the analysis only re-analyzes those code blocks for which the presence condition has changed or all code blocks, if the variability model or the build rules have changed. In the former case, the unchanged information of the variability model and the build rules can be reused, while in the latter case the information about the code blocks can be reused.

Benefits

The incremental verification support reduces the analysis effort to those artifacts actually affected by changes, e.g., during software product line evolution. Hence, it provides faster feedback to developers compared to the non-incremental variant of the same analysis. Further, the feedback concerns only the changes recently applied to the software product line, which reduces the effort for inspecting the analysis results and draws the attention to the latest, yet erroneous parts the developers worked on.

1. <https://github.com/neo4j/neo4j> [↑](#footnote-ref-1)
2. Liebig, Jörg, et al. "An analysis of the variability in forty preprocessor-based software product lines." Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1. ACM, 2010. [↑](#footnote-ref-2)
3. Nadi, S., Berger, T., Kastner, C., & Czarnecki, K. (2015). Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. IEEE Transactions on Software Engineering, 41(8), 820–841. https://doi.org/10.1109/TSE.2015.2415793 [↑](#footnote-ref-3)
4. Lea Gerling and Klaus Schmid. 2019. Variability-Aware Semantic Slicing Using Code Property Graphs. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (SPLC '19), ACM, 65-71. DOI: https://doi.org/10.1145/3336294.3336312 [↑](#footnote-ref-4)