ITEA Office
High Tech Campus 69 - 3
5656 AG Eindhoven
The Netherlands

T + 31 88 003 6136
E info@itea3.org
W www.itea3.org

ITEA 3 is a EUREKA strategic ICT cluster programme

D6.2: Asset Verification Automation Technologies PoC 1.0 Implementation

Author	Affiliation
Jacques Robin	Université Paris 1 Panthéon-Sorbonne (UP1PS)
Damir Nesic Jonas Westman, Dilian Gurov	Kungliga Tekniska Högskolan
Sascha El-Sharkawy	Stiftung Universität Hildesheim
Sebastian Reiter, Anton Paule	FZI Forschungszentrum Informatik
Ali Parsai	Universiteit Antwerpen
Matthieu Pfeiffer	Magillem
Borja Lopez, Elena Gallego	Knowledge-Centric Solution The Reuse Company
Anne Augustin, Linda Schmuhl	Model Engineering Solutions

Last edited by Jacques Robin on 15/01/2019

Executive summary

D6.2 delivers the first versions of the implementation of REVAMP's verification automated tools which architectural designs were delivered in D6.1. These tools are currently stand-alone tools, not integrated in a tool-chain. This integration will be delivered in D6.4.

This tool set currently include 8 tools, 5 research Proof-of-Concept (PoC) demonstrators developed by academic partners and 3 commercial tool prototypes developed by industrial partners. Among those tools 2 perform formal verification, while the others rather focus on computing various quality analysis metrics. One tool focuses on analysing product line variability models, while the others focus on analysing reusable product line artefacts such as requirements model, hardware models, source code and test sets as well as the relationships holding between those artefacts themselves and between those artefacts and variability models.

In this deliverable we quickly present the implementation of each of these tools in turn. For each one we remind its objectives, the kind of input artefacts that it accepts as input to verification, the kinds of properties that it can verify on these artefacts and the kinds of verification results and quality metrics that it is able to produce as output. We also clearly identify any discrepancy that may exist between the functionalities and architecture of the tools as specified in the design model delivered in D6.1 and those actually implemented in the first implementation version now delivered in D6.2. We explain the causes of these discrepancies and whether they are planned to be resolved in the future D6.3 design 2.0 and D6.4 implementation 2.0 to be delivered in 2019.

We also state what programming language, libraries and application frameworks were used to implement them, and what versions of these. Finally, we indicate where to find a copy of the code of these tools to download for testing purposes, and where to find training material to learn how to use them.

Table of Contents

Acronyms

FZI	Forschungszentrum Informatik
KCS	Knowledge-Centric Solution
KTH	Kungliga Tekniska Högskolan (Royal Institute of Technology)
MES	Model Engineering Solutions
PL	Product Line
PLE	Product Line Engineering
PoC	Proof-of-Concept
RT	Round-Trip
RQS	Requirement Quality Suite
SIS	Software-Intensive System
SOI	System Of Interest
SUH	Stiftung Universität Hildesheim
SUT	System-Under-Test
TRC	The Reuse Company
UA	Universiteit Antwerpen
UP1PS	Université Paris 1 Panthéon-Sorbonne

1. Overview of the deliverable

D6.2 delivers the first versions of the implementation of REVAMP's verification automated tools which architectural designs were delivered in D6.1. By verification here we mean verification in the broadest sense of the activities carried out in the V&V (Verification and Validation) step of a SIS engineering lifecycle. Its scope is thus much wider than formal verification of the SIS properties, including other forms of V&V such as quality analysis metric computation and testing.

Table 1 below gives an overview of REVAMP's V&V automation tools, including their name, the REVAMP partner providing its input, output and implementation platform.

Class	Input Asset kind	Verified properties	Name	TRL	Provider	Implementation Platform
Formal Verification	Feature model	Presence of defects	VariaMos	PoC	UP1PS	Java archive
	C code, formal requirements	Conformity of C code to formal requirements	KTH C code verifier	PoC	KTH	C# and F#
	C code, build information (e.g., make files), variability model (e.g., feature models)	Consistency of code and variability model	KernelHaven	PoC	SUH	Java
Quality metric computation	Requirements following domain ontology constrained semi-natural language templates	Correctness Consistency Completeness	RQS	Commercial product	KCS-TRC	C#.NET
	C code, build information (e.g., make files), variability model (e.g., feature models)	Smell detection	KernelHaven	PoC	SUH	Java
	HW legacy assets, products description, SoC, architecture, IP	Consistency and completeness of assets	Magillem Crystal Bulb	Commercial product	MDS	Web application server (Spring Boot + Angular) Coded in Java

Class	Input Asset kind	Verified properties	Name	TRL	Provider	Implementation Platform
	description and features, configurability, memory maps					
	IP-XACT and HDL files	Syntactic and semantic checkers	Magillem Platform Assembly	Commercial product	MDS	Eclipse based tool
Testing	MATLAB Simulink model, software requirement specification	Requirement conformity	MTest	Commercial product	MES	MATLAB
	Java Software (production code, test code, build and test environments)	Test Quality	LittleDarwin	OpenSource Product	UA	Python
	Virtual prototype (SystemC/C++-Code) of the SPL, variability model (e.g., feature models)	Test case generation	ViTAmineE	PoC	FZI	Java

Table 1: REVAMP V&V automation tool set overview

In the rest of this document, we elaborate the key properties of the first implementation of this tool in order. We start presenting by the research Proof-of-Concepts (PoC) tools before presenting the commercial tools. Within each of these categories, we start by presenting the tools that perform some formal verification, before presenting those performing only quality analysis metric computations.

2. Proof-of-Concept tools

2.1. Formal verification tools

2.1.1. VariaMos (UP1PS)

2.1.1.1. Objectives summary

VariaMos (Mazo, et al., 2015) aims to integrate the following services:
- Graphical edition of variability models following the built-in VariaMos product line feature model meta-models (abstract syntax and concrete syntax meta-models)
- Interactive product design space exploration by iterative selection of features by the designer from the options represented in the variability model, followed by automatic elimination of feature options by cross-feature constraint propagation
- Automated product derivation from the point selected in the design space
- Feature model verification by detection and localization of defects in the feature model (Rincón, et al., 2015)

2.1.1.2. Input artefacts

The verification services of VariaMos take as input a feature model (problem space) and an asset model (solution space) both constructed and linked together with the VariaMos feature modelling graphical edition service;

2.1.1.3. Verified properties

The properties verified are the absence of the following set of defects that can be encountered in feature models:
- **Void feature model**, *i.e,* feature model for which no valid configuration exists due to over-constrained constraint set
- **False product line**, *i.e.,* a feature model for which only a single valid configuration exists and therefore does not represent any genuine variability;
- **Dead features**, *i.e.,* features that due to other constraints cannot appear in any valid configuration;
- **False optional features**: *i.e.,* features that are defined as optional in the feature tree and yet due to other constraints appear in all valid configurations;
- **Wrong cardinalities**, *i.e.,* locally specified cardinality upper and lower bound for a set of sub-features in the feature tree that, due to other constraints are incorrect, for example a locally specified lower bound that is lower than the minimum number of the sub-features present in any valid configuration or a locally specified upper bound that is higher than the maximum number of such sub-features present in any valid configuration

2.1.1.4. Computed quality metrics

The quality metrics currently computed on a feature model are:
- **Number of valid products** measures the size of the valid configuration space;
- **Product line homogeneity** is defined as $1 - \#f/\#p$ where $\#f$ is the number of features appearing in a single product divided and $\#p$ is the total number of valid products derivable from the feature model; its value is in [0,1], 0 corresponding to the case where every product has a single unique feature and 1 corresponding to the case where there is no variability.

- **Product line variability factor** measures the degree of independence of the features it is defined as $\#p/2^{\#f}$ where $\#p$ is the total number of valid products derivable from the feature model and $\#f$ is its number of features excluding the root feature.
- **Extra constraint representativeness** measures the density of cross-tree constraints among features, defined as $\#c/\#f$ where $\#c$ is the number of features involved in a cross-feature constraint and $\#f$ is the total number of features in the tree;

2.1.1.5. Implemented elements from design model delivered in D6.1

The current version of VariaMos 1.0.1.18 differs from the design model delivered in D6.1 in several subtle ways. Let us review them in turn for the three main sub-components of VariaMos:

- The *VariaMos Model Editor*
- The general *VariaMos Variability Model Verification* component and its specialization *VariaMos Feature Model Verification* component
- The *VariaMos Product Line (PL) Configuration* component

Concerning the *VariaMos Model Editor,* the main discrepancy between the designs modelled in figure 11 of D6.1 and the implementation delivered in D6.2 is that the latter does not separate *model* edition functionalities from the *diagram* edition functionalities in two different components. Consequently, the current *User Interface (UI)* does not include as idealized at design-time a model element hierarchy editor pane separated from the various diagram edition tabs as asset modelling tools such as Modelio provide. This is due to the choice of reusing the open-source JGraphX[1] library for the UI which is a powerful *diagramming* UI tool but not a full-fledge modelling tool providing this distinction out-of-the-box. The distinction must thus be programmed around JGraphX rather than leveraging it. This is a major and complex implementation effort that could not be delivered for D6.2. It will be considered for inclusion in D6.4. As a result, the *showElt* and *hideElt* operations of the *VariaMos Diagram Editor* component of figure 11 in D6.1 are not yet implemented as distinct from the *newElt* and *delElt* operations.

Others yet unimplemented operations include *cloneProp and mvProp* from the *VariaMos Model Editor* component, which would allow cloning and moving selected properties from one model element to another by a dragging action. This would require be able to show all properties of every element on the diagram which is also not currently supported. Some properties are only displayed and edited in the property editor but not in the diagram.

In terms of diagram layout operations of the *VariaMos Diagram Editor*, only the translate operation is currently implemented. The other such operations in figure 11 of D6.1 *rotate*, scale, align, *equiSpace* and *equiSize* to respectively rotate a diagram element, scale it, align multiple elements along a line, and make their size or the space between them uniform will be considered for D6.4. They provide convenience for precise graphical layout. But they also risk of overcrowding the menu bar with an overwhelming number of options, so there is a usability trade-off involved. We will base our decision to invest the resources needed to add them or not from feedback from VariaMos users within the REVAMP^2 consortium.

Concerning the models of the generic *VariaMos Variability Model Verification* component and its specialized *VariaMos Feature Model Verification* component shown in figure 14 of D6.1, all their operations were implemented except for the *redundant constraint* verification option. This option was not prioritized since, in contrast to all the other verification option, it does not identify a semantic defect that can result in the generation of an invalid configuration. It is merely identified

[1] https://github.com/jgraph/jgraphx.

as a presentation conciseness defect, which is why its implementation was left for the next iterations and delivered in D6.4.

Concerning the model of the *VariaMos PL Configuration* component shown in figure 12 of D6.1 all operations are implemented.

Since the main original contribution of VariaMos in the REVAMP toolchain concerns the verification automation WP6 work package, we decided to prioritize implementing the verification automation operations of the design model of D6.1 over the completeness of the feature model UI editor operations.

2.1.1.6. Implementation technologies

Programming language: VariaMos is implemented in Java 8 and it relies on SWI-Prolog 7 as the underlying constraint solver.

Application frameworks: VariaMos UI is implemented using the JGraphX open-source diagramming library which itself relies on the Java Swing GUI widget toolkit

Data integration facilities: None so far. VariaMos can import and export files in formats that are specific to VariaMos:
- .vmum files to import and export variability models;
- .vmsm files to import and export variability language abstract syntax meta-models;
- .vmom files to import and export variability language operational semantics meta-models
- .conf to import and export a (partial or complete) configuration of a variability model

.vnum, .vmsm and .vmom files are XML files following the convention of the JGraphX to serialize a JGraphX graph into an XML file. The most nested levels in these XML files contain XML elements that correspond to the meta-classes and meta-attributes of the meta-models of a VariaMos variability model, a VariaMos syntactic meta-model or a VariaMos operational semantics meta-models tree schemas. This approach was the simplest to implement but present the drawback of not separating these meta-models from their graphical representation as a JGraphX graph. The .conf file is a JSON flle with one value for each option in the variability model.

Control integration facilities: None so far. VariaMos is currently a monolithic program that can only be used through it graphical UI. Overcoming this limitation to allow VariaMos services to be called programmatically from outside should be the main focus of D6.4.

2.1.1.7. Implementation and documentation location

The source code of the current implementation. **VariaMos 1.0.1.20** can be downloaded at the following URL: https://variamos.com/home/variamos/configuration/

A short video tutorial of the feature model verification and configuration can be found at the following URL: https://www.youtube.com/watch?time_continue=34&v=VEvROmikSnY

2.1.2. Kernel Haven (SUH)

2.1.2.1. Objective summary

KernelHaven is an open infrastructure for Software Product Line (SPL) analysis. It is intended both as a production-quality analysis tool set as well as a research support tool, e.g., to support researchers in systematically exploring research hypothesis. For flexibility and ease of experimentation KernelHaven components are plug-ins for extracting certain information from SPL artefacts and processing this information, e.g., to check the correctness and consistency of variability information or to apply metrics. A configuration-based setup along with automatic documentation functionality allows different experiments and supports their easy reproduction.

2.1.2.2. Input artefacts

The core components of KernelHaven are three extraction pipelines, the data processing, and a pipeline configurator as illustrated in Figure 1 and described in this section.

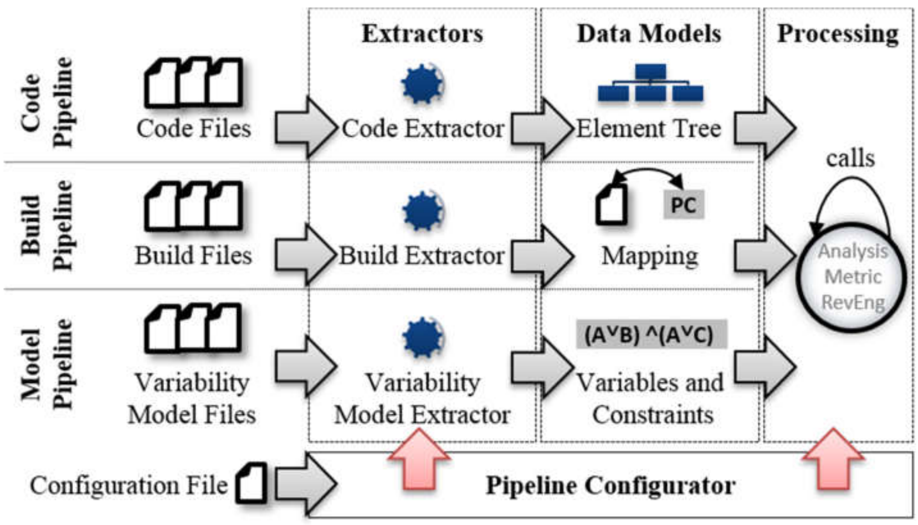

Figure 1: KernelHaven architecture.

Each *extraction pipeline* extracts and provides information of a particular type of artefact typically considered in variability-based analyses. The extractors may be exchanged to allow extraction from different file formats:

- The **code pipeline** in the upper part of Figure 1 extracts information from code files or files used for code generation. The result is a set of element trees. An element tree represents a single code file and provides information about the available code elements on different levels of abstraction.
- The **build pipeline** in the middle of Figure 1 extracts and provides information from build files. The result of this extraction is a map of files and their presence conditions (PC in Figure 1). These conditions define constraints, which must be satisfied to compile and link or, in general, build a specific (set of) file(s).
- The **(variability) model pipeline** in the lower part of Figure 1 translates information from variability model files into a list of features and propositional formulas. They represent the features and constraints, which define the planned products of the SPL.

2.1.2.3. Verified properties

The basic infrastructure of KernelHaven does not provide any verification capabilities. Instead, its plug-in architecture allows extending KernelHaven by static analysis and verification concepts. The following verification plug-ins are currently publicly available:

- **DeadCode Analysis**: This is a re-implementation of the dead code analysis as published by Tartler et al. (Tartler, et al., 2011) and realized in the Undertaker tool (Undertaker, 2018). The *DeadCode Analysis* verifies whether configurable code blocks are indeed configurable with respect to the underlying variability model.
- **Configuration Mismatch Analysis**: This is a more general analysis than the *DeadCode Analysis* as it verifies whether configurable code is always intended by the variability model or whether it becomes unconfigurable in some (partial) configurations (El-Sharkawy, et al., 2017).
- **Incremental Verification**: This is a commit-wise *DeadCode Analysis*, which analyses the delta to a previous commit to determine which parts require a new analysis. The result is a much faster *DeadCode Analysis* (~10x speed up) suitable to be integrated into a continuous integration environment.
- **Code Metrics**: This is an extension to KernelHaven, which currently provides seven code metrics in over 7.300 different variations. The novel combination of information from variability model artefacts with dependency information from code artefacts allows detecting complex variability structures, which cannot be detected with existing approaches (El-Sharkawy, et al., 2018).

2.1.2.4. Computed quality metrics

KernelHaven supports seven variability-aware code metrics (MetricHaven, 2018). Most of this metrics may be applied to non-variable code parts, variations points only, or to both kinds of code elements at once to analyse the impact of variability information:

- **Number of Internal/External Variables per Function** (Ferreira, et al., 2016): This metric measures the number of variables or features, which are used to configure a code function. Variations of the metric are to analyse how many variables are used outside of the function, e.g., to include the code file or the function within a code file, or how many variables are used inside the function for fine-grained customizations.
- **Cyclomatic Complexity** (McCabe, 1976): This metric measure the complexity of control structures used in the programming language, variation points, or the combination of both.
- **DLoC**: This metric measures the delivered lines of code (statements, without comments or empty lines) used for implementing a function, the delivered lines of code surrounded by variation points, or the fraction of both.
- **Nesting Depth**: This metric measures the maximum and average nesting depth of statements within control structures, variation points or the combination of both.
- **Fan-In/Out**: This metric measures the number of function calls within a function or how often a function is called by other functions. In addition, function calls may be weighted based on variables used in variations points, which are used to en-/disable the function calls (Ferreira, et al., 2016).
- **Tangling Degree**: This metric measures tangling degree values (number of distinct feature variables used in variation points) for each CPP block with an expression (no else statements) and sums them up for each function.
- **Blocks per Function**: This metric measures the number of (nested) variations points per function.

In addition, each code metric, which operates on variation points, may be combined with information from the variability model:

- **Scattering Degree**: Determines for each feature in how many variation points (i.e., c pre-processor blocks) or code files it is used in.
- **Cross Tree Constraint Ratio**: Determines for each feature in how many constraints it is used in.
- **Feature Definition Distance**: Computes the distance in the file system between usage of a feature in a code file and its definition in the variability model.
- **Feature Types**: Allows specifying weights for features based on their data types, for typed variability modelling techniques (e.g. Kconfig).
- **Feature Hierarchies**: Weights features based on their hierarchy level in the feature model, as deeply nested features are harder to maintain than top-level features (Bagheri, et al., 2011).
- **Variability Model Structures**: Considers the number of edges between features in the variability model, e.g., edges of nesting structures but also edges created through constraints.

2.1.2.5. Implementation in D6.2 vs. Design in D6.1

Since the D6.1 deliverable, KernelHaven was extended as follows:

- **Preparators**: This interface allows applying a normalization to artefacts before their analyzation. For instance, for the analyzation of code artefacts of the Bosch PS-EC product line, a preparatory may be used to transform numerical expressions into Boolean formulas (El-Sharkawy, et al., 2018; Krafczyk, et al., 2018).
- **Extension of Metric Framework**: Further metrics and combinations were implemented in order to support more than 7.300 metric combinations as described in Section 2.1.2.3.
- **DIMACS Importer**: A DIMACS importer was realized to provide import capabilities for pure::variants models. This interface allows a formal analysis, whether modelled constraints of the variability model are consistent to implemented dependencies of code assets.
- **Realization of Incremental Verification Framework**: A framework for supporting the commit-wise verification of product line assets while reusing partial analysis results from previous analyses was implemented. As first prototype, the *DeadCode Analysis* (cf. Section 2.1.2.3) was ported to run as incremental analysis.
- **Oberserver Interface for External Tools**: This extension allows third party tools to use KernelHaven as library and to receive directly desired analysis results via an observer interface.
- **Export to SQLite**: This export capability was implemented to provide additional integration capabilities. This is currently used by ScopeSET to visualize analysis results of KernelHaven.

2.1.2.6. Implementation technologies

Programming language:
KernelHaven is implemented in Java 8 and may be configured through property-files.

Application frameworks:

The KernelHaven infrastructure does not use any third party library, but some of the plug-ins[2] may use third party content:

- **KconfigReaderExtractor**: This extractor make use of the kconfigreader tool[3].
- **UndertakerExtractor**: This extractor make use of the Undertaker tool[4].
- **TypeChefExtractor**: This extractor make use of the TypeChef tool[5].
- **srcMLExtractor**: This extractor make use of the srcML tool[6].
- **KbuildMinerExtractor**: This tool make use of the KbuildMiner tool[7].
- **CnfUtils**: This utility framework uses the following libraries: Apache Commons Lang (v. 2.6), Google Guava (v. 14), Jbool Expressions (v. 1.13), Sat4J (v 2.3.5)
- **IOUtils**: This utility framework uses the following libraries: Apache Commons IO (v. 2.5), Apache Commons Lang 3 (v. 3.6), Apache POI (v. 3.17)

Data integration facilities:
The following data formats can be handled by KernelHaven:

- Comma Separated Values (CSV): CSV-files can be used for importing and exporting data. This may be used to import data during the extraction phase (e.g., variability models), or analysis results (e.g., list of presence conditions) to compute results of consuming analysis units (e.g., computation of feature effects).
- Excel Spreadsheets (XLS, XLSX): Excel documents can be used for importing and exporting data. This may be used to import data during the extraction phase (e.g., variability models), or analysis results (e.g., list of presence conditions) to compute results of consuming analysis units (e.g., computation of feature effects).
- SQLite: SQLite can be used for exporting data.
- DIMACS: Feature models expressed in the DIMACS format may be imported into KernelHaven. This includes pure::variants feature models, which can be converted to the DIMACS format using the DIMACS Exporter tool[8] developed by the Robert Bosch GmbH.

Control integration facilities:
KernelHaven may be used as a library by other Java-based programs. In this case, the calling application may register itself as a processing unit to receive analysis results.

2.1.2.7. Implementation and documentation location

- Sources, the infrastructure, plug-ins as well as pre-packed bundles of KernelHaven may be obtained from its repository: https://github.com/KernelHaven/KernelHaven
- The pre-packed bundles are shipped with a manual, which is also online available as a wiki here: https://github.com/KernelHaven/KernelHaven/wiki

In addition, the following video tutorials are available:

- A short tutorial about its concepts (~5 Minutes): https://www.youtube.com/watch?v=IbNc-H1NoZU

[2] In Kernel Haven writing a plug-in means to create a new JAR, place it in the plug-ins directory, and adapt the configuration to use our extension.
[3] https://github.com/ckaestne/kconfigreader
[4] https://vamos.informatik.uni-erlangen.de/trac/undertaker/
[5] https://ckaestne.github.io/TypeChef/
[6] https://www.srcml.org/
[7] https://github.com/ckaestne/KBuildMiner
[8] Available in the REVAMP SVN folder https://svn.fzi.de/svn/revamp/WP4%20PL%20extraction/tools/PV2DIMACS

- A more detailed video explaining how to use KernelHaven (~23 Minutes): https://www.youtube.com/watch?v=xKde6tPY_jA

2.1.3. KTH C Code Verifier

2.1.3.1. Objective summary

The purpose of the tool is to take as input a C implementation file already annotated with contracts for the functional requirements, and produce as output the same annotated C file but also annotated with auxiliary annotations needed for successful verification. The tool can also then automatically verify this file using VCC as backend.

2.1.3.2. Input artefacts

A C implementation file already annotated with functional requirements (in the annotation language of VCC).

2.1.3.3. Verified properties

The tool can verify that a C implementation adheres to its contract, which includes functional requirements and absence of run-time exceptions.
The result of verification is that either the contract is fulfilled or it is not.

2.1.3.4. Computed quality metrics

Correctness (w.r.t functional requirements)
Termination (in most cases)
Absence of run-time exceptions

2.1.3.5. Implementation in D6.2 vs. Design in D6.1

The design described in D6.1. included also a design of a requirement specification tool, and a tool for translating requirements and generating annotations for them in the C-file. We have a prototype implementation for this, but it is highly domain-specific, and in an early stage of development. We plan to generalize and further improve this implementation.

2.1.3.6. Implementation technologies

Programming language:
C# (version 7.0)
F# (version 4.1)

Application frameworks:
VCC
Standard libraries for C# and F#

Data integration facilities:
Input: C source files (with or without VCC annotations)
Output: C source files (with VCC annotations)

Control integration facilities:
Used from the command line and takes a C source filename as input.
There are flags for whether annotation and/or verification should be performed, as well as for specifying an output filename (optional).

2.1.3.7. Implementation and documentation location

For information on how to use the tool, there is a help flag available from the CLI (command line interface). For information on how to specify functional properties, see the VCC user manual. Tool implementation can be found in the ReVaMP repository together with D6.2 deliverable.

2.2. Quality metric computation tools

2.2.1. ViTAminE/Dragonfly (FZI)

2.2.1.1. Objective summary

Beside formal verification, simulation-based testing is an important part in the qualification of SIS PLs. The main difference between both approaches is that simulation need an input vector to stimulate the system under test (SUT). This can be viewed as verifying a single point of the input space of the SUT. If the SUT is an instance of a PL, the problem is aggravated, because the simulation verifies only one single point/instance of the PL. Summarizing, running one simulation verifies one SIS PL instance with one dedicated input. For an efficient testing, a strategy for selecting these instances (PL and input space) is required. With regard to SIS PL testing, nowadays a set of input vectors is applied to each SIS PL instance separately, resulting in many unnecessarily execution of test cases.
The objective of this PoC is to provide an iterative algorithm to select feasible instances from the input and PL 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 this sampling of the fitness landscape, the PoC selects new test vectors. The assumption is that the fitness landscape is partial continuous, enabling the derivation of the most critical and therefore the subset of representative test cases for the SIS PL, with regard to the objective function. 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 PoC 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.

2.2.1.2. Input artefacts

The main input is the simulation model of the SIS PL. The PoC focuses on simulation models based on SystemC/C++. To support the user and reduce the effort for executing a single simulation, the PoC 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 input 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. This specification is used to describe the valid search space used for test case generation. Waiting for the approval of pure:;variants' Variability Exchange Language as an interoperability standard, this specification is currently expressed in a Vitamine specific language. The PoC uses the same format to specify the variability of the SIS PL as well as the variability of the input space. Technically there is no differentiation between input variability and PL variability. The current implementation (PoC1.0) utilize a separate file that specifies the degrees of freedom based ona mathematical representation of the search space. In future version the variability specification used in the REVaMP² tool chain should be integrated, as shown in section 'Implementation in D6.2 vs. Design in D6.1'.

2.2.1.3. Verified properties

The verified properties are based on the simulation models. In current studies, the focus is on timing evaluation of PLs. This is possible if the simulation models contain annotations about the expected execution time of the software. Approaches for timing prediction in the context of PLs are researched in WP5.

2.2.1.4. Computed quality metrics

The PoC generates test cases that can be executed to verify the specified properties, such as timing. To support the easy execution of multiple simulation runs, the PoC generates a configuration file that can be read by the simulation during runtime. The configuration file is based on IP-XACT.

2.2.1.5. Implementation in D6.2 vs. Design in D6.1

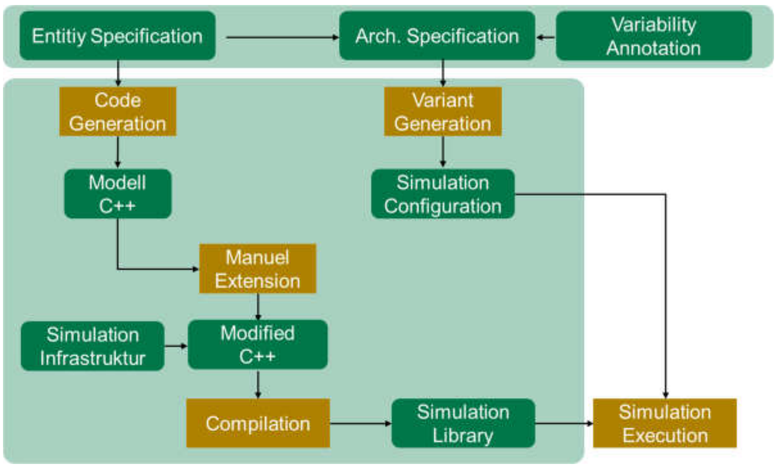

Figure 2: Structure of the ViTAminE/Dragonfly framework

The model of deliverable D6.1 (shown in Figure 35) sketches the overall structure of the PoC. It consists of the *Variability_Specification,* the *Variant_Instance_Generator* and the *VP_Assessor*. In D6.2 a running version of the editor, to specify the simulation model and a configuration file are provided. The required code generators for the SystemC/C++ simulation models as well as for the IP-XACT based configuration files are integrated. With the current version of the PoC it is possible to specify the simulation model of a SIS PL with the help of UML class diagrams. Each class represents one simulation entity that can be instantiated, configured and linked with other simulation entities. Code generators derive structural code, with mock up functions for each simulation entity. Additionally, the configuration file can be specified with an UML composite structure diagram and the required IP-XACT file can be generated. This composite structure

diagram specifies which simulation entities to instantiate, how to configure each and how to interconnect them. In terms of electronic simulations, it specifies the Top-Module of the system. Figure 2 specifies the ViTAminE/Dragonfly framework. On top, the graphical UML-based editor is shown. The UML diagrams specify the simulation model and the simulation instance. The annotation of variability information is subject for future work. Code generation steps are used to derive the simulation model with extensions for runtime configuration. The functional implementation has to be done manually by the user. After the manual extension, the model is compiled, resulting in the so-called simulation library. It is an executable that can be configured with a configuration file during runtime to execute a simulation instance. This simulation library covers all instances of the SIS PL, which can be instantiated with the configuration file. The current focus is on runtime variability, because this can be easily mapped to the simulation library approach. Compile time variability can be used too, but in this case, multiple executables have to be provided or the variation point has to be mapped to runtime variability.. In some cases, the mapping is not straight forward, especially if the different variant cannot be mapped to different name spaces. We are currently investigating these special cases to design a general procedure to transfer compile time variability into runtime variability.
For future versions (> D6.3) the variability specification is integrated into the UML editor. Current discussions within the project consortia indicate the usage of VEL for the specification of variability. Future work will focus on the integration of VEL and the cooperation with the exploration algorithm, which was developed in parallel.

2.2.1.6. Implementation technologies

Programming language:
The PoC is implemented as Eclipse plug-ins that extend the Eclipse Papyrus editor. The Plug-Ins and code generators are implemented in Java.

Application frameworks:
SystemC/C++ for the simulation models (currently used 2.2.3)
Eclipse Modeling Tools Luna 4.4.2
Papyrus 1.0.2
Xtext 2.8.1

Data integration facilities:

- IP-XACT compatible configuration file
- Eclipse UML importer/exporter

Control integration facilities:
No public interfaces. Integration via Eclipse Plug-Ins is possible.

2.2.1.7. Implementation and documentation location

For more information and access to the tool contact sreiter@fzi.de.

2.2.2. LittleDarwin (UA)

2.2.2.1. Objective summary

Mutation testing is a well-studied method for increasing the quality of a test suite. LittleDarwin is designed as a mutation testing framework able to cope with large and complex Java software systems, while still being easily extensible with new experimental components. LittleDarwin addresses two existing problems in the domain of mutation testing: having a tool able to work within an industrial setting, and yet, be open to extension for cutting edge techniques provided by academia. LittleDarwin already offers higher-order mutation, null type mutants, mutant sampling, manual mutation, and mutant subsumption analysis. There is no tool today available with all these features that is able to work with typical industrial software systems.

2.2.2.2. Input artefacts

LittleDarwin takes as input a complete Java Software Environment. This includes production code, test code, test harnesses, build system, and the required external libraries.

2.2.2.3. Verified properties

LittleDarwin assesses the quality of a test suite using mutation testing. Mutation testing is the process of injecting faults into a software system to verify whether the test suite detects the injected fault. Mutation testing starts with a green test suite — a test suite in which all the tests pass. First, a faulty version of the software is created by introducing faults into the system (Mutation). This is done by applying a known transformation (Mutation Operator) on a certain part of the code. After generating the faulty version of the software (Mutant), it is passed onto the test suite. If there is an error or failure during the execution of the test suite, the mutant is marked as killed (Killed Mutant). If all tests pass, it means that the test suite could not catch the fault, and the mutant has survived (Survived Mutant). By discovering the survived mutants, mutation testing provides a method to find the weaknesses in a test suite, and provides targets for the test developer to address.

2.2.2.4. Computed quality metrics

Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of mutation coverage. Mutation coverage is the percentage of the survived non-equivalent mutants to all non-equivalent mutants. A test suite is said to achieve full mutation test adequacy whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100%. Such test suite is called a mutation-adequate test suite.

2.2.2.5. Implementation technologies

Programming language:
LittleDarwin is written in Python version 2.7.

Application frameworks:
LittleDarwin is mainly self-reliant, however, it requires the existence of an underlying local database technology in Python to support Shelve. Shelve requires the existence of either dbm, gdbm, or bsddb to work, and the choice of the database format is performed based on the available interface.
LittleDarwin needs Python version 2.7 to execute. The dependencies are kept at a minimum to increase its portability. LittleDarwin has been tested on Linux, MacOS, and Windows operating systems with success.

Data integration facilities:
LittleDarwin currently supports mutation of Java code. As such, it requires its subjects to adhere to Java grammar syntax up to version 8. It does not run the tests directly, but rather through the

build system, therefore, the only requirement for the test execution is that it can be done via command line.

Control integration facilities:
LittleDarwin recieves all its required configuration options via command line, and therefore can be easily integrated in a continous integration system. All that is needed to automate the execution of LittleDarwin is to run it with the correct command line arguments.

2.2.2.6. Implementation and documentation location

LittleDarwin's source code and manual can be found at https://littledarwin.parsai.net/

3. Commercial tools

3.1. Quality metric computation tools

3.1.1. Magillem Crystal Bulb & EDA tools (Magillem)

3.1.1.1. Objective summary

Crystal Bulb is a platform running on a central server, providing access to information that has to be exchanged daily between the various stakeholders (architects, designers, verification engineers, marketing, SW & tools developer, etc.), through a lightweight client in a web browser. The information is structured in catalogs for products, SoC and IP. Links between objects are automatically created during the population of the database, to check the coherency between data and to allow the navigation inside the catalogs.
The specific information regarding the hardware description of SoC or IP objects is extracted from the IP-XACT description and/or legacy assets in other formats (e.g. Excel, csv, …), for which verification are performed. From Magillem Crystal Bulb, the authorized user will be able to checkout, edit and modify IP-XACT information in the appropriate EDA tools associated with the dedicated API and generators (e.g. “diff and merge” operation with other IPXACT files is realized within the EDA tool environment).

3.1.1.2. Verified properties

- Verification of extracted assets
    - Correctness and completeness of the HW assets (Products, SoC, IP)
    - Consistency of assets to with regards to the defined referential
    - Availability of features in the extracted products
    - Pinout consistency checkers
- IP-XACT compliance
    - Syntactic and semantic checkers

3.1.1.3. Computed quality metrics

Number and type of assets extracted (Products, SoC, IP)
Number of errors & warnings detected during import
Number of inconsistencies detected

3.1.1.4. Implementation in D6.2 vs. Design in D6.1

Regarding the design in D6.1, the implementation has been focused on Magillem Crystal Bulb

3.1.1.5. Implementation technologies

Programming language:
The Magillem tool suite is developed in Java.

Application frameworks:
Magillem Platform Assembly and Magillem Content Publisher are Eclipse-based tools.
Magillem Crystal Bulb is a web application server, based on Spring Boot & Angular frameworks.

Data integration facilities:
Inputs that can be imported:
- IP-XACT files for description of IP and SoC

- CMSIS for description of SoC and register maps
- Legacy of assets and their features can be imported as Excel or csv files
- Word documentation related to HW assets can be imported in MCP

Control integration facilities:

- Magillem Crystal Bulb (MCB) (web application server) includes an API to get information related to HW assets. At present it only allows to read information. The API will be extended to be able to write information while ensuring consistency of assets as a whole.
- Magillem Content Publisher (MCB) (Eclipse based tool) embeds an API that enables to trigger the actions allowed by the tool i.e. creation, verification and consolidation of documentation.
- Magillem Platform Assembly (MPA) (Eclipse based tool) includes an API to handle IP-XACT items, named Tight Generator Interface and

3.1.1.6. Implementation and documentation location

Information about the tools can be found on Magillem website at the following locations:

- Magillem Crystal Bulb

www.magillem.com/products-areas/magillem-crystal-bulb

- Magillem Content Publisher

www.magillem.com/products-areas/magillem-content-publisher

- Magillem Platform Assembly

www.magillem.com/products-areas/magillem-platform-assembly

The Magillem tools involved in REVaMP² are available to partners for use within the context of the project. Tools and the corresponding licences can be requested by email to pfeiffer@magillem.com and license@magillem.com

3.1.2. RQS (KCS-TRC)

3.1.2.1. Objective summary

The purpose of the verification process implemented in VERIFICATION Studio is to provide evidence that a work-product or set of them fulfils its specification and characteristics.

The verification process can be done at work-product level and at specification level. These two levels match the two levels of quality computing that VERIFICATION Studio has:

- Correctness: it’s performed at work-product level. Verifications at this level can be implemented as new type of correctness metric and will be named Correctness Checklist
- Completeness: it’s performed at specification level. Verifications at this level can be implemented as new type of completeness metric and will be named Completeness Checklist

This verification process must be defined manually by filling a quiz composed of checks or checklists. Every correctness checklist will verify a single work-product, meanwhile every completeness checklist will verify a set of work-products by means of completing a checklist related to a specification.

As a result, a verification action is intended to serve as a mean to provide objective evidence that a work-product has (or not) been verified. The verification action is a case of process to be defined at work-product level that don't have to be applied to all items in a specification.

During the whole process, a scoreboard with the results and statistics will be available.

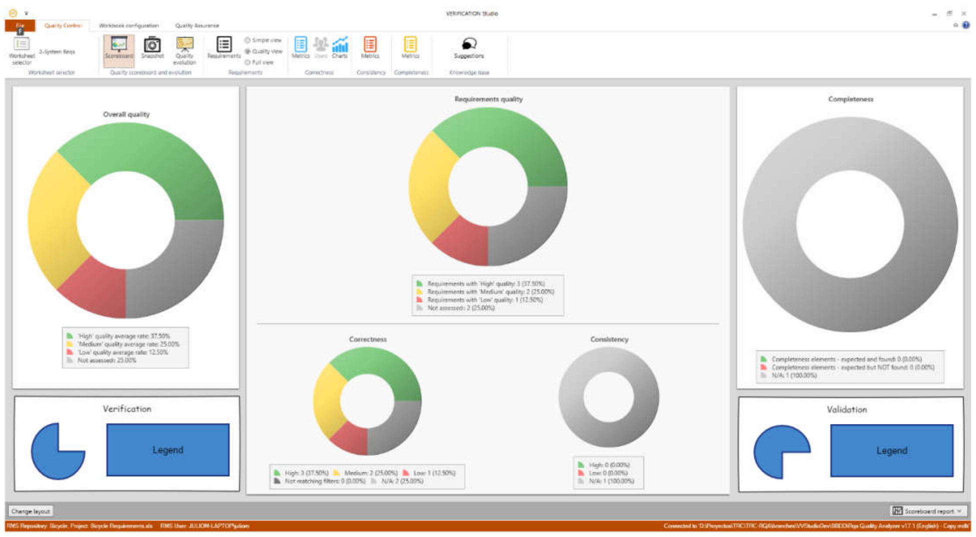

Figure 3. Mockup dashboard for Verification Actions

3.1.2.2. Input artefacts

Requirements (IBM Doors, PTC Integrity, Excel, Reqtify, ReqIF), Models (Rhapsody, Papyrus, Magic Draw), Simulations (Simulink), Feature Models (Pure::variants), or any OSLC-based source. Also, the tool suite supports an XSLT-based connector to map any other format to the internal data model.

3.1.2.3. Verified properties

VERIFICATION Studio analyses work-products from two different perspectives:

- Correctness: it's performed at work-product level. Verifications at this level can be implemented as new type of correctness metric and will be named Correctness Checklist
- Completeness: it's performed at specification level. Verifications at this level can be implemented as new type of completeness metric and will be named Completeness Checklist

3.1.2.4. Computed quality metrics

The Checklist evaluation process will summarize a checklist in a single value depending on the ranges defined.

The results for a correctness checklist metric will be composed by several tabs:

- Statistics:
  - Pie chart about the number of requirement per quality level
  - List question in the checklist and num. of requirements answered (yes, no, n/a & empty) with detail of the requirements in each answer.
- Requirements: the quality for every requirement along with the issues and summary

- Filtering: the graphics and lists of requirements matching and not matching the filters given in the metric configuration

The results for a completeness checklist metric will be composed by several tabs:

- Metric result: Pie chart
- Checklist: list of questions and answers

VERIFICATION Studio will be able to generate Verification Actions (at work-product level) out of the result of all the checklists. Every single verification action is intended to serve as a mean to provide objective evidence that a work-product has (or not) been verified. The verification action has the following attributes:

- Verifiable item: the work-product that is being verified and in which the verification action is being defined
- Source items: A selection of work-products
- Verification technique: One of the following: Inspection, analysis, demonstration, test, analogy or similarity, simulation, sampling, V&V Studio Quality Analysis, other
- Decomposition Level: One of the following: SOI, Subsystem, component
- Objective: Free text
- Activity to perform: Free text
- Expected evidence: The expected results as free text
- Expected numeric result: Number
- Obtained evidence: The obtained results as free text
- Obtained numeric result: Number
- Performed by: The organization of responsible for the verification activity as free text
- Starting date: date
- Ending date: date
- Estimated Time (Days)
- Time (Days)
- Estimated Time (hours/person)
- Labour (hours/person)
- Estimated Funds (€/$)
- Funds (€/$)
- Facility resources: free text
- Verified: Penta-state (YES, NO, SUGGESTED YES, SUGGESTED NO, EMPTY, N/A)
- Verified date: date
- Verified agent: Free text
- Automatic verification rule: a function able to compare the expected and the obtained numeric result in case of guide suggestion for the “Verified” attribute
  - One of the list: <, <=, =, >, >=, !=>=,!=
- Or manual verification using a checklist
- Specific attributes list: a list of names and values

3.1.2.5. Implementation in D6.2 vs. Design in D6.1

From the implementation point of view, D6.1 stands for RQS (Requirements Quality Suite) to analyse correctness properties out of requirements. In D6.2, the input artefacts have been redefined, so that the tool becomes VERIFICATION Studio to analyse not only requirements but also another artefact types (enumerated in 3.1.2.2). Also, the properties to analyse are redefined in D6.2 to include completeness and consistency checklists.

3.1.2.6. Implementation technologies

Programming language:
The whole tool suite is developed in .NET Environment (C# and VB.NET) under the .NET Framework 4.6.1

Application frameworks:
.NET Framework 4.6.1
DevExpress v17.2 for the UI

Data integration facilities:

- ReqIF for requirements import/export
- MS Excel for requirements and ontology import/export
- MS Access, MySQL and SQL Server for the assessment results

Control integration facilities:
VERIFICATION Studio can be used as a native library for other .NET tools.
Interoperability:

- OSLC-KM based services to import/export work-products
- REST service for ontology management
- REST service for quality assessment

3.1.2.7. Implementation and documentation location

VERIFICATION Studio is not released yet. The former RQS v15.1 together with documentation can be found at *www.reusecompany.com*

3.1.3. MTest (MES)

3.1.3.1. Objective summary

The MES Test Manager® (MTest) is a model test management framework that supports ISO 26262-compliant, requirements-based unit testing of Simulink®, Embedded Coder®, and TargetLink® models. The tool supports MiL, SiL, PiL, back-to-back and regression testing, and test case definition methods using measured data, classification trees in CTE/TESTONA, and MTCD (a test specification language for model testing developed by MES).
Beside a precise test stimuli definition, the assessment of the simulation outputs is a key criterion of the test quality. MTest provides the possibility to define both separately. Thus, the test oracle (requirement observers) can be derived directly from the requirements and will be used automatically for the evaluation of the system behaviour in each test case. When using the formalized natural-language requirement language MARS (MTest Assessable Requirement Syntax), these observers are generated automatically. A set of system variants typically shares a lot of requirements, so all derived artefacts from these can be reused easily. By contrast, the variant-specific requirements need to be handled separately. In a first development stage, the variations in the requirement specification are transferred into a MARS specific requirement definition which then can be used in universal requirements valid for all variations of the system. For a specific variant, the corresponding requirement definitions are applied to specialize the requirement observers. The specialized requirement observers then are used for the evaluation of the SUT’s behaviour.

3.1.3.2. Input artefacts

The main input artefact for MTest is the SUT (system under test) in form of a Simulink model. Additionally, requirement specifications are the base for deriving test cases as well as requirement observers. Test cases are the stimulus for the SUT, requirement observers model the expected behaviour of the SUT and evaluate the correctness of the execution outputs according to the respective requirements.
Requirement-based testing assesses the conformity of the system’s behaviour to the corresponding requirement specification. For system variations, variation-specific evaluations are needed to assess the conformity of the selected variant. Computed quality metrics
MTest provides important metrics concerning the test quality. These include particularly the requirements coverage. For each requirement, it is determined whether the simulation of each test stimuli leads to the expected behaviour, represented by the evaluation of the corresponding generated requirement observers. If only one requirement observer fails the requirement is marked as violated. Additionally, coverage metrics on model and code measure the broadness of the test stimuli. In combination, these metrics are a measure for the quality of the executed tests regarding the requirement base.

3.1.3.3. Implementation in D6.2 vs. Design in D6.1

3.1.3.4. Implementation technologies

Programming language:
MTest is mainly implemented using MATLAB’s programming language in combination with different in-house Java libraries. It supports MATLAB from version 7.5 up to the current version.
Application frameworks:

Data integration facilities:

Requirements can be imported from Excel. If they are not compliant to MARS, they need to be transformed into valid MARS requirements first in order to be able to generate the expected output of the SUT (requirement observers) automatically.
Test cases can be defined as classification trees, MTCD scripts as well as MATLAB-specific mat-files containing signal curves. Each will be used to generate test data for simulating the SUT. The simulation outputs and evaluation results are documented in an HTML report and can as well be exported in a Test XML.

Control integration facilities:

3.1.3.5. Implementation and documentation location

If you need further information or require access to MTest please contact linda.schmuhl@model-engineers.com.

Bibliography

Bagheri Ebrahim and Gasevic Dragan Assessing the Maintainability of Software Product Line Feature Models Using Structural Metrics [Journal] // Software Quality Journal. - 2011. - 3 : Vol. 19. - pp. 579-612.
El-Sharkawy Sascha [et al.] Reverse Engineering Variability in an Industrial Product Line: Observations and Lessons Learned [Conference] // Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC '18) - Volume A. - [s.l.] : ACM, 2018. - accepted.
El-Sharkawy Sascha, Krafczyk Adam and Schmid Klaus An Empirical Study of Configuration Mismatches in Linux [Conference] // Proceedings of the 21st International Systems and Software Product Line Conference (SPLC '17) - Volume A. - [s.l.] : ACM, 2017. - pp. 19-28.
El-Sharkawy Sascha, Yamagishi-Eichler Nozomi and Schmid Klaus Metrics for Analyzing Variability and Its Implementation in Software Product Lines: A Systematic Literature Review [Journal] // Information and Software Technology. - [s.l.] : Elsevier, 2018. - In Press, Accepted Manuscript.
Ferreira G. [et al.] Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel [Conference] // In Proceedings of the 20th International Software Product Line Conference (SPLC). - Beijing, China : ACM, 2016. - Vol. I. - pp. 65-74.
Krafczyk Adam, El-Sharkawy Sascha and Schmid Klaus Reverse Engineering Code Dependencies: Converting Integer-Based Variability to Propositional Logic [Conference] // Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC '18) - Volume B. - [s.l.] : ACM, 2018. - accepted.
Mazo R. [et al.] VariaMos: an extensible tool for engineering (dynamic) product lines [Conference] // Proceedings of the 19th International Conference on Software Product Lines. - Nashville, TN, USA : [s.n.], 2015.
McCabe T. J. A Complexity Measure [Journal] // IEEE Transactions on Software Engineering. - [s.l.] : IEEE, 1976. - Vols. SE-2.
MetricHaven KernelHaven: [Online]. - 2018. - https://github.com/KernelHaven/MetricHaven.
Rincón L [et al.] Methods to identify corrections of defects on product line models [Article] // Electronic Notes in Theoretical Computer Science. - 2015. - Vol. 314.

Tartler Reinhard [et al.] Feature Consistency in Compile-Time Configurable System Software [Conference] // Proceedings of the EuroSys 2011 Conference (EuroSys '11). - [s.l.] : ACM, 2011. - pp. 47-60.

Undertaker [Online]. - 2018. - https://vamos.informatik.uni-erlangen.de/trac/undertaker.