ReVaMP²

Project name: 15010 REVaMP²

REVaMP²

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

D3.1: Identification of relevant state of the art

ITEA 3 Call 2

Project start: November 2016
Project end: December 2019

Deliverable	D3.1: Identification of relevant state of the art		
Confidentiality	Public	Type	Document
Project	REVaMP²	Date	2018/02/15
Status	Completed	Version	05
File identifier	REVAMP2_D3.1_v05.docx		

Editors, contact persons	Jabier Martinez and Ali Parsai
E-mail	jabier.martinez@lip6.fr ali.parsai@uantwerpen.be

AUTHORS

Name	Affiliation
Alessandro Cornaglia	FZI, Germany
Ali Parsai	University of Antwerp, Belgium
Andrey Sadovykh	SOFTEAM, France
Anton Paule	FZI, Germany
Antonin Abherve	SOFTEAM, France
Borja López	The Reuse Company, Spain
Carlos Cetina	University San Jorge, Spain
Christian Kröher	University of Hildesheim, Germany
Damir Nesic	KTH, Sweden
Geert Vanstraelen	Macq, Belgium
Gilles Malfreyt	Thales, France
Jabier Martinez	Sorbonne Université, UPMC, France
Jacques Robin	University Paris 1 Panthéon-Sorbonne, France
Jaime Font	University San Jorge, Spain
Jan-Philipp Steghöfer	University of Gothenburg, Sweden
Klaus Schmid	University of Hildesheim, Germany
Marc Bollen	Sirris, Belgium
Matthieu Pfeiffer	Magillem, France
Michael Benkel	ScopeSET, Germany
Mukelabai	University of Gothenburg, Sweden
Olivier Biot	Sirris, Belgium
Raúl Mazo	University Paris 1 Panthéon-Sorbonne, France
Sascha El-Sharkawy	University of Hildesheim, Germany
Sebastian Herold	Karlstad university, Sweden
Sebastian Reiter	FZI, Germany
Serge Demeyer	University of Antwerp, Belgium
Tewfik Ziadi	Sorbonne Université, UPMC, France
Thorsten Berger	University of Gothenburg, Sweden
Tobias Beichter	Bosch, Germany
Uwe Ryssel	pure-systems GmbH, Germany

INTERNAL REVIEWERS

Name	Organization	E-mail
Klaus Schmid	University of Hildesheim, Germany	schmid@sse.uni-hildesheim.de
Gilles Malfreyt	Thales, France	gilles.malfreyt@thalesgroup.com

EXTERNAL REVIEWERS

Name	Organization	E-mail
Øystein Haugen	Østfold University College, Norway	oystein.haugen@hiof.no
Roberto E. Lopez-Herrejon	Ecole de technologie superieuré, Montreal, Canada	roberto.lopez@etsmtl.ca

CHANGE HISTORY

Version	Date	Reason for change	Sections / Pages Affected
01	30 June 2017	First draft	Initial version
02	21 September 2017	Complete version	Refinement of all sections
03	13 October 2017	Reviewers added comments and editors started to check their suggestions	Refinement of all sections
04	24 November 2017	Editors addressed internal reviewers' comments and fixed layout and referencing issues	Refinement of all sections
05	15 February 2018	Editors addressed external reviewers' comments	Refinement of all sections

Executive summary

This document presents the state-of-the-art related to round-trip engineering in the context of variability management projects. It covers general variability management topics and it elaborates on methodologies for round-trip engineering including product line extraction, product line verification, and product line co-evolution. It also presents several tool integration methods that can help achieving a holistic solution to these interconnected challenges. This public document, courtesy of the REVaMP² consortium, discusses also other projects that have dealt with similar or related challenges.

Table of Contents

Glossary and acronyms

	Definition	Examples
Artefact	Any intermediate ‘work product’ or ‘by-product’ from the software development lifecycle	A requirements document is an artefact. The implementation of a software component is also an artefact.
Annotative approach	A technique to enable the derivation process of variants in software product lines consisting in annotating the implementation parts with features or feature combinations. This way, they will be kept during derivation if the desired feature configuration satisfies the annotation or removed otherwise. See also compositional approach as an alternative.	Pre-processor directives like #ifdef in C or C++ source code.
Asset	Any entity or artefact that has potential or actual value to a person or organization.	Domain features, domain models, domain requirements specification, domain architecture, domain components, domain test cases, domain process description, use cases, logical principles, environmental behavioural data, design, specifications, algorithms, source code, function libraries, documentation, test suites, test reports, manual procedures, usage data, or maintenance data. In general, from a risk analysis perspective, asset is something that a stakeholder wants to protect. An asset is important to that stakeholder for whom the risk analysis is done.
Binding Time	The time at which a variation point is resolved. After this time this variation can no longer be changed.	Examples in SPL Engineering: programming time, pre-processor time, compile time, link time (when the compiler assembles the executable file), load time, execution time.

	Definition	Examples
Compositional approach	A technique to enable the derivation process of variants in software product lines consisting in a core where we can compose/include implementation parts associated to each feature or feature combinations. See also annotative approach as an alternative.	Aspect-oriented programming (e.g., AspectJ [Andrade et al. 2013]) or the Feature-oriented approach (e.g., FeatureHouse approach [Apel et al. 2013]).
(Product) Configuration	The output of a configuration process where the product variability is resolved to satisfy the specific needs of a customer or market.	A selection of features from a feature model.
Legacy Assets	An *asset* which maintenance and further development presents issues.	A software component that has been in a company for a long time and that any of the initial developers are still in the company or it was developed using old techniques. It might still have value but a modernization will be needed to increase its quality or maintainability.
Problem space	The problem space is constituted by the domain analysis and requirements engineering phases in software product line engineering. See also solution space.	The variability model and the product configurations are part of the problem space.
(Software) Product Line (SPL)	A set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. [Northrop et al. 2009]	Examples of economically successful Software Product Lines can be found at http://splc.net/fame.html
Reengineering	Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.	A semi-autonomous aircraft system can be reengineered into a fully-autonomous system. See for example F-4 target drones.
Refactoring	A disciplined technique for restructuring an existing body of software code, altering its internal structure without changing its external behaviour.	Adding, removing, replacing, introducing, hiding, extracting, encapsulating, of methods, classes, variables, parameters, modules. See refactoring.com for more examples.

	Definition	Examples
Reverse Engineering	Reverse Engineering is the process of analysing a system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. The objectives are re-documenting software and identifying potential problems, in preparation for reengineering.	Extraction of UML class diagrams from source code of a undocumented software system.
Round-Trip Engineering (RTE)	A functionality of software development tools that aligns and synchronizes related software artefacts, such as source code, models, configuration files, and even documentation, in a cycle and in different abstraction levels. The need for round-trip engineering arises when the same information is present in multiple artefacts and therefore an inconsistency may occur if not all artefacts are consistently updated to reflect a given change.	The synchronisation and co-evolution between code artefacts and UML models.
Solution space	The solution space in software product line engineering is constituted by the phase for the design and implementation of the architecture that enable the derivation of the family of products from a set of reusable assets. See also problem space.	The implementation of a component associated to a feature and its inclusion in the product derivation system is part of the solution space.
Variability Management	Variability Management is an organizational competence/activity to adapt a product to several different customer requirements. The different variants constitute the product family.	A company that creates smartphones can provide different variants of the same model for different markets due to regulatory or customer requirements.
Variability Modelling	The process of describing more than one variant of a system.	Variability expressed by feature models, variability expressed in relation to a base model, COVAMOF [Sinnema et al. 2004]

	Definition	Examples
(Product) Variant	A product resulting from the realization of a set of variation points derived from a product configuration.	From a portfolio perspective a product variant is a particular form or state of a product that differs in some respect from other versions or from a standard.
Variation Point	A variation point represents a decision leading to different variants. At the system implementation level, variation points are parts of the assets that are made variable, referring to the Variability Model. A variation point can be associated with a specific binding time.	A variation point that adds the functionality of credit card payments in a vending machine. In the vending machine source code, this functionality is included at design time through pre-processor directives.
Version (asset)	A version of an asset is the recorded state of that asset at a given point in time.	Two versions of an asset may represent same or different content of that asset. Thus versions reflect the same asset at different points in time.

1. Introduction

Software-Intensive Systems and Services raise new engineering challenges. They require more agile, round-trip engineering processes to better leverage legacy assets while adopting and applying more systematic variability management.

This public document presents the state-of-the-art on the different parts of this challenging round-trip engineering process that is at the heart of the REVaMP² project and that can drastically improve the competitiveness of software-intensive companies. Thus, this document covers topics ranging from variability management and methodologies for round-trip engineering including product line extraction, product line verification and product line co-evolution. We complement it with the state-of-the-art on tool integration as it is important for proposing a holistic solution to these interconnected challenges. We also present other projects that have dealt with topics related to REVaMP² to acknowledge them as the sources of contributions that improved the state-of-the-art.

This document is written as a guide to and reference material for any person interested in variability management, extraction, verification and/or co-evolution. It will also serve as one of the pillars of the REVaMP² project where the idea is to build on top of the state-of-the-art and on top of a common understanding shared among all the partners (both researchers and practitioners). The tool integration section is also representative of the state-of-the-art regarding tool interoperability.

The structure of the document is as follows: Section 2 introduces relevant concepts and background information about the state-of-the-art on variability management and product line engineering. Section 3 presents and discusses Round-trip engineering as a holistic approach to deal with variability management. To go in-depth in this round-trip engineering process, Section 4, 5, and 6 present the state-of-the-art on product line extraction, verification, and co-evolution respectively. Each of these sections are structured in the form of an introduction, a presentation of the characteristics of the existing approaches, a description of methodologies and tools (with a tool comparison), and a list of open challenges. Section 7 follows the same structure as the previous chapters but focusing on tool integration. Section 8 presents previous projects with a discussion about its relation with round-trip engineering and their contributions to the state-of-the-art. Finally, Section 8 presents the conclusions of this document that the REVaMP² consortium is proud to publicly share.

2. Variability management and product line engineering

Software Product Line Engineering (SPLE) aims at effectively reusing software in a systematic way. Instead of developing individual systems, SPLE proposes to develop a family of many similar systems based on a *variability-enabled platform*. SPLE leverages commonalities and helps to manage and model variability. Software artefacts are the foundation of this approach. These artefacts contain variability information, where each such piece of information represents a variation point that enables a decision about the behaviour of the software part it represents, and thus, about the behaviour of the final product. During a configuration process, these variation points are bound with concrete variants, essentially deriving a concrete product instance.

There are several seminal works about SPLE, first and foremost the book published by the Software Engineering Institute (SEI, CMU Pittsburgh) [Clements et al. 2001]. Subsequent works describe SPLE on the abstract level of a course book [Pohl et al. 2005], or close to industrial practice using case studies [Van Der Linden et al. 2007]. The book on Generative Programming [Czarnecki et al. 2000] is of particular interest since it combines product line concepts with model-based engineering. It also offers a good introduction to variability modelling with feature models. The most recent book is probably the one by Apel et al. [Apel et al. 2016], providing concepts around product-line engineering focused around the notion of features as the prime artefacts, together with various variability mechanisms, such as annotative (e.g., preprocessor-based) variation points or compositional techniques (e.g., artefact fragments related to features).

This chapter will describe the most important concepts of SPLE in Section 2.1.

2.1. SPLE in a nutshell

SPLE is based on three important foundations. *Variability Management* ensures that the manifold variants of a product family are correctly managed and is usually based on a variability model. *Domain Engineering* and *Application Engineering* during the development process ensures that the requirements of the entire product family can be met.

Process. The distinction of domain engineering and application engineering is reflected in a two-lifecycle model as depicted in Figure 2.1. The goal is to clearly separate the development of reusable assets and the development of product-specific ones. During domain engineering activities, common and variable artefacts of the application domain are developed as part of the product line platform. These artefacts are used during application engineering activities to create the required software products. Domain engineering is thus the development phase that prepares for targeted reuse. In contrast, the development steps during application engineering exploit this preparation by reusing the prepared assets.

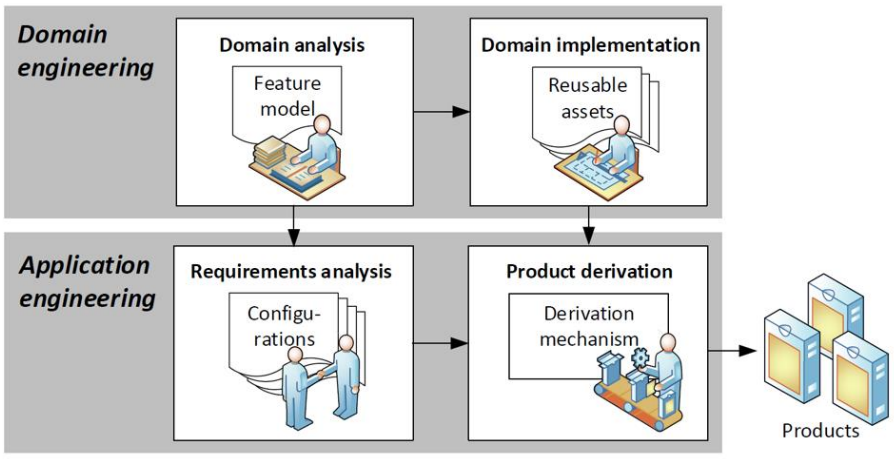

Figure 2.1: Software Product Line Engineering

Architecture. To implement a product line, an architecture that describes the basic organisation and structure of the product line is necessary. It thus constitutes the common framework for all products. The variability mechanisms in the architecture will be used to realise the diversity of variants.

2.2. Practical Approaches to Product Line Engineering

Approaches to realise product lines in practice can be roughly distinguished in three different categories (clone-and-own, configurators, and DSLs). The clone-and-own approach is to develop a single system whose code is copied and adapted for different products. Often, version control systems such as SVN and Git and their branch- and merge-functionality are used for this purpose. A study about code-cloning practices in the industry shows that this approach is still used, but scales poorly [Dubinsky et al. 2013]. The study also shows that clone-and-own can work for up to twenty individual products if it is strictly regulated and controlled.

The approach to realise product lines that is scientifically and practically best understood and supported by languages, notations, and tools is based on the use of *configurators* for the derivation of individual products from a variability-enhanced platform. For this purpose, variability needs to be actively modelled (often by using so-called *features*) and managed. The main benefit is scalability, even for configuration spaces (i.e., the space of possible products that can be derived) that exceed ten thousand individual products. It is worth noting that abstraction and modelling of variability as features comprises a major part of the development process in such cases.

A more flexible, but less standardised and often more involved approach is to develop a domain-specific language (DSL) [Voelter et al. 2013]. A product is represented as one description in the DSL that is realised by transformations and code-generation.

2.3. Variability modelling

Modelling variability is essential for approaches based on configurations. Variability models are the input for configurators and describe variability, meta-information, and dependencies within the product line. From the perspective of variability, it is useful to distinguish the following four classes of artefacts:

- Variability elements such as features, variation points, decisions, and decision rationales
- Configurations describing the relevant feature selections
- Assets such as requirements, source code and documentation
- Products resulting from the assembly/transformation of reusable assets

Several tools have been proposed for variability modelling and management in SPLs. thorough overview of variability modelling approaches was presented by Sinnema et al. [Sinnema et al. 2007]. They classify five academic (CBFM [Czarnecki et al. 2005c], COVAMOF [Sinnema et al. 2004], VSL [Becker et al. 2003], ConIPF [Hotz et al. 2006], Koalish [Asikainen et al. 2003]), and one commercial (pure::variants [PureVariants 2006]) variability modelling language. Each represents a different modelling style: CBFM, ConIPF, and pure::variants model variability in terms of features, COVAMOF and VSL in terms of variation points, and Koalish is embedded in the architecture description language (ADL) Koala [Van Ommering et al. 2000]. The study by Sinnema et al. employs a small exemplary product line to classify each approach based on their modelling facilities and tool support. However, the authors emphasise the lack of defined modelling processes, in particular for the extraction and maintenance of variability information. A recent survey of CASE tools shows the diversity of solutions [Bashroush et al. 2017]. We refer to this paper for a more updated view of the existing approaches.

It is worth noting that out of a total of 91 approaches for variability management that are described in the literature (according to a study by Chen and Babar [Chen et al. 2011]), 33 are based on feature models, thus our focus in the following will be on feature-based approaches.

Feature-based approaches to software product line engineering in general and variability modelling in particular, use features as units of capability. In this context, the concept of features is used in requirement specification, product configuration and configuration management, development and delivery to customers, parameterization for reusable assets, product management for different market segments etc. [Kang et al. 2013]. The dependencies and constraints among features are represented in a tree-like menu of the configurations options or features called a feature model.

Almost all feature models are based on the FODA (Feature-Oriented Domain Analysis) notation introduced by Kang et al. [Kang et al. 1990] An overview of successors of FODA is, e.g., presented by the same authors [Kang et al. 2009] and also provides an outline of feature modelling research between the beginning of the nineties up to 2009. An additional notable systematic review on feature modelling focuses on formal semantics of feature models [Bontemps et al. 2004].

Figure 2.2 presents an example feature model in the FODA notation. The model illustrates core concepts shared by many variability modelling languages that use the concept of features. In this example, the variability of the Journaling Flash File System—one of the numerous files systems supported in open-source operating systems, is modelled.

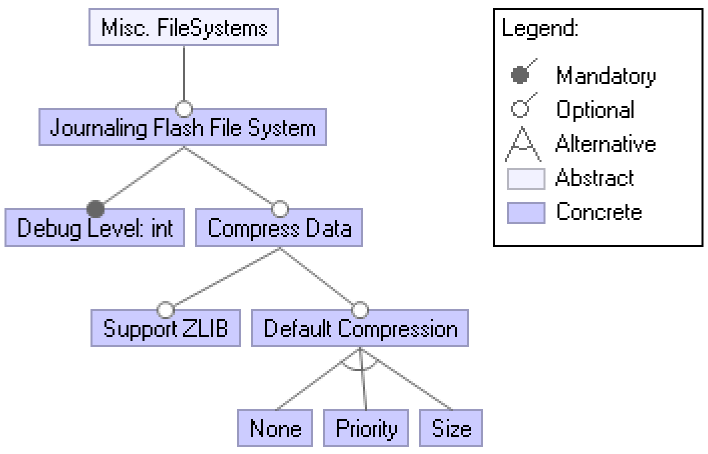

Figure 2.2: Feature model of the JFFS2 file system.

Features in a feature diagram are represented by boxes, and dependencies among them are depicted by the hierarchy; for instance, the *Compress Data* feature allows a further choice of sub features: *Support ZLIB*, *or Default Compression*. Filled dots indicate mandatory features (like *Debug Level*), which must be selected in all cases where the parent is selected and hollow dots represent optional features, which may or may not be included in product configurations where the parent is selected. In addition, features can be related by a group constraint such as XOR (a.k.a., alternative)—exactly one sub-feature is selected, or OR—one or more sub-features can be selected. The XOR-constraint is represented by an arc as shown in Figure 2.2 in the case of sub-features of *Default Compression* whereas the OR-constraint would be represented by a filled arc. Other dependencies among features could be defined such as which features need to be selected together, i.e., one feature *requires* another, and which feature are mutually exclusive, i.e., the selection of one feature *excludes* another. Such constraints are referred to as cross-tree constraints. An example of a *requires*-constraint would be: *Support ZLIB* requires *ZLIB inflate, represented* textually as *Support ZLIB → ZLIB inflate.* More complex constraints can be modelled using constraint modelling languages like the Object-Constraint Language (OCL).

Schobbens et al. [Schobbens et al. 2007] investigate seven feature modelling languages, all of them FODA variations. Based on the argument that most of them are not sufficiently formalised to avoid ambiguities, they develop a common abstract syntax ("Free Feature Diagrams") and define a formal semantics for each language. They use this abstract syntax as the foundation for a new language and semantics ("Varied Feature Diagrams") that is as expressive as the existing ones, but syntactically more concise. Interestingly, the authors make the argument that their work excludes the possibility to argue for extensions of variability modelling languages based on missing expressiveness. Some newer work [Berger et al. 2010b, Berger and She 2010, She et al. 2010, Eichelberger et al. 2013], however, challenges this argument.

Besides graphical ways of representing variability, more recently also textual techniques are increasingly used. A survey of such techniques is given in [Eichelberger et al. 2015b].

Languages and Tools. The following list comprises existing and usable variability modelling approaches and notations. The list is sorted by how often the approaches are used in practice based on a study on variability modelling. Decision models, mentioned below, are an additional well-known modelling technique. However, studies [Czarnecki et al. 2012] show that they are conceptually very similar to feature models.

Language or Tool	Description
Feature models	Formal way of describing variability by defining features and their dependencies
UML-based representations [Clauß et al. 2001]	Use of UML extensions (stereotypes) to define feature diagrams by adding elements describing variability using UML diagrams (e.g., <<optional>>, <<mandatory>> in UML classes with the names of features)
Spreadsheets [Berger et al. 2013]	Some companies that have highly configurable systems (software product lines) may not use feature models but keep an overview of their features or configuration options in a spreadsheet
Key-value pairs (in XML or text-based configuration files) [Berger et al. 2013]	Configuration files used to specify the parameters and initial settings for individual products
Product matrices [Berger et al. 2013]	Non formal representation of variability information by using a matrix of products and their respective features
Domain Specific Languages	Variability is modelled using domain specific language mechanisms
Decision models [Dhungana et al. 2008, Schmid et al. 2004]	Focuses on capturing decisions to be made in configuring products
Aspect-oriented languages	Aspect Oriented Software development deals with crosscutting concerns in software systems and aims at introducing ways to modularise these concerns and denote their relationships to other concerns in the software process [Stoiber et al. 2007]. This is similar to software product-line engineering that exploits commonalities and manages variabilities of software products
Configuration mechanisms of component frameworks (Spring, OSGi, EJB, etc.)	This refers to component-oriented software development and assembly (CODA) that is used to reduce the development effort as much as possible by encouraging the reuse of existing components to assemble new applications or products [Manickam et al. 2013]. The runtime program that wires components deployed in a particular application is called a component framework. Spring, OSGi and EJB are examples of component frameworks.

Language or Tool	Description
Architecture description languages [Clements et al. 1996]	Architecture description languages are used to describe software architectures. Part of what can be described by an ADL is the variability of an architecture by representing the variations in the products that can be derived from an architecture; for instance structural variability such as adding or deleting components [Kogut et al. 1994]
Goal models	A goal model is a hierarchy of system goals that relates high-level goals to low-level system requirements. Modelling variability using goal models aims at addressing variability in requirements [Lapouchnian et al. 2009]

Variability-modeling languages with existing open-source tooling:

Language or Tool	Description
Kconfig (xconfig), CDL (configtool) [Berger et al. 2010c]	Kconfig is a variability modelling language for the Linux kernel that allows Developers to easily select the high level features of the project to be enabled or disabled at build time. CDL (Component Description Language) is part of eCos, a real-time operating system for embedded devices and is used for describing the interface and behaviour of software components.
EASy-Producer [EASy-Producer 2016]	EASy-Producer is an open-source tool that supports the lightweight engineering of SPLs and variability-rich software ecosystems [Schmid et al. 2015]. It integrates both, interactive configuration capabilities and a DSL-based approach to variability modelling, configuration definition, and product derivation. The tool-set includes a dedicated variability modelling language (IVML), which supports beyond variability modelling also meta-modelling (for model-driven engineering) and can support ecosystems. Ecosystems go beyond traditional SPLs as they might extend across many stakeholder organisations.
Dopler Tool Suite (based on decision models) [Dhungana et al. 2010, Dhungana et al. 2007]	The DOPLER (Decision-Oriented Product Line Engineering for effective Reuse) tool suite provides integrated support for diverse stakeholders involved in product derivation and also includes capabilities to manage application-specific requirements during the SPLE life cycle.
TVL [Classen et al. 2011, Hubaux et al. 2010]	The textual variability language (TVL) is a Feature modelling dialect meant for software architects and engineers to complement graphical notations during variability modelling.

Language or Tool	Description
Clafer [Bkak et al. 2010]	Clafer is a lightweight language for domain modelling with support for variability modelling that aims at improving understanding of the problem domain in the early stages of software development. As a meta-modelling language, Clafer offers first-class support for feature modelling and provides concise notations for meta-models and feature models.
FeatureIDE [Thüm et al. 2014b]	FeatureIDE is an extensible framework for feature-oriented software development for the development of SPLs. It is an Eclipse-based Integrated Development Environment that supports all phases of development from domain analysis, domain design, implementation to requirements analysis, software generation and quality assurance.
AHEAD Tool Suite [Batory et al. 2004]	The AHEAD tool suite is a set of tools that support Feature Oriented Programming which uses compositional programming. In compositional programming, features are implemented by modularised units that programs within an SPL can share and a specific product is defined by a composition of feature modules.
COVAMOF [Sinnema et al. 2004]	COVAMOF is a variability modelling approach that allows for modelling variability in all abstraction layers of a product line by treating variation points and dependencies as first class citizens and allowing to model relations between simple (one-to-one) and complex (n-to-m) dependencies.
BVR (Base Variability Resolution) [Vasilevskiy et al. 2015]	BVR is a language focused on feature modeling (vspec modeling), configurations (resolution modeling) and specializing of the derivation process (product derivation). BVR shares the principles of the Common Variability Language (CVL) (See Section 7.3).

Commercial languages with existing tooling:

Language or Tool	Description
Gears by BigLever Software, Inc. [Krueger et al. 2007]	Gears allows SPLs to be managed with a single set of requirements and single set of source code as though it were a were a single system. Gears can manage configurable software artefacts such as source code and test cases, product feature profiles and can automatically assemble and configure software assets based on the product feature profiles
pure::variants by pure::systems GmbH [PureVariants 2006]	Pure::variants is a tool set that supports variant and variability management in all phases of the SPLE lifecycle. It integrates into development processes and offers requirement management, design and development support, configuration management and quality management as well as application lifecycle management.

Language or Tool	Description
Product Modeler by Configit, Inc. [Configit 2017]	The Configit product modeller provides modelling and runtime environments for developing configurator applications.
XFeature by P&P Software [XFeature 2017]	XFeature, a tool provided as a plug-in for the eclipse platform, is a feature modelling tool that supports modelling SPLs and products that can be instantiated from them. Uses allowed to create their own feature meta-model.

Table 2.1: Languages and tools for variability modelling

Additional commercial, not necessarily feature-based configurators include, amongst others: Oracle Configurator/Modeler, SAP configurator, Camos Product Configurator, Siebel Configurator (Oracle), CVM, Tecnalia PLUM, Hephaestus. Studies with industrial partners show that many additional tools are in use, often home-grown developments based on Eclipse EMF, xtext, or IBM RSA.

Annotative and compositional approaches for derivation:
Regarding the variability mechanisms to derive variants, we can distinguish between annotative and compositional approaches. Annotative approaches generally make use of pre-processors such as the C/C++ pre-processor. That particular pre-processor evaluates and removes source code that has been annotated with the `#if` or `#ifdef` directives before compilation. These directives contain expressions (so called “presence conditions” [Berger et al. 2010, Czarnecki et al. 2005]) for the evaluation of feature symbols from the variability model. These expressions are evaluated based on a concrete configuration during the build process. Source code sections annotated with expressions that evaluate to false are omitted. This is also the predominant variability mechanism in open source product lines. Compositional approaches can use component frameworks (such as OSGi [Alliance OSGi 2009]) or plugin mechanisms to aggregate products from different components based on a configuration. Such mechanisms are also used in software ecosystems with open platforms [Berger et al. 2014].

2.4. Model-driven Product-Line Engineering

Model-driven product line engineering is often based on the third approach outlined in Section 2.2 in which domain-specific languages are used to model variability, enriched with *domain scoping*, i.e., defining which features a product line should support, a discussion of the concrete realisation of variation points, and how the overhead can be managed during the build process [Schmidt et al. 2005]. This combination is plausible. In particular, the applicability of DSLs for variability modelling has been confirmed by Völter et al. [Voelter et al. 2013]
A generic approach to realise SPLs based on feature models and program generators is presented by Czarnecki and Eisenecker in their book Generative Programming [Czarnecki et al. 2000]. In particular, the concrete application of the approach using C++ template meta-programming is described.

Muthig et al. [Muthig et al. 2000] discuss the applicability of the OMG MDA standard for model-based product lines. They argue that, while many extensions of UML to address variability have been introduced via the manipulation of syntactical UML elements, no concise semantical integration of product line concepts into the UML meta-model is possible. Nevertheless, they conclude that MDA is applicable to the realisation of product lines. The essence of MDA is that there is a division between PIM (Platform Independent Model) and PSM (Platform Specific Model). This distinction is not the same as that of PL between Domain and Application, but it is conceptually not very different. One of the main advantages is the integration of associated component technologies such as CORBA, EJB, XML, SOAP, and .Net.

A similar argument is brought forward by Kim et al. [Kim et al. 2005] who emphasise the platform-independent nature of MDA. Concretely, the authors propose a nine-step process (DREAM) who takes both paradigms into account. An evaluation of the approaches using industrial-size projects is unfortunately missing in both publications.

An approach for model-based product line development in the domain of mobile applications is presented by White et al. [White et al. 2008] The approach makes use of a configurator that is independent of specific infrastructure, a mapping between the features of a feature model and the capabilities of a mobile device, as well as a constraint solver to resolve dependencies to such hardware capabilities. The last aspect is particularly useful to configure products with a large configuration space that target devices with a highly diverse set of capabilities.

Krishna et al. [Krishna et al. 2005] discuss model-based techniques to realise product lines for distributed embedded systems with real-time requirements. Their approach (FOCUS) aims at adapting generic middleware technology to the specific quality requirements of the target systems. The case study providing prototypical evaluation is based on the “Boeing Bold Stroke avionics mission computing PLA” (Product Line Architecture). It is unclear whether this preliminary approach has been pursued further.

Wasowski et al. [Wasowski et al. 2004] present an approach for model-based development of product lines based on step-wise model restrictions. It is applied to state charts, a type of model often used to describe the behaviour of reactive embedded systems. Based on the largest and most complex model describing the (theoretical) product with all possible features (often also called 150% model), concrete products are derived by applying restrictions in a step-by-step process. The approach is based on the practical experience of two industrial partners.

Batory et al. [Batory et al. 2003] introduce the theoretical concepts behind the Ahead Tool Suite (see above), in particular the algebraic structure of nested expressions and the step-wise refinement of products (consisting of Java files and resources) to derive a concrete product from the product line. The authors thus follow an approach that can be construed as the antithesis to what Wasowski suggests. A large military product line is used to evaluate the approach, thus demonstrating the scalability of the approach.

Additional relevant work includes a combination of MDA and SPLE [Gonzalez et al. 2005] and the use of Early Aspects (i.e., aspects within requirements, similar to aspect-oriented programming) to realise product lines [Batista et al. 2008].

Regarding variability in models, i.e., the ability to embed variation points directly in models, has also been investigated in several studies. The highly generic CVL (Common Variability Language) standard proposal [OMG-CVL 2012] is probably the most widely spread. A detailed explanation of CVL is available in Section 7.3.

Czarnecki et al. [Czarnecki et al. 2005, Czarnecki et al. 2005b] present and approach to realise variability in models by using *model templates*. Model elements are extended with presence conditions (logical expressions about features) and a generator removes those parts of the model's abstract syntax tree whose presence conditions are not fulfilled in a specific configuration. Such an approach can cause the derived model to become syntactically or semantically incorrect. For instance, transitions in UML state charts might become disconnected from start or end states. Such errors have to be prevented by introducing additional constraints. Therefore, an approach to verify constraints in models templates using a modified OCL semantics is proposed [Czarnecki et al. 2006].

FeatureMapper [Heidenreich et al. 200] is one approach based on Czarnecki's model templates. The implementation in this case is based on Eclipse EMF and is freely available and used in projects. Developers specify the mapping between features and model elements in a graphical editor. The notation of the target models can be arbitrary, as long as it is based on ECore. Concrete instance models are derived based on a feature configuration.

3. Round-trip variability management

3.1. Introduction

Traditionally, top-down approaches have been proposed to establish a SIS PLE solution. The PLE paradigm separates two processes; domain engineering (where the commonalities are identified and realized as reusable assets) and application engineering (where specific software products are derived by reusing the assets of the SPL) [Clements et al. 2001]. However, following those approaches requires a significant investment before any product is derived from the SPL and is only followed by a minority of the companies. In fact, a recent survey [Berger et al. 2013] reveals that around 50% of the SPLs are built taking into account existing products which are re-engineered into an SPL. The extractive approach to SPLs [Krueger 2001] capitalizes on existing systems to initiate a product line, formalizing variability among a set of similar products into a variability model and extracting the reusable assets directly from the initial set of products. The resulting SPL can generate the products used as input (among others) with the benefit of having the variability among the products formalized, enabling a systematic reuse.

Purely top-down model-driven approaches for integrating modelling technologies have failed to gain widespread adoption in the SIS industry. For more than ten years the Object Management Group (OMG) has been promoting the Model-Driven Architecture (MDA) as a solution to deal with systems portability and evolution issues. In the MDA paradigm, the model is the centre of the development, while code is automatically generated for any target programming language (e.g., Java, C++, C#) or target platform. While these ideas have proliferated and created an ecosystem of state of the art modelling tools, e.g. based on OMG specifications such as UML, SysML, and MARTE, the general acceptance of MDA tools, however, is rather weak. This is due, in part, to lack of support for scalability of tools in engineering large industrial systems.

State of the art modelling tools do no longer focus on top-down generation, but introduce a different approach: round-trip modelling. The idea here is that, even if models are not at the centre of the development process, they can be intimately connected to the code level artefacts, allowing teams to navigate more or less freely between both worlds. Round-trip modelling requires three kinds of technology:

- Reverse engineering (i.e., model generation from code)
- Code generation (i.e., code generation from models)
- Code-model-synchronization (applying modifications on one side incrementally to the other side).

However, there is a lack of open source system modelling tools supporting round-trip model engineering, making such support one of the most important differentiators in the modelling tools' market. The tools supporting round-trip modelling only support a very limited number of programming languages. The most important CASE tools in the market (IBM Rational, Enterprise Architect, Visual Paradigm and MagicDraw) all support round-trip engineering with varying support for object-oriented languages such as Java, C++, C#, and PHP.

3.2. The tasks and challenges of round-trip variability management

The goal to enable round-trip variability management will support more agile and automated SIS PLE with a reduction of costs. Its starting point will be the currently available tools for Manual

Variability Modelling and Product Variant Configuration Automation from such manually created models (described in this document).
Then, existing tools will be complemented by new classes of innovative tools and services for:

- **PL Asset Extraction Automation**: addresses the need to automate the extraction and visualization of product lines from legacy assets. This is needed because the extraction, verification and refactoring tools will not simultaneously reach 100% automation and quality. Human expertise will always be needed to adjust their parameters to trade-off automation for quality, evaluate their results and manually edit them. The realistic goal of REVaMP² is to minimize such manual edition steps, not to entirely eliminate them
- **PL Asset Verification Automation**: addresses the need to automate the verification of the quality of product line variability models and assets. These constraints can be for example, inter-feature consistency constraints, safety and real-time constraints that must hold for the whole configuration space or the existence of a non-empty intersection of this space with some business configuration goal
- **PL Asset Co-Evolution Automation:** addresses the need for PL refactoring automation, needed for the synchronization between the different elements that conforms the PL (e.g., changes performed into the products can be spread to the reusable assets and vice versa; changes performed into the variability model can be spread to the products and vice versa).

To achieve this, there are several challenges that need to be addressed, such as the different types of artefacts that must be considered; the level of granularity at which we consider the variability; to give support for the different approach that exists for specifying the variability; the lack of information when reverse engineering from an artefact from a higher level of abstraction to an artefact from a lower level of abstraction or the integration into a tool-chain where the order of operations is unknown beforehand.

4. Product Line Extraction

4.1. Introduction

More than fifty percent of industrial practitioners formally implement an SPL only after the instantiation of several similar product variants using ad-hoc reuse techniques [Berger et al. 2013]. An example of ad-hoc techniques is to “copy-paste-modify” the complete project of an existing product to adjust it to the needs of a new requested product, or the “clone-and-own” that refers to the practice of creating a new variant by cloning a copy of the complete product implementation into a separate branch and treat the copy as a new product from thereon [Dubinsky et al. 2013]. For the adoption of SPL practices, having legacy product variants can be considered as an enabler for a quick adoption using an extractive approach [Krueger 2001]. However, in practice, extractive SPL adoption is still challenging. Mining the artefact variants to extract the feature model and the reusable assets gives rise to decisions which need to be made by domain experts as well as many technical issues to analyse and reengineer the assets.

It is also a common practice to implement in-house solutions for variability management with great potential for improvement if state-of-the-art SPLE practices were applied. A recurrent scenario is the use of very late binding of the variability (e.g., “ifs” conditionals at execution time) where an earlier one will be desired (e.g., design time decision to include or discard a given component). Another recurrent scenario is a company that is completely focused on the SPL solution space (implementation assets and build process) while the problem space (features and feature constraints) is implicit (i.e., no defined formalism and knowledge scattered among the stakeholders). In these scenarios, reverse engineering and reengineering become primordial for the extraction and the transition to state-of-the-art SPLs with a better management of the variability.

The goal of extraction is to input legacy assets with implicit commonalities and output a variability model and assets factorized into a target SPL representation. In other words, in extractive SPL adoption, an organization capitalizes on existing custom software systems by extracting the common and varying implementation parts into a single SPL [Krueger 2001]. SPL extraction from legacy assets has received a lot of attention in recent years: A literature study on reengineering legacy applications has shown the diversity of proposed approaches to deal with this challenge [Assunção et al. 2017] and more than one hundred case studies are catalogued showing great interest from industry and research [Martinez et al. 2017b].

The extractive SPL adoption approach is compatible with both reactive and proactive approaches [Krueger 2001]. For the reactive approach, the company can use an extractive approach to create an SPL that focuses on exactly the same products that they had in the existing variants, or they can only consider a subset of the features of the variants (e.g., those with higher payoff). For the proactive approach, the features in the existing variants can be leveraged while the envisioned ones will have to be created from scratch. In both cases, reactive or proactive, capitalizing on existing assets is an option that must be considered to reuse previous development efforts [Krüger et al. 2016].

4.2. **Characteristics to comparatively evaluate extraction approaches**

Extraction approaches present several challenges at a technical level. In this document we focus on those technical aspects, but we acknowledge other important factors in SPL adoption such as organizational change [Bosch 2001], economic models [Ali et al. 2009] or advanced scoping [Schmid 2000] that are not considered in this state-of-the-art overview.

Artefact types: The first characteristic to compare extraction approaches is to determine whether the approach is targeting specific types of artefacts or whether there is an intention to provide a generic solution to the challenge. The potential of reuse is not only restricted to source code given that documentation, designs, models or components are examples of software artefacts that are being reused too. Therefore, we consider that the artefact types that are supported as input is a relevant characteristic of extraction approaches. The ESPLA catalogue for Extractive SPL Adoption case studies [Martinez et al. 2017b] showcases the diversity of artefact types found in the case studies in the literatures as we can observe in Figure 4.1.

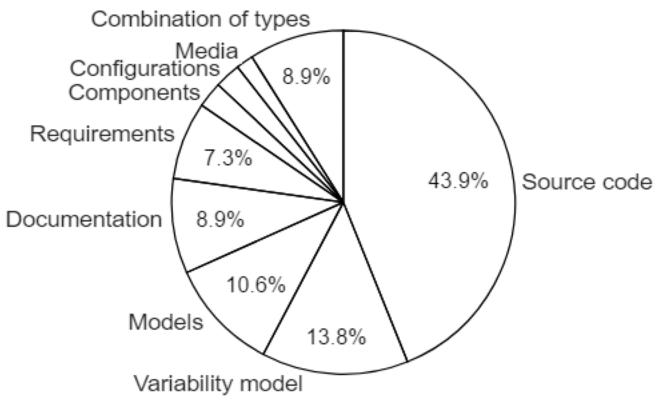

Figure 4.1: Types of artefacts for the extraction of SPL assets

Analysis: The second characteristics to compare are related to the *activities* that it supports. Dealing with legacy artefacts is not straightforward and requires approaches and tools facilitating their analysis. We need to support the task to analyse the variants for knowledge extraction and program comprehension.

Beyond analysis, there are approaches specifically related to SPL extraction. Given the complexity of the products, complete upfront knowledge of the existing features throughout the artefact variants is not always available. Several domains of expertise are often required to build a product and different stakeholders are responsible for different functionalities. In this context, we can assume that domain knowledge about the features of legacy variants is scattered across the organization. In some cases, it may not be properly documented or in a worse scenario, part of the knowledge could even be lost.

The following Figure 4.2 illustrates relevant activities [Martinez et al. 2015] to transition from a set of variants (on the left) to an SPL representation (on the right). We will briefly introduce these activities below.

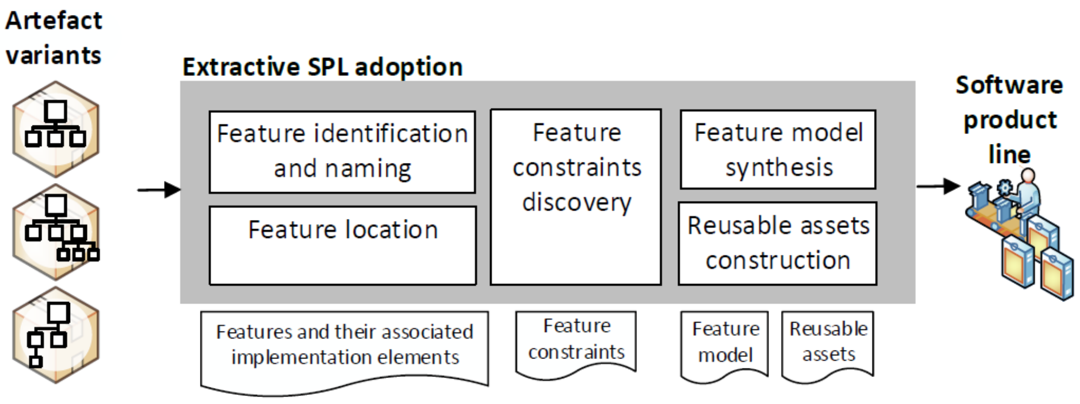

Figure 4.2: Activities of extraction approaches

Feature identification and location: In a scenario of incomplete information, feature identification should be performed. As discussed by Northrop [Northrop 2004], "if an organization will rely heavily on legacy assets, it should begin the inventorying part of the mining existing assets", a practice area which consists in "creating an inventory of candidate legacy components together with a list of the relevant characteristics of those components" [Northrop et al. 2009]. In feature identification, we aim to obtain the list of features within the scope of the input variants as well as the implementation elements related to each feature. During the identification of features there is the feature naming sub-activity to define the names that will be used in the feature model.

In another scenario, if the features are known in advance, feature location can be performed directly to identify the implementation elements associated to each feature.

Feature constraints discovery: The activity feature constraints discovery is important to guarantee the validity of configurations in the envisioned FM. Feature constraints such as the basic "requires" and "excludes" relations between features are identified among more complex relations that can involve more than two features.

Feature model synthesis: FM synthesis consists in defining the structure of the FM to be as comprehensible for the SPL stakeholders as possible. This is because the same configuration space can be expressed in very different ways in a FM (e.g., feature hierarchy).

Reusable assets construction: The implementation elements associated with each feature obtained during feature identification or location, are important for the reusable assets construction which prepares the assets targeting a predefined SPL derivation mechanism.

4.3. Extraction automation approaches

4.3.1. Methodologies

Reverse engineering as defined in the Horseshoe model

The Horseshoe process for Reengineering gives a framework for positioning the various activities when working on legacy software including reverse engineering, reengineering and forward engineering. Figure 4.3 illustrates the Horseshoe model.

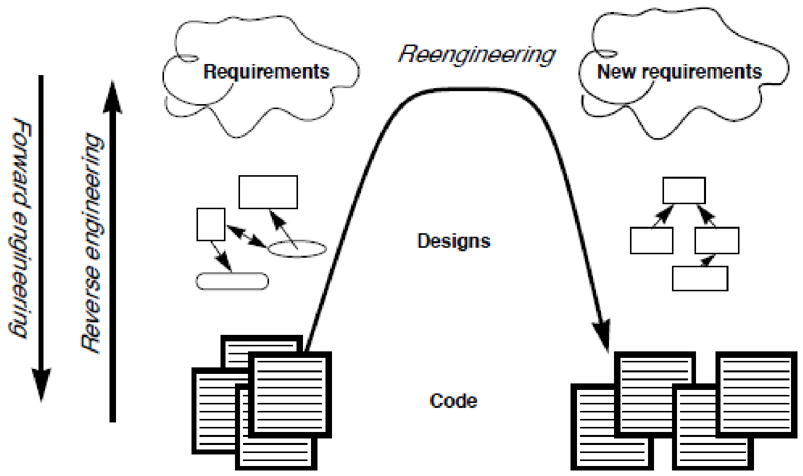

Figure 4.3: Forward, reverse and reengineering

Reverse Engineering is the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. The objectives are re-documenting software and identifying potential problems, in preparation for reengineering. Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system." [Demeyer et al. 2002].

Most tools for Reengineering focus in fact only on reverse engineering, that is, generally, some form(s) of static analysis of the code base, and some form(s) of representation of the results of these analyses. They generally provide no means for re-structuring nor forward engineering, and consider that these actions, when necessary, will be accomplished outside of the tool, using other external tools or manually.

For tools that cover the full horseshoe cycle (the 3 sub-processes), in general reverse engineering produces models that are then processed by re-structuring, and then the transformed models are generally used by Forward Engineering to generate target code. There are a lot of technical choices that may distinguish these tools:

- The level of abstraction:
    - Directly at the code level, with few or no abstraction: in this case, the re-structuring phase may be named more specifically refactoring;
    - Structural/architectural level, representing and transforming the software architecture: in this case, the re-structuring phase may be named more specifically re-architecture;
    - Specification level: in this case, the re-structuring phase may be named more specifically re-specification;
- The nature of technical artefacts on which the tools may apply:
    - Those working at code level may work directly on textual representations of the code, or use Abstract Syntax Trees (AST), or even (meta)models of the code – to mitigate a possible strong dependency of the tool to supported languages, some tools even use a unified (meta)model common to several (generally object-oriented) programming languages.

    - Those working at architectural level may use standardized formal or semiformal architecture representation using concepts such as Classes and Components (UML, ADL, …) or less formalized and non-standardized representations using simpler concepts such as Blocks and Dependencies among blocks.
- The kinds of transformations that can be supported by re-structuring:
    - Those working at code level may, typically: a) translate from a programming language to another (or another version of the same language); b) replace the usage of an obsolete library by the usage of another (more recent) library (typically functionally equivalent but with different interfaces); c) more generally produce mass transformations that are relatively easy to automate but that would be costly and risky to do manually.
    - Those working at an architectural level may, typically: a) replace an obsolete middleware (typically covering both communications and a notion of modules / components) by a more recent one; b) target the usage of a Component Framework (like OMG, CCM, or UCM).
- The level of user implication during transformations of re-structuring:
    - Those working at code level typically support fully (or almost fully) automatic transformations (refactorings are a kind of such transformations);
    - Those working at architectural level typically support less automated transformations, and need explicit user actions to guide the process.
- The completeness of the generation that can be produced by phase forward engineering:
    - Those working at code level typically can fully generate the transformed code;
    - Typically, those working at architectural level cannot directly generate fully correct and directly compilable code, and need user interaction to complete the automated generation done by the tool.

Knowledge extraction and program comprehension

When adopting an extractive product line approach, one must start the extraction process from the two dominant approaches that practitioners use to mimic true product line architectures: 1) clone-and-own; 2) build configurations (or other means to introduce variability in a single system). The first was explained in the introduction and the latter refers to build configuration tools — KBuild is a well-known example for Linux— used to manage the numerous configuration options for generating the appropriate build files [Lofuto et al. 2010]. For clone-and-own, Fischer et al. support the transition from clone-and-own to actual SPLs by offering extraction and composition tools embedded within Eclipse [Fischer et al. 2015]. Other researchers investigated ways to offer support integrated within the version control system [Montalvillo and Díaz 2015, Linsbauer et al. 2016, Schwägerl et al. 2017]. Regarding build Systems, once the feature models scale up it is worthwhile to investigate potential conflicts. Nadi et al. reverse engineered anomalies in the build systems and how the software engineers dealt with them [Nadi et al. 2013]. Abal et al. performed a qualitative analysis of bugs caused by defective product derivations [Abal et al. 2014]. Al-Kofahi et al. proposed a static analysis tool to detect configuration-dependent bugs [Al-Kofahi et al. 2015]. Dintzner et al. reverse engineered changes in the feature models [Dintzner et al. 2014].

4.3.2. **Tools**

We find here a list of tools used during product line extraction. To complement this reading, we recommend the systematic mapping study of Assunção et al. [Assunção et al. 2017] which presents some tools described here and others that exist in the literature.

Pure::variants variability extractor

For pure::variants, prototypical extractors are available, which extract existing variability from legacy source code files. They search for language-specific artefacts, like #ifdef C/C++ preprocessor directives to find the contained variation points. To separate variation points from other switches (e.g., #ifdef _WIN32_ in C/C++) the extractors only take switch identifiers that match a user-specific pattern (e.g., having a prefix VP_). Beside C/C++ #ifdef directives the extractors also support Java conditions and annotations, and assembler directives.
The results of the extraction process are VEL models [Schulze et al. 2015] containing the concrete location of the variation points. The extracted information about the variation points can be used for example to remove variation points in the source code, which are not relevant for a specific variant and which should be hidden from other users [Beuche et al. 2016].

BUT4Reuse: Bottom-Up Technologies for Reuse

BUT4Reuse [Martinez et al. 2017] is a tool-supported framework dedicated to automate relevant tasks for extractive SPL adoption (i.e., feature identification, feature location, mining feature constraints, extraction of reusable assets, feature model synthesis and visualisations). The tool targets two different users:

- SPL Adopters: Companies with existing similar software products that aim to get the benefits claimed by SPLE.
- SPL Integrators: Researchers or practitioners that aim to evaluate concrete techniques for the different steps of extractive SPL adoption. The software engineering research community can integrate innovative techniques for comparison and benchmarking.

BUT4Reuse is designed to be generic and extensible [Martinez et al. 2015].

- Generic by enabling its use in different scenarios with product variants of different software artefact types (e.g., source code in Java, C, MOF-based models, requirements in ReqIf or plugin-based architectures). Currently it supports more than fifteen different artefact types and formats.
- Extensible by allowing to add different concrete techniques or algorithms for the relevant activities of extractive SPL adoption. Several validation studies using BUT4Reuse for different extensions have already been published [Martinez et al. 2014, 2014b, 2015b, 2016, 2016b, 2016c].

BUT4Reuse tool is built on Eclipse and the source code and documentation are publicly available under the EPL license. More information is presented in Section 7.3.3.4 as it can be used as a tool integration framework.
Website: http://but4reuse.github.io

ECCO: Extraction and composition for clone-and-own

ECCO is a tool to enhance clone-and-own that actively supports the development and maintenance of software product variants [Fischer et al. 2015]. A software engineer selects the desired features and ECCO finds the proper software artefacts to reuse and then provides guidance during the manual completion by hinting which software artefacts may need adaptation.
Website: http://jku-isse.github.io/ecco/

FLiMEA: Feature Location in Models through Evolutionary Algorithms

FLiMEA [Font et al. 2016, Font et al. 2017] is a software engineering approach for Feature Location in Models that relies on an Evolutionary Algorithm to locate features in product models and formalize them as model fragments. FLiMEA is specifically designed to target product models created with a DSL conforming to MOF as the artefact for locating the features. To do so, FLiMEA performs a search (guided by an interchangeable fitness function) over alternative model fragment realizations for the feature being located (generated through genetic operations). In addition, FLiMEA can be tailored to work under different domains. Particularly, FLiMEA provides different ways of embedding the domain knowledge from the engineers depending on the nature of the family of models and the type of information available (such as describing the feature to be located using natural language).

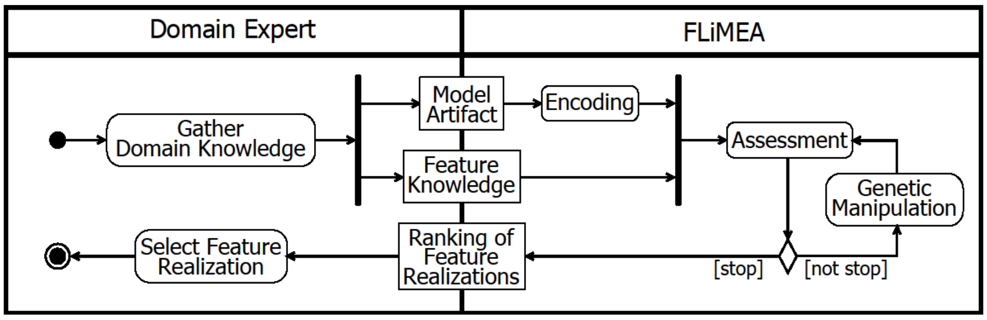

Figure 4.4: Activity diagram for FLiMEA

Figure 4.4 shows an activity diagram for FLiMEA; the left side shows the activities performed by the domain expert while the right side shows the activities automatically performed by the approach. The middle side shows the objects generated in the process. First, the domain expert gathers domain knowledge relevant for the feature that is going to be located. This knowledge is formalized as the model artefact where the feature is going to be located and the feature knowledge (that holds all the knowledge that the domain experts can gather and produce about the feature that is going to be located). Both, the artefact and the knowledge are provided to FLiMEA and this results in a ranking of feature realizations. The ranking will be presented to the domain experts and they will use their domain knowledge again to decide (with the information provided by the approach) which of the realization better fits their needs.
FLiMEA comes in the form of an evolutionary algorithm, a population of individuals (candidate solutions for the problem) is evolved and assessed through several iterations in the search for the best possible individual. When applied to model artefacts, the population of individuals will be in the form of model fragments. These individuals need to be properly encoded (see Encoding in Figure 4.4), enabling the evolutionary algorithm to work efficiently with them. Next, each candidate solution from the population is evaluated using a fitness function (a formalization of the overall quality goal) to determine how well it performs as a solution to the problem (see Assessment). As a result, the population of solutions is ranked depending on their fitness value and, based on the ranked population, some genetic manipulations are performed over the individuals (see Genetic Manipulation). This cycle of genetic manipulations and assessment will be repeated until some stop criteria is met (e.g., fixed number of generations or time).

SciTools Understand
Understand is a static analysis tool focused on source code comprehension, metrics, and standards testing. It is designed to help maintain and understand large amounts of legacy or newly created source code. It provides a cross-platform, multi-language, maintenance-oriented IDE (interactive development environment). The source code may include C, C++, C#, Objective C/Objective C++, Ada, Assembly, Visual Basic, COBOL, Fortran, Java, JOVIAL, Pascal/Delphi, PL/M, Python, VHDL, and Web (PHP, HTML, CSS, JavaScript, and XML). It offers code

navigation using a detailed cross-referencing, a syntax-colorizing “smart” editor, and a variety of graphical reverse engineering views.
Website: https://scitools.com/documents/manuals/pdf/understand.pdf

FETCh: Fact Extraction Tool Chain
FETCh (Fact Extraction Tool Chain) is a tool chain consisting of 100% open-source tools. It bundles and unifies a set of open-source tools supporting key issues in reverse engineering. The target of FETCh is to explore large C/C++/Java software systems for dependency analysis, pattern detection, visualization, metric calculation, and other types of static analysis. It consists of the following tools:

- *SourceNavigator*: SourceNavigator is a multilanguage IDE with parsing support. It parses the source code for structural programming constructs through robust lexical analysis; making it tolerant towards variants of C and C++. The results of parsing are stored in database tables together with location and scoping information. Subsequently, relations between structural constructs such as method invocations and attribute accesses are obtained via a cross-referencing step.
- *pmccabe*: pmccabe calculates McCabe-style cyclomatic complexity of C and C++ source code. It includes a non-commented line counter; a program to only removes comments from source code; a program to calculate the amount of change which has occurred between two source trees or files; and a program to invoke *vi* given a function name rather than a file name. pmccabe attempts to calculate the apparent complexity rather than the complexity after the preprocessing is done. Two types of cyclomatic complexity are generated: first type counts each C switch statement as 1 regardless of the number of cases included, and second type counts each case within the switch. pmccabe also calculates the starting line for each function, the number of lines consumed by the function, and the number of C statements within the function.
- *JavaNCSS*: JavaNCSS is a simple command line utility which measures two standard source code metrics for the Java programming language: Non-Commenting Source Statements (NCSS), and McCabe Cyclomatic Complexity. The metrics are collected globally, for each class, and for each method.
- *IEMetrics:* IEMetrics is a metrics engine for functions/methods.
- *namespaceScript:* namespaceScript is a script to track namespaces and their scopes. It has been extended to also track usage declarations (both for namespaces and classes), as well as the scope of classes.
- *snavtofamix*: snavtofamix mines the output of the above tools, combined with home-made parser extensions, and generates a FAMIX model in Case Data Interchange Format.
- *Crocopat:* Crocopat is a graph query engine which supports the querying of large software models in Rigi Standard Format using Prolog-like Relation Manipulation Language (RML) scripts. In contrast to typical binary relation query languages as Grok, Crocopat’s support for n-ary relations is particularly expressive for software patterns. Moreover, the highly efficient internal data structure enables superior performance to Prolog for calculating closures (e.g., a call-graph) and patterns with multiple roles (e.g., design patterns).
- *Guess*: A graph exploration system which uses as its input graphs that are described in ASCII format (GDF). These input graphs specify various properties of nodes and edges as required for polymetric views. Various layout algorithms are provided.
- *CDIF2RSF:* translation scripts from CDIF to a Rigi Standard Format style fact base.
- *RSF2MSE and RSF2GXL*: a translation script from our RSF format to the MSE exchange format of the Moose environment and another one to the GXL exchange format.

Website: http://lore.ua.ac.be/fetchWiki

HeadWay Structure101

Structure101 is an ‘agile architecture development environment’. This tool builds a structural representation of the architecture of a legacy software, firstly based on its physical organization in folders and files. Structure101 is essentially a static analysis and visualization environment. However, it does not allow the user to directly proceed to refactor the code. Structure101 is to be used by software architects, who will reason on the current architecture, model the future architecture, and give instructions for refactoring to the development team. Once these refactorings are done, Structure101 will reanalyze the source base, compute complexities and dependencies, and a new cycle can start where the software architect reasons on the architecture.

From a static analysis of the dependencies in the code, Structure101 can display, on the working architectural representation, essentially 1) the cyclomatic complexity of various software parts and 2) the dependencies among software parts, at all levels. Both properties give a score to the software on a two-dimensional plan. One can consider that the global goal of an architect working on a legacy software is to improve its score on both dimensions: reducing complexity, and reducing dependencies. More pragmatically, complexity in this tool is given as a hint to locate software parts that may be the most interesting to simplify. And dependencies are used to infer the architecture of the software, in particular Structure101 promotes layered architectures, where layers are deduced from dependencies. For example, cross dependencies are generally considered a bad design, and users of Structure101 are guided and encouraged to reduce their number.

Website: https://structure101.com

Inventive Toolkit

Inventive toolkit is a platform for software analysis providing multiple tools dedicated to optimise operational processes (creation, evolution, transformation and migration of software). Beyond the code, any type of data from various sources and in various formats (logs, bugs history, authors, test coverage, cost analyses) can be imported and stored in a model, correlated in an iterative way in order to produce results usable for additional researches. Figure 4.5 illustrates this process.

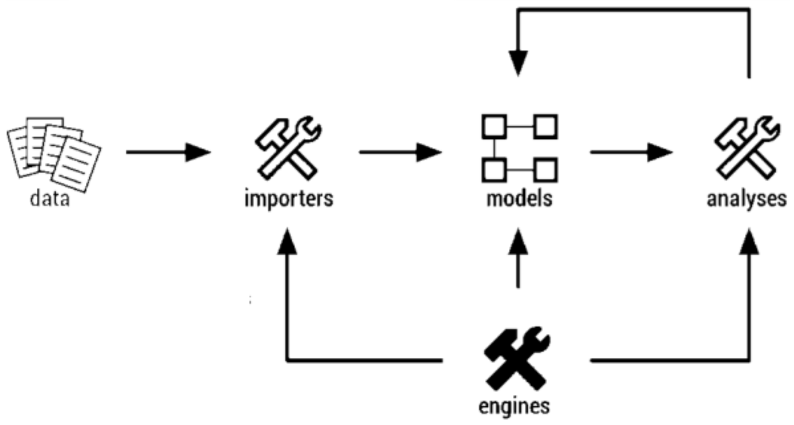

Figure 4.5: Inventive toolkit management of the data

This tool helps software architects to understand and be able to manage old, complex or business-critical software involving multiple programming languages. It aims to recover lost

knowledge or to deliver dedicated tools to address problems like complexity, scalability, modularization or reengineering. It helps simplifying complex interactions among components, automatically identifying patterns through a machine learning process and proposing modifications if needed. It analyses the overall system quality, checks architecture consistency, manages design evolutions and provides a real-time management of the projects allowing the search on any information contained in the model. It computes any pertinent metric and consolidates this information into a global system of high-level indicators.
Website: http://synectique.eu

Mia-Software tool suite

The Mia-Software tool suite provides a set of tools addressing to the whole Horseshoe process.
Source code analysers and Mia-Discovery cover 1) Reverse Engineering, giving a coarse-grain view on the code base, with its general architecture, cross references, and first level impact analysis. Source code analysers can also create conceptual models from the code, and Mia-Studio (containing Mia-Transformation and Mia-Generation) covers 2) Re-Structuring and 3) Forward Engineering, providing an Integrated Development Environment for querying and transforming models created from the analysis of the code base, and automatically generating code from conceptual models.
Using these tools, one can address the goals of: Retro-engineering (abstracting an existing code base by a model), Componentisation (for example identifying and separating the business processes from the human-machine interface within the existing system or systems), and Re-conception (transforming the retro-engineering model into a UML conceptual model). The tool is provided with a methodology including the study of existing code base, pilot migration and industrialized migration.
Website: http://www.mia-software.com

KconfigReader

The KconfigReader tool [KconfigReader 2017] extracts variability model information from the raw data of Kconfig-files [Kconfig 2017] and converts this data into propositional formulas either as files with the model extension or as CNF formula in DIMACS-format [SAT 2015]. Based on this conversion, the variability model can be checked for validity and consistency using a SAT-solver.

TypeChef

TypeChef [TypeChef 2017, Kenner et al. 2010, Kästner et al. 2011] is a variability-aware C-code parser that parses the entire source code, performs macro-expansion, includes all variability information, and produces a variability-augmented Abstract Syntax Tree (AST). In particular, this AST contains detailed information about the conditional (#if*) blocks, like their position in the code and the corresponding logical expression, which has to be satisfied in order to enable the respective block as part of a specific product. Based on this AST, the main goal of TypeChef is to find variability-related bugs but different analyses can be performed, such as variability-aware type checking or data-flow analysis.

Undertaker

The Undertaker tool [Undertaker 2017, Tartler et al. 2011] extracts and analyzes variability information from C source code files and Kconfig-based variability models to detect inconsistencies between the variability definitions in both types of artefacts. In contrast to TypeChef, the variability information extraction for source code files only provides a list of conditional blocks (#if* ... #endif) for each file input to Undertaker, while no analysis of the content of the block happens. In particular, it is not checked whether the blocks actually contain valid C-code. The benefit of TypeChef is that it is much faster. Each Undertaker-block further provides its location in the file (line numbers) and a propositional formula defining the condition under which the respective block will be part of the final product. The extraction of the variability is similar to the one performed by KconfigReader resulting in a propositional formula. The propositional formulas from code blocks and from the variability are than input to a SAT solver to check, e.g., for identifying dead or undead code blocks [Tartler et al. 2011]. Undertaker can extract conditions from C-preprocessor statements (e.g., #if(A && B), but this indicate "local" conditions and not global constraints (if there is always a relationship between both features).

Makex, Golem and KbuildMiner

Makex [Nadi et al. 2011, Nadi et al. 2012], Golem [Dietrich et al. 2012], and KBuildMiner [KBuildMiner 2017, Berger et al. 2010] extract variability information from Kbuild [Kbuild 2017] files, which define the build rules and processes. This variability information consists of presence conditions encoded as propositional formulas, which have to be satisfied to include a specific (set of) source file(s) into a final product's build. The tools realize two different strategies for extracting presence conditions from Kbuild-files. 1) Golem queries the build system to probe which files are controlled by which variables of the variability model and relates these variables to formulas. In contrast, 2) Makex and KBuildMiner use a text-based analysis of all available build files. A comparison of all three implementations show, that Golem and KBuildMiner have the highest accuracy in extracting presence conditions from build files, while Makex is the fastest of these tools [Dietrich et al. 2012].

KernelHaven

The tools introduced above (KconfigReader, Undertaker, TypeChef, and KbuildMiner) are designed for specific purposes, like specific types of analyses. In order to reuse their extraction capabilities in a generalized way, KernelHaven [KernelHaven 2017] is an experimentation workbench for SPL analysis. This workbench allows the exchange of such tools wrapped as plug-ins to extract variability information from different asset types and discover variability related constraints. More information is presented in Section 7.3.3.5 as it can be a mean for tool integration.

4.3.3. Tool comparison

Table 4.1 compares the tools based on the criteria presented in Section 4.2.

	Artefact types	Analysis	Feature identification and location	Feature constraints discovery	Feature model synthesis	Reusable assets extraction
Pure::variants var. extractor	Java, C/C++	Yes	Partially	Partially	Partially	Yes
BUT4Reuse	Generic	Yes	Yes	Yes	Yes	Yes
ECCO	Generic	Yes	Yes	No	No	No
FLiMEA	MOF Models	No	Yes	Partially	Partially	Yes
SciTools Understand	Generic	Yes	No	No	No	No
FETCh	Java, C/C++	Yes	No	No	No	No
HeadWay Structure101	Generic	Yes	No	No	No	No
Inventive Toolkit	Generic	Yes	No	No	No	No
Mia-Software tool suite	Generic, Java, Cobol	Yes	No	No	No	No
KconfigReader	KConfig files	Yes	No	No	No	No
TypeChef	C	Yes	Partially	No	No	No
Undertaker	C and KConfig	Yes	Yes	No	No	No
Makex, Golem and KbuildMiner	KBuild files	Yes	Yes	No	No	No
KernelHaven	Generic	Yes	Yes	Yes	No	No

Table 4.1: Tool comparison for extraction

4.4. Open challenges in product line extraction

4.4.1. The diversity of artefact types

PL extraction requires applicability to a broad range of different artefact types to account for the diversity of assets used in development. The use of a general or standard approach based on asset language adapters is an interoperability solution already explored by tools such as the mentioned BUT4Reuse, ECCO or KernelHaven. The idea is to decompose the input assets as sets of atomic elements that will be manipulated by the different algorithms. The definition of this generic approaches is still an open challenge and it will be required to analyse legacy assets not only at an abstract syntactic level, but also at a richer semantic one.
In addition, as we observed in Table 4.1, source code has received most of the attention while other artefact types, such as requirement specifications or models, have been addressed less.

4.4.2. **Identification and assessment of the available assets**

When dealing with legacy artefacts, it is not easy to “harvest” and identify the boundaries of potentially reusable assets. It is also challenging the assessment of the reuse potential of an identified asset, given that bringing it to a reusable state can be expensive. Regarding assessment for reuse, the version control systems can be a source of information that should be mined as it keeps not only the versions of its evolution but also other important meta-data like the involved developers and timestamps.

4.4.3. **Increase the quality of automatic approaches**

Despite the interest of automation of extraction tasks, full automation is not always feasible. In automatic approaches, false positives or negatives represents a lot of manual effort to be fixed, and the results use to be obtained in a single automatic step creating results difficult for human comprehension. There is a need to continue improving the quality of the automatic approaches to be more reliable or to investigate in guided ones.

4.4.4. **Visualisation support for domain experts**

Several works in the context of SPL addresses visualization ranging from product configuration (e.g., [Nestor et al. 2008]) to SPL testing (e.g., Lopez-Herrejon et al. [Lopez-Herrejon et al. 2013] proposed the use of visualization techniques to display features for pairwise testing). Less work has been targeting visualisation techniques for SPL extraction. Martinez et al. [Martinez et al. 2014b] proposed radial ego network visualisations to reason about feature relations and eventually discover feature constraints. In another work [Martinez et al. 2014], they reused visualisation techniques to analyse the identified commonality and variability that were already used in clone detection approaches. The extraction of an SPL requires domain experts to take part in the process and visualisations to support this process is still an important open challenge.

4.4.5. **Extraction from variants that evolved in parallel**

Variants used for extraction may have evolved in parallel with bug fixes, refactoring or other types of changes making it difficult to apply existing extraction techniques. Existing techniques will have to be extended to work with the “noise” created by variants that evolved in parallel.

4.4.6. **Benchmarking support**

Realistic, non-trivial, comparable, and reproducible settings are needed to compare techniques for extractive SPL adoption. Some efforts already exist for feature location using variants of open source systems such as the Linux Kernel [Xing et al. 2013] or Eclipse integrated development environments [Martinez et al. 2016b]. However, more are desired covering all the activities of extractive SPL adoption.

5. Product Line Co-Evolution

5.1. Introduction

We define co-evolution as the process where at least two evolution streams evolve independently, but are connected and can depend on each other. Changes in one evolution stream can also require changes in the other streams. Doing these changes manually is time-consuming and error-prone. So, the needed synchronization of evolution streams should be (semi-)automatized as far as possible.

In product lines there are different co-evolution scenarios. In this document we cover the following scenarios:

1. Co-evolution between problem space and solution space
Changing of variability in problem space (e.g., adding/removing of features in feature models) usually requires also changes in the solution space (e.g., adding/removing of feature specific source code) and vice versa.

2. Co-evolution between product lines and derived variants
In the ideal case, independent evolution will only occur in the product line itself. Variants will be derived from the product line and if the product line is changed, a newly derived variant will replace the old variant. So, there is no co-evolution at all. However, in practice, the variant itself will be also adapted. For example, project-specific changes are done, which should not be part of the product line; or new potential features will be developed first in a variant and introduced into the product line later. The variant itself may also evolve. Thus, for synchronization between product lines and variants, there are two challenges: 1) How to update a changed variant to apply changes done in the product line? 2) How to introduce changes, which can be new features or changes of existing features, done already in a variant into the product line?

3. Co-evolution between hardware capabilities and software
If the hardware capabilities change (e.g., more/less CPU power, memory, I/O ports) software should also change, especially if the hardware resource limits are reached or real time constraints are not met. In product lines, these limits have to be met for all valid variants.

4. Co-evolution between meta-model and models
When meta-models change, usually also the models, which are the instances of the meta-models, have to be adapted. For example, in MDE Domain-Specific Languages (DSLs) are described using a meta-model, and the variant descriptions are formulated using this DSL. Thus, when the DSL itself is changed also all descriptions using this DSL have to be adapted.

5.2. Characteristics to comparatively evaluate evolution approaches

This subsection is kept for maintaining the structural coherence with other sections but in the case of co-evolution, for our study, we will just consider that the approaches vary in the supported scenarios and the types of targeted artefacts.

5.3. Evolution automation approaches

5.3.1. Methodologies

1. Co-evolution between problem space and solution space

Neves et al. [Neves et al. 2015] introduce a safe evolution concept for compositional and annotative approaches for implementing Software Product Lines (SPLs). As part of this concept and based on an analysis of the evolution of two industrial SPLs, the authors identify a set of safe evolution templates (e.g., guidelines to add a new optional feature in a compositional approach). These templates are safe in the sense that the corresponding changes preserve the behaviour of products derived from the SPL before actually applying those changes. Further, the authors claim that the templates also cover the evolution scenario of introducing new products to an existing SPL without affecting already available products in that SPL. Both scenarios are of particular interest, e.g., to support the co-evolution of a SPL and its related products. However, the safe evolution templates only cover changes, which conform to the refinement notion presented in [Borba et al. 2012] in order to be safe. This excludes other types of changes and evolution scenarios often performed in practice.

In order to overcome the limitations of the safe evolution templates in [Neves et al., 2015], Sampaio et al. [Sampaio et al. 2016] formalize partially safe evolution templates. These templates cover additional types of changes to SPLs and derived products, which are considered to be partially safe as they do not refine all products of a SPL as defined by the initial SPL refinement notion in [Borba et al. 2012]. However, Sampaio et al. show the applicability of their templates based on an analysis of real-world evolution scenarios from the Linux kernel.

Passos et al. [Passos et al., 2016] provide a set of variability co-evolution patterns by analysing a sample of the Linux kernel commit history. In particular, the focus of their study is on the addition and removal of features in the variability model and how the mapping (in terms of build files) and the implementation of these features (C-code files) change as a result. In contrast to the templates identified by Neves et al. [Neves et al., 2015] and Sampaio et al. [Sampaio et al. 2016], the authors do not claim their patterns to be safe, but covering more evolution scenarios. Further, the variability co-evolution patterns only apply to the SPL and do not cover scenarios in which a (new) product should be integrated into the SPL. However, these patterns provide a conceptual basis for the development of co-evolution services as they describe SPL co-evolution scenarios, which should be supported by such services.

Seidl et al. [Seidl et al. 2012] provide a classification for model-based SPL evolution capturing the effects of variability model changes on the feature mapping and the solution space artefacts. In particular, the authors introduce a set of feature remapping operators to enable the co-evolution of the variability model and the feature mapping. While the remapping operators cover similar evolution scenarios as identified by the authors above, this approach assumes all artefacts to be models in the sense of EMOF [MOF 2017], e.g., implemented in EMF models [EMF 2017]. This technical restriction complicates a direct application of the operators, e.g., as the use cases may not provide such models or use different technologies for their models. A contribution by Vasilevskiy et al. [Vasilevskiy et al. 2016] consisted in dealing with the problems occurring in the CVL variability model when the base model (unexpectedly) changes.

Svahnberg et al. [Svahnberg et al. 1999] evaluate two industrial case studies and provide a set of evolution scenarios derived from these studies. Based on this evaluation, the authors propose a set of guidelines, which support the evolution of SPLs. While these guidelines do not particularly address co-evolution scenarios, they are of relative importance as they describe general pitfalls in SPL evolution.

2. Co-evolution between product lines and derived variants

As described above, the synchronization of changes between product lines and their variants can be done in two directions:
In [Schulze et al. 2016] the so-called update scenario is described. The task is to get changes done in the product line (and therefore changes contained in a new derived variant) into an older variant, which was meanwhile manually changed by the user. To identify the changes, a three-way comparison is needed based on the originally derived variant (i.e., the variant without user changes), the newly derived variant (containing the product line changes), and the changed originally derived variant (containing the user changes). Using the work's approach, conflict-free changes from the product line can be applied automatically to the variant. For conflicts, i.e., if different changes are done at the same point, precedence rules can be defined to resolve conflicts automatically.

The other direction is the feedback scenario [Hellebrand et al. 2017], where changes done in a variant should be introduced into the product line. The authors describe different use cases, in which ways this introduction can be done. For example, it is not automatically decidable, whether a change in a variant represents a fix for a single variant, a fix for a feature or feature combination, or a completely new feature, which also has to be introduced to the feature model.

3. Co-evolution between hardware capabilities and software

In Product Line Co-Evolution, the concept of timing can be an important aspect and has to be taken into account. In fact, the analysis of the timing behaviour of a system that is subject to real-time constraints is essential for its validation. For systems from safety-critical domains (e.g., avionics, automotive, medical, etc.), functional correctness needs to be assured and the expected timing behaviour needs to be validated. For this reason, a set of timing requirements is usually defined for such systems. and they are often validated by relying on some worst-case execution time (WCET) analysis techniques.

It exists an implicit relation between the software and the hardware that build up a legacy system and this relation has to be preserved while moving legacy code to a new hardware platform. The timing analysis is also important in this case. Modern hardware platforms that are available on the market are more powerful and versatile compared with older ones. The efficiency of the new platforms also affects the timing behaviour of the whole system; in fact, it is expected that more computation power leads to better performance. Unfortunately, the timing behaviour due to the new hardware can invalidate the system consistency and, thus, the system correctness (e.g., if the code that reads some input data from a sensor, with a fixed sample rate, is moved to a new processor that executes the code twice as fast, half of the read values are inconsistent in the new system).

Similarly, a modification of the legacy software can lead to a disruption of the implicit relation (e.g., addition of a line of code that forces the processor to execute a highly time-consuming operation, like a floating-point operation).

It is possible to analyse the execution time behaviour of a system by tracing its execution via complex hardware tracers, like the TRACE32 tool developed by Lauterbach [Lauterbach 2017].

These tracers are highly accurate most of the time, but they are extremely expensive and they require that both the hardware platform and the software application is available. Unfortunately, this is often not the case when an evaluation has to be conducted to check the validity of moving some legacy code to a new platform (the hardware platform is not yet available). Furthermore, it is inefficient to analyse the code with complex traces for every software change and for all the possible different input configurations.

To cope with the tracers' limitations and in particular with their limitation in predicting the timings for all the possible inputs of a fixed program, simulation based approaches are proposed, such as by Ottlik et al. [Ottlik et al. 2017]. The approach guarantees good results in terms of both accuracy and total amount of time necessary to predict the execution time of a program. In a first stage, the proposed technique builds a context-sensitive timing database (based on the basic blocks granularity) without modelling the hardware resources (e.g., caches, pipelines, etc.) directly. In the second stage, the execution of the program is simulated in a source machine that is run on a fast host machine. During the simulation, the simulator associates for every executed basic block an execution time value that is chosen from the timing database depending on the execution context of the basic block. Despite the accuracy provided by the analysis, this approach requires to create a new timing database for every single change in the executable (source code modification, different compiler optimizations, different compiler, etc.) and for every different hardware platform.

Given that execution time is of great importance in software-hardware co-evolution, we discuss some state-of-the-art regarding its measurement. The group of Prof. Andreas Gerstlauer [Xinnian et al. 2017] presents another approach that is used to predict software execution time without benchmarking. The approach is more software independent and is called LACross. The LACross-approach provides a fast and accurate performance prediction via a learning-based cross-platform prediction technique aimed at predicting the time-varying performance of a program on a target platform using hardware counter statistics obtained while running natively on a host platform. This technique relies on the fact that there exists a correlation between the executions of the same program on two different processors. It requires a learning phase, where the information necessary to teach a machine learning approach is extracted from both a host machine and the target. Then, the program is executed only on the host machine (generally faster than the target one) and some specific information is extracted. The new information is then used as input to the machine-learning algorithm that predicts the execution time for the target platform. LACross appears to provide good results and it introduced a revolutionary approach, but it still has some weak points. It is based on a machine learning approach, and as all machine-learning approaches its results are strongly dependent on the training set. An insufficient training set will imply highly inaccurate timing predictions, and no guidelines are provided to create a good training set. Furthermore, it is adequate for predicting the execution time of a program for different inputs, but it is highly dependent on the hardware configurations (both host and target platforms) and on the compiler optimizations (both host and target platforms). Any modification to the hardware configurations or to the compiler optimizations invalidates the results.

The existing approaches enable the timing prediction of different software variants running on the same platform. Considering both the state of the art and the project objectives of REVaMP^2 to evolve legacy software to new platforms, a new approach is necessary that is able to predict the execution time of a system for different hardware configurations. The new approach has to be able to quickly and accurately predict the execution time of a system even in case of small modifications of the source code (e.g., adaptation to new hardware platforms).

4. Co-evolution between meta-models and models

The main problem when evolving a meta-model is that the conformance between instance models and the meta-model may be broken (depending on the type of changes performed in the meta-model). Models rely on a meta-model to define the domain, but the domain needs to be evolved over time (for different reasons such as improvements in the domain or bug fixes). Therefore, when evolving the meta-model, we must assure that existing models still conform to the evolved meta-model.

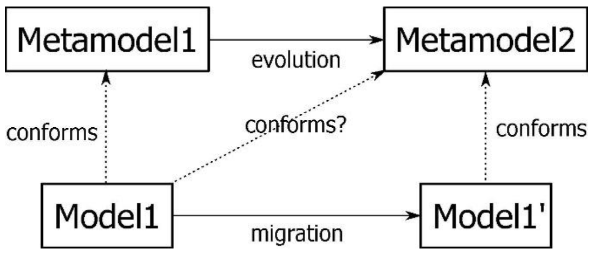

Figure 5.1: Co-evolution of model and meta-model problem

Figure 5.1 shows an example of the co-evolution of models and meta-models problem. The left part shows a meta-model (Metamodel1) and a model (model1) that conforms to that meta-model. That is, the model is expressed following the elements and rules among elements described in the meta-model. Then, the meta-model is evolved into Metamodel2 (usually evolutions are done to address existing issues or to extend the expressiveness of the language). Depending on the changes performed to evolve the meta-model, models that conform to the previous version of the meta-model will not conform to the new version of the meta-model. In those cases, the common practice is to perform a migration of the models, needing to transform them to conform to the new version of the meta-model.

There are several approaches described in the literature to achieve the co-evolution of model and meta-model while maintaining consistency. Those approaches focus on the migration of the models to conform to the evolved meta-model. In particular, [Rose et al. 2009] organize existing approaches in three different categories:

- Manual specification: a transformation is manually encoded. This is the most obvious solution, requiring a lot of work by the developer. There are no specific research efforts towards this topic.
- Operator-based co-evolution: a library of co-evolution operators is defined. These co-operators evolve both, the meta-model and the model (actually, the operator contributes to a M2M transformation that is executed when all the changes over the meta-model are done) [Wachsmuth et al. 2007, Herrmannsdoerfer et al. 2008, Herrmannsdoerfer et al. 2009, Herrmannsdoerfer et al. 2009b].
- Meta-model-matching: a migration strategy is inferred by analyzing the evolved meta-model and the meta-model history. It can be applied in two different ways:
    - Differencing approach: both the original and the evolved meta-model are compared (with a diff tool) to generate a difference model that holds the changes between the original and the evolved meta-model, then it is used to create a migration strategy [Gruschko et al. 2007, Cicchetti et al. 2008, Falleri et al. 2008, Sprinkle 2003, Sprinkle 2004].
    - Change recording approach changes performed to the meta-model are monitored and "recorded" to be used later to generate a migration strategy [Eysholdt et al. 2009].

Garces et al. [Garces et al. 2009] propose a set of heuristics to automatically compute equivalences and differences between two meta-model versions in order to adapt models to evolving meta-models. The computed equivalences and differences act as input for a higher-order transformation, producing an executable adaptation transformation.

[Narayanan et al 2009] present a co-evolution approach for models using the Model Change Language (MCL). MCL is a declarative and graphical language supporting a set of co-evolution idioms. The evolver defines relationships among elements of the meta-model versions. There are different kinds of relationships starting from simple one-to-one mappings between classes to more complex mappings for creating new subclasses or changing the containment hierarchy. More complex co-evolution rules will be defined in the context of the defined mappings as constraints and actions in terms of C++ code. However, the provided idioms only cover syntactical, not semantical co-evolution concerns. The co-evolution is achieved by a model-to-model transformation.

[Herrmannsdoerfer et al. 2009] propose COPE, an integrated approach to specify the coupled evolution of meta-models and models. In their approach, the co-evolution of meta-models and corresponding models is realized by a set of so-called coupled transactions, composing a whole co-evolution problem of modular transformations. Coupled transactions are further divided into custom coupled transactions and reusable coupled transactions, where the latter are predefined and thus help to reduce the migration effort. As custom transactions can be expressed using a Turing-complete language the necessary expressiveness is guaranteed.

[Wimmer et al. 2010] tackle the co-evolution problem from a different viewpoint. Instead of describing it as a transformation between two meta-models, they create an integrated meta-model by automatically merging the initial and the revised model. Then, they use in place transformations, e.g., using ATL for supporting the evolution (transformation).

[Rose et al. 2010] describe Epsilon Flock, a model-to-model transformation language tailored for model migration. Epsilon Flock contributes several features including: a hybrid approach to relating source and target model elements, migration between models specified in heterogeneous modelling technologies, and a more concise syntax than existing approaches.

[Dhungana et al. 2010b] present an approach based on model fragments. They provide tool support for the automated detection of changes, facilitating meta-model evolution and change propagation to already existing variability models.

[Creff et al. 2012] propose an incremental evolution by extension of the product line. They aim to benefit from new development in product derivation, and re-invest them into the SPL models. They introduce an assisted feedback algorithm to extend the SPL to emerging product derivation requirements.

[Passos et al. 2013] develop a vision of software evolution based on a feature-oriented perspective. They provided a feature-oriented project management and system development platform supporting traceability and analyses. In this work, the SPL is specified by means of base models, fragment substitution and meta-model expressiveness.

The Variable Meta-model (VMM) [Font et al. 2017b] is an approach for co-evolving the model fragments realizing the features and the language of the models (meta-model), and comes in response to issues raised by traditional model migration. In [Font et al. 2015b] the authors analysed model migration techniques from the literature and identified three issues related to its application to co-evolve product models and meta-models:

- Overhead: There is an increase in the number of elements used to model the same element of the induction hob.
- Automation: Since the migration of the models cannot be performed automatically, an engineer needs to generate the M2M transformation and take decisions when applying it.
- Trust leak: The modification of the model fragments (through the migrations) decreases the trust gained by those models during that generation.

The VMM applies variability modelling ideas to express each evolution of the language (meta-model) in terms of commonalities and variabilities, ensuring the conformance of all model fragments (old fragments and new fragments) with the VMM. In order to eliminate the need for migration, when a new generation (meta-model revision) is created by the engineers, a new meta-model that supports both generations can be automatically built by the VMM. For instance, model fragments that conform to meta-model version 1 and model fragments that conform to meta-model version 2 will also conform to this VMM. A product model that has been built combining model fragments from both generations will conform to the VMM.

The VMM is incrementally built as the meta-model used to specify the language evolves, including the new revisions of the meta-model in terms of commonalities and variabilities. Since the VMM will be enhanced each time a new generation is created, a single VMM that includes all generations of the product line will exist.

For instance, when using the VMM to manage two meta-model revisions (mm1 and mm2), the approach will compare both revisions and extract the common parts into a base model and the variable parts into a library of model fragments. Then, three different models will be generated: the original mm1, the original mm2 and the combination of both (called mm1&2). Product models built using model fragments only from mm1 will conform to it, product models built using model fragments only from mm2 will conform to it, and product models that combine model fragments from mm1 and mm2 will conform to the mm1&2 meta-model.

5.3.2. **Tools**

SPLEMMA [Romero et al. 2013] supports SPL evolution by enabling SPL maintainers to define authorized (allowed) evolution operations on a SPL. The maintainers therefore define these operations by means of an evolution model, which in turn is based on an evolution meta-model provided by SPLEMMA. This meta-model defines abstract elements, like features, mappings, or source artefacts, their relations and abstract changes (addition, removal or change), which can be applied to these elements. In this way, a derived evolution model specifying a specific evolution scenario can be validated against the meta-model for correctness and consistency of the defined changes. SPLEMMA uses this validated evolution model to generate evolution tools, which automate the evolution scenario described by the model. Therefore, these evolution tools use generic validation services as well as SPL-specific services, which execute the actual evolution operations on the SPL. While in REVaMP² we share the same intent to provide individual tools (services) to support co-evolution scenarios, we do not aim at generating these services on demand. Further, the co-evolution services in REVaMP² should be self-contained such that they can be applied to different SPLs, e.g. those provided by the use cases, without the need of having SPL-specific services in place that perform the actual changes. Further, SPLEMMA only supports co-evolution of different types of artefacts of the SPL. In REVaMP², we also consider scenarios, like the co-evolution of SPLs and derived products.

The feature remapping operators introduced in [Seidl et al. 2012] are implemented as part of FeatureMapper [FeatureMapper 2013] along with Refactory [Reimann et al. 2010] for the evolution of models. This toolset enables the execution of so-called remapping plans after the introduction of model changes is detected and processed by Refactory. These remapping plans contain the necessary remapping operation(s) for the applied changes (evolution). Besides the limitations of the overall concepts with respect to REVaMP² described above, the evaluation was only performed on a fictitious SPL with two evolution steps. Hence, the application to real-world SPLs, like those in the REVaMP² case studies, is questionable.

The approach defined in [Schulze et al. 2016] is implemented in the variant management tool pure::variants [PureVariants 2006] for text-based artefacts and for requirements. For text-based artefacts the three-way comparison and merging of Eclipse is used. For requirements a specific three-way-comparison and an automatic merging approach was implemented for one requirement tool. But the support for other artefacts is missing, mainly because of the lack of good three-way-comparison tools for specific artefact types like graph-oriented models.

Lauterbach TRACE32 tracer and debugger [Lauterbach 2017] is a tool for timing analysis that can be useful in software-hardware co-evolution. This approach requires that both a hardware and software variant is available to trace the actual software execution. The tracer provided by Lauterbach is a well-known tool in the embedded and real-time domain because of its high performance and because of its capability. It supports a large range of technologies (e.g. JTAG, ETM, etc.) and it allows both to trace and to debug software and hardware. The Lauterbach tool can be used to analyse a wide range of processor architectures (e.g. ARM, TriCore, PowerPC, etc.) and different operating systems. TRACE32 results to be a valid tool for the analysis because it permits to extract essential information (e.g., performance counters values, hardware information, etc.) and timing for the execution of a software in a hardware architecture and therefore enables the benchmarking of a legacy system instance.

The VMM approach [Font et al. 2017b] mentioned in Section 5.3.1 is also a tool-based strategy for meta-model co-evolution.

5.4. Open challenges in product line evolution

5.4.1. Co-evolution between problem space and solution space

The consistent co-evolution between problem and solution space requires changes applied to the variability model (problem space) to be in line with the variability realization in related artefacts, like build and code artefacts (solution space), and vice-versa. In order to detect emerging inconsistencies due to incomplete or conflicting changes, e.g., in only one of the two spaces, the variability information in both spaces is typically transformed into logical formulas and evaluated by external solvers. Most approaches supporting such co-evolution scenarios rely on pure Boolean variability representations, like in basic feature modelling approaches (cf. Section 2.3). However, in many cases we also need to take non-Boolean variability into account. For example,

instead of defining all variable characteristics as Boolean features, some characteristics may also be defined as a set of numerical values or even strings. Hence, a major challenge in this project is to extend the focus of existing approaches by non-Boolean variability, which also requires the integration of more powerful solvers, like SMT or CSP solvers.
The determination of consistent co-evolution between problem and solution space requires the presence of variability mappings between the two spaces. For example, information about which feature of a feature model affects the presence or absence of certain (parts of the) code artefacts needs to be available. However, an explicit definition of such variability mappings is not always available, which demands for extraction mechanisms. While this is in general a challenge for asset extraction (cf. Section 4), the co-evolution scenarios may influence this extraction, e.g., by triggering an update of the variability mappings due to changes in the problem or solution space. Hence, the challenge in this project is to support, in particular, the variability (mapping) extraction, as a prerequisite for consistent co-evolution support.
Some of the approaches explicitly focus on the consistent co-evolution between problem and solution space in the presence of artefact models, like in Model-Driven Engineering (MDE). In this development situation, the traditional artefacts, like build and code artefacts, will not be available, but will be generated based on variability-aware artefact models. Hence, a major challenge is to provide co-evolution support for both MDE-based and non-MDE-based product lines in a comprehensive manner.
A general challenge concerning the support for co-evolution of different types of artefacts is to enhance existing approaches by incremental information. For example, if a change only applies to a subset of the available artefacts, the analysis for consistency between problem and solution space should only consider this changed subset instead of analysing the entire product line. In particular, incremental information should be used in different co-evolution scenarios including traditional types of artefacts as well as MDE-based product lines.

5.4.2. **Co-evolution between product lines and derived variants**

For the update-scenario the challenge is to support any kind of artefacts. Currently only text-based artefacts, like source code, and structured requirements are supported. Other kinds of artefacts, like graph-based models should be supported as well. Mainly the challenge is to get a suitable three-way comparison algorithm for a certain kind of artefact. Another challenge is to add proper user support to minimize time-consuming user interactions, especially for the resolution of conflicts occurring during the merging of changes.
The challenges in the feedback scenario are similar: Different artefact types have to be supported and the user interaction should be minimized. But in this scenario the comparison is more complex, since two variant versions have to be compared to the product line itself. Especially when using negative variability, i.e., if the product line assets are supersets of variant assets, the comparison results can be ambiguous, which increases the need for user interaction.

6. Product Line Verification

6.1. Introduction

In this section, we briefly overview the current State-of-the-Art (SotA) in SPL verification tools. As shown in Figure 6.1 (SISPL in the figures stands for Software-Intensive SPLs), an SPLE verification tool takes as input a SPL artefact and a formally defined property to verify on this artefact. It relies on an automated reasoning engine to produce as output a verification result. We thus restrict our overview to the area of SPLE formal verification and not on the much broader area of informal SPLE Quality Assurance (QA) or Analysis. We acknowledge the importance of tools based on testing for SPLE but we restrict our scope to reasoning on formal models. We also exclude tools that automatically extract such formal models from source code, since they have been covered in Section 4, as well as tools that detect the impact of changes to one SPLE artefact to another since they have been covered in Section 5.

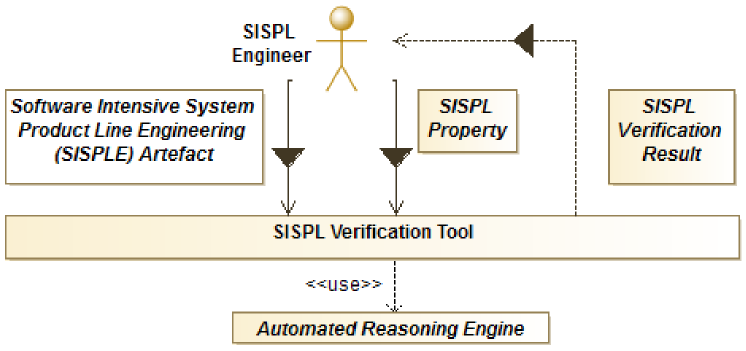
SISPL Engineer
Software Intensive System Product Line Engineering (SISPLE) Artefact
SISPL Property
SISPL Verification Result
SISPL Verification Tool
<<use>>
Automated Reasoning Engine

Figure 6.1: Product line verification tool

6.2. Characteristics to comparatively evaluate verification tools

We provide in this sub-section, what we feel is the minimal necessary background to understand the overview of previous work in SPLE that we survey afterwards. We thus try to provide in what follows a categorization and succinct definition of:

- the SPL properties that have been automatically verified by tools presented and empirically evaluated in previous published research;
- the automated reasoning and knowledge representation concepts underlying the engines that were used to perform such verification tasks;
- the possible results of such verification tasks.

This categorization will form the basis of the list of characteristics that we will use to comparatively evaluate SPL verification tools.

Categorization of SPL properties to verify

Starting with the properties to verify, a high-level categorization of them is given in Figure 6.2 as an UML class diagram. The top-level of this diagram first distinguishes between four top-level classes of properties according to their object: the SPLE variability model, a configuration that can be selected from this model, the component realization artefacts to reuse as basis for product derivation from such configuration or the requirements of such components.

All properties must be expressed in some constraint modelling language. An overview of the sub-categorization is shown in Figure 6.2 based on our initial analysis of the state-of-the-art. The most used constraint modelling languages to express properties to verify on a product line are the OMG-standard OCL. OCL has a programmer-friendly syntax inspired from hybrid strongly-typed object-oriented functional programming languages and a variety of formal logics with their mathematical syntax.

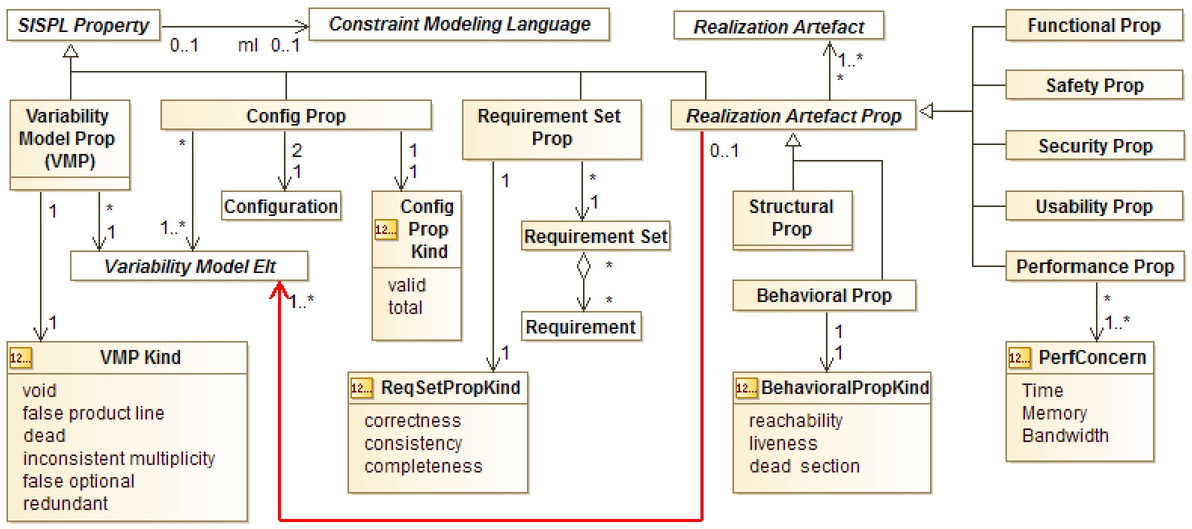

Figure 6.2: Taxonomy of SPL properties to verify

This variety is succinctly modelled at the bottom left of Figure 6.2. The formulas of all logics are made of at least two elements: basic constituents called *atomic formulas*, and *logical connectives* combining formulas into more complex ones. For example, the classical *(a.k.a.* Boolean) propositional logic formula $P \wedge Q \Rightarrow R$ is composed of propositions P, Q and R, combined with connectives $\wedge$ and $\Rightarrow$. Classical logic formulas are formed by connecting atomic formulas with the Boolean connective set: $\neg$ (classical negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence).

Propositional logic is said to be of order zero, because their atomic formulae such as P or Q are both *logically* atomic in the sense of not containing any logical connectives, but also *syntactically* atomic in the sense of properties being defined only by a symbolic name without any internal structure. In contrast, in higher order logics, the atomic formulae are only atomic in the logical sense. In addition, higher order logics also include *quantifiers,* such as $\forall$ and $\exists$ that add set-theoretic semantics to the formulas. So for example, the classical ***High-Order Logic (HOL)*** formula $\forall P,F,X, \exists Y\ (P(F(X),Y) \wedge Q \Rightarrow R(F(Y))$ the atomic formula $P(F(X),Y)$ is a syntactic tree combining four atomic symbols, P, F, X and Y. The order of a logic is the maximum height in the atomic formula syntactic trees where universally quantified symbols can appear, hence in ***First-Order Logic (FOL)*** such symbols can only appear at the leaves of the tree.

Two logics primarily differ from one another in terms of:

- Their vocabulary of connectives and quantifiers;
- The high-level grammar rules on how to combine them with atomic formulae to form non-atomic formulae
- The low-level grammar rules on how to form atomic formulae out of atomic symbols.

Temporal logics extend them with additional quantifiers over an implicit temporal variable to model how properties of a system hold and change over time. They are thus essentially used to verify behavioural properties of SPL components. For example, *Linear Temporal Logic (LTL)* extends classical logic with two temporal quantifiers, *Next* (abbreviated `N`) and *Until* (abbreviated `U`), The formula `NA` where `A` is an LTL formula models that the property represented by A will hold after the next system state. The formula `A U B` models that the property represented by A will hold at least until the property represented by B will. LTL allows modelling state changes only over a *single* system state change *path.* Computational Tree Logics (CTL and CTL*) add other quantifiers over *sets* of alternatives or concurrent system state change paths. The multimodal μ-calculus adds modal quantifiers that express the necessity or possibility of a property to hold following the execution of a given action in a given state. Its expressivity subsumes that of LTL, CTL and CTL*.

Adding new connectives and quantifiers allowing the formation of more complex formulas out of simpler ones is not the only way to extend the expressive power of a logic. Another orthogonal way is to allow the formation of new classes of atomic formulas. This is done by adding new symbols called *constraints* linking variable symbols using predicates and functions from mathematical domains beyond the uninterpreted symbolic syntactic trees of pure logic. A *constraint theory* is a mathematical theory to interpret symbols from such a domain inside these new atomic formulas so as to simplify or solve them. For example, the linear integer arithmetic constraint theory adds equality and inequality (≤) predicates as well as the addition, sign inversion and multiplication functions to form atomic formulas such as $2X - 1 \leq Y$, or $XY = 0$. It also adds the axioms of integer arithmetic to solve systems of such equations and inequations. Other constraint theories model operations on data structures such as bit vectors, arrays, lists and trees. Note that classical first-order logic can be obtained by extending classical propositional logic with the equality theory over uninterpreted symbolic trees.

To go back to the taxonomy of properties to verify in an SPL shown in Figure 6.2, let us define the most common kinds of properties to verify on a product line variability model:

1. Whether the product line is *void, i.e.,* whether all configurations are associated with an inconsistent set of inter-option constraints;
2. Whether it is *false*, which is said of an SPL without any variability, i.e., only one of its option combinations are associated with a consistent set of inter-option constraints;
3. Whether an option of a product line is *dead,* which is said of an option that cannot appear in any valid configuration due to the conjunction of inter-option constraints specified in the variability model;
4. Whether the conjunction of *multiplicity* constraints associated to its options is *inconsistent.* This can be considered as special case of void model.
5. Whether what is explicitly declared as an option in the variability model of an SPL, is in fact implicitly a mandatory element of any working configuration *(false optional).* For example, the option presence is directly required to make a mandatory element properly function or to make a falsely optional element work;
6. Whether an element of the variability model is *redundant* because its subtraction from the model would not change the set of possible products derivable from the product line, and hence it expresses in another way information that is already expressed somewhere else in the model.

Concerning a configuration two main properties can be verified:

1. Whether it is *total, i.e.,* whether one option has been chosen for all alternatives of the variability model; if no choice has been made for at least one alternative, the configuration is then said to be partial;
2. Whether it is valid, *i.e.,* whether the options chosen so far do not violate any constraints of the variability model.

Concerning requirement sets, it can be verified whether they are individually *correct*, globally *consistent* and globally *complete*. Such verification can only be automated if precise, objective criteria for correctness, consistency and completeness have been defined. This can be done for example if the language used to specify the requirements is only semi-natural, following strict syntactic templates and using a controlled vocabulary which terms are defined in an ontology.

As shown in Figure 6.2, properties to verify for realization artefacts, be them models or code, of a SPLE, subcategorize along two orthogonal dimensions. The first distinguishes between, on the one hand, structural properties such as syntactic conformity that can be verified by static analysis and, on the other hand, behavioural properties such as state reachability, concurrent system liveness and dead behavioural code unit or model element, that can only be verified by dynamic analysis. The second dimension distinguishes the properties by the system requirements concern that motivates their verification: functional, safety, security, usability and performance, the latter be it in terms of time, memory or bandwidth resources.

In Figure 6.2, the red and thick association from the `Realization Artefact Prop` class to the `Variability Model Elt` class highlights the fact that in this survey we only summarize previous research concerning verification of realization artefacts *in the context of an explicit variability model*. We thus only cover what [Thüm et al. 2014] calls family-based analysis, excluding what they call product-based analysis and feature-based analysis. This allows us to focus on the core concerns of REVaMP^2, instead of considering the immense but not directly relevant literature on software-intensive system formal verification as a whole.

6.2.1. **Categorization of automated reasoning engines for SPL property verification**

With this well-defined focus, we can categorize the main automated reasoning engines used for SPLE verification in a single class diagram shown in Figure 6.3.

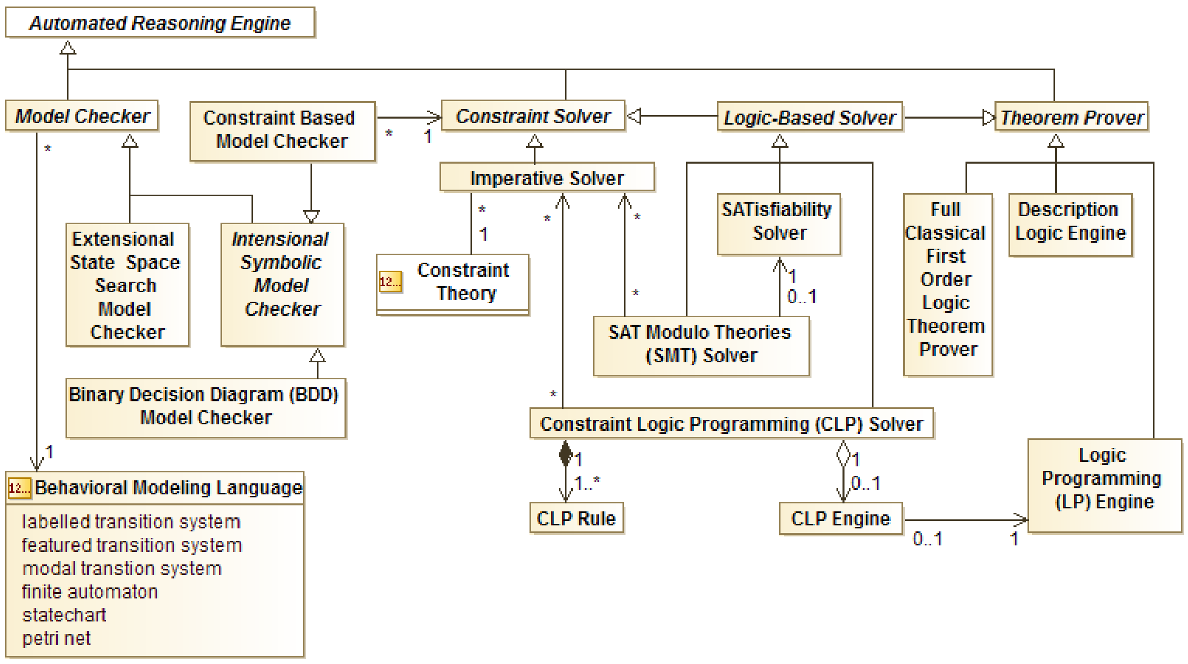
Automated Reasoning Engine
Model Checker
Constraint Based Model Checker
Constraint Solver
Logic-Based Solver
Theorem Prover
Imperative Solver
Extensional State Space Search Model Checker
Intensional Symbolic Model Checker
Constraint Theory
SATisfiability Solver
Full Classical First Order Logic Theorem Prover
Description Logic Engine
SAT Modulo Theories (SMT) Solver
Binary Decision Diagram (BDD) Model Checker
Constraint Logic Programming (CLP) Solver
Behavioral Modeling Language
labelled transition system
featured transition system
modal transtion system
finite automaton
statechart
petri net
CLP Rule
CLP Engine
Logic Programming (LP) Engine

Figure 6.3: A taxonomy of automated reasoning engines used for SPLE verification

The three top-level categories are *model checkers* which originated in formal verification research, *constraint solvers* which originated at the confluence of Artificial Intelligence (AI) and operation research and *theorem provers* which originated at the confluence of AI and computational mathematics.

A model checking problem consists of determining whether the behavioural model `M` of a system verifies a property `F`. `M` is usually expressed in a state change modelling language such as a Labelled Transition System (LTS), a finite automaton, a statechart or a petri net. *Featured Transition Systems (FTS)* extend labelled transition systems with feature-based variability. They can thus concisely capture possible state changes of systems variants. *A Modal Transition Systems (MTS)* extend LTS by qualifying transitions with the modalities necessary, possible and impossible. This allows modelling required, alternative or exclusive behavioural features. `F` is usually expressed as a temporal logic formula.

When `F` *universally* quantifies over the state change paths captured intentionally by `M`, the result of a model checking run can be either:

- `true` when all paths from `S` have been explored and they all satisfy `F`;
- a counterexample path that violates `F`;
- `unknown` if the model checker has exhausted its computational resources before both finding a counterexample path or exploring all of them.

Conversely, when `F` *existentially* quantifies over state change paths, the result can be either:

- one path that verifies `F`,
- `false` when all paths have been explored and none verifies `F`;
- `unknown` if the model checker runs out of computational resources before having explored all the paths and found one verifying `F`.

A *Constraint Satisfaction Problem (CSP)* consists in a logically conjunctive set of mathematical relations called *constraints* between a set of variables taking values in a given domain. A CSP solution is an assignment of values to the variables that satisfies all the constraints. The mathematical relations between variables are composed of functions in the variable domains and Boolean predicates such as equality and inequalities.

A CSP can be *overconstrained* if it has no solution, *exactly constrained* when it has a single solution and *underconstrained*, when it has multiple solutions. For an underconstrained CSP over a finite domain, a constraint solver can return all the possible solutions in extension through an exhaustive search procedure. For an underconstrained CSP over an infinite domain, a constraint solver can only return an intentional representation of the solution space as a simplified CSP with fewer, simpler constraints and some, but not all variables assigned to a definite value. For example, a system of N linear equations relating N+1 real variables is underconstrained, while one with N+1 equations is exactly constrained.

As input, a theorem prover for a logic `L` takes (a) a formula `C` in `L` representing the conjecture to prove and (b) another formula `A` in `L`, representing a conjunction of axioms assumed to be true and from which to derive `C`. A theorem prover implements the set of inference rules of `L`, together with a variety of strategies to search the space of possible application order of these rules to `C` and `A`. It outputs a proof that `A` logically entails `C` as a sequence of inference rule application instances. Most theorem provers output a refutation proof *i.e.,* a sequence that yields `false` from the formula $A \wedge \neg C$. By contraposition, if $A \wedge \neg C \Rightarrow$ `false`, then `true` $\Rightarrow \neg A \vee C$, which is by definition equivalent to $A \Rightarrow C$ and hence that `A` entails `C`.

The first category of model checkers shown on the left of Figure 6.3 first transforms the model checking problem into an automaton emptiness checking problem and then solves the later by a state space search. It builds one automaton representing M, another representing $\neg F$, and a third representing their intersection. It then tests the emptiness of this third automaton by searching for a path leading to a cycle containing an accepting state. The first such path found provides a counter example that `M` verifies `F`. In this approach, each system state is explicitly represented by an automata state.

The second category of model checkers, takes another approach by encoding system state *sets* intentionally as bit vectors with one bit per state and system state transition *sets* as bit vectors with one bit per transition source state and one bit per target state. For example, the transition from state `a` to state `b` is defined by `1001` if the bits follow the order `(current a, current b, next a, next b)`. Using such encoding, the truth value of $\neg F$ can then be computed in a compositional fashion by leveraging at each step compact, non-redundant, specialized data structures to compute the value of Boolean functions called *Binary Decision Diagram (BDD)*.

The bit vector encoding of state sets and transition can also be leveraged to reformulate the model checking problem as a CSP with Boolean variables.

The middle of Figure 6.3 shows that constraint solver can be sub-categorized into two main subclasses:

- *Imperative* solvers implementing various constraint solving algorithms in an imperative programming language such as C or in object-oriented imperative languages such as C++, C#, Java, JavaScript or Python;
- *Logic-based* solvers that reformulate the CSP into a theorem proving problem.

The algorithms implemented by these solvers are all specific to a given constraint theory. Such theory consists of (a) the mathematical domain in which the CSP variables can take their values, (b) the operations inside these domains to combine these variables into constraint expressions and (c) the predicates taking these expressions as arguments.

For example, in the linear equation system CSP: $\begin{cases}3X - 2Y = 1\\ X + 2Y = 2\end{cases}$, $X, Y \in R$, the variable domain is the set of real numbers, the operations are + and – and the predicate is =.

The left of Figure 6.3, sub-categorizes theorem provers into *Full Classical First-Order Logic (FCFOL) Theorem Provers, Description Logic (DL) Engines* and *Logic Programming (LP) Engines or Higher Logic (HL)*. DL and LP differ from the first in that they prove theorem formulated syntactic restriction of FCFOL for which the proving task is decidable and the worst-case complexity is lower, and in some sub-cases tractable. DLs restrict FCFOL to binary and ternary predicates useful to logically formulate taxonomic ontologies. DL are used to formalize semantic web language standards. LP engine prove theorems in Horn logic, (whether zero, first or high-order) which restricts non-atomic formulae to be formed solely of conjunctions of so-called ***Horn clauses***, *i.e.,* implications from a conjunction of atomic formulae to a single atomic formula. Hence, propositional formula

(`P ∧ Q ⇒ R) ∧ (true ⇒ P`) is a Horn formula but (`P ∧ ¬Q ⇒ R ∨ S)` is not.

Defining an LP program consists of declaring:

- One *goal* (a.k.a. *query)* as a Horn clause hypothesis to prove with empty conclusions *(i.e.,* of the form $G_1 \land \ldots \land G_N \Rightarrow$ `true)`
- Known *facts* as Horn clauses are axioms with empty premises (*i.e.,* of the form, true ⇒ `fact`),
- Known *rules* to derive the goal from the facts as Horn clauses with neither empty premises nor empty conclusion.

Horn logic has been chosen as the knowledge representation language of LP for three main reasons. First, Horn logic provides a direct, legible, formal logical semantics to rule bases that modularly decompose knowledge into a set of simple rules of the form: `IF (premise1 holds) AND … AND (premiseN holds) THEN (conclusion holds).`

Second, proving the satisfiability of a Horn propositional logic formula has linear worst-case complexity whereas that of a non-Horn propositional formula is exponential. Third, proving the satisfiability of a Horn first-order logic formula (without tree equality constraints) can be fully automated through the repeated application of a single binary inference rule with a linear search strategy where the output of one inference is always half the input of the next one. This makes their proof very easy to understand. In contrast, efficiently proving non-Horn first-order logic formulae requires using several inference rules in subtle order. Understanding the resulting proofs thus requires some expertise in both logic and search strategies.

Orthogonally to their Horn logic restriction, LP engines differ from FCFOL logic theorem provers and DL engines in another important way: the former make the *Closed World Assumption (CWA),* while the two latter makes the *Open-World-Assumption (OWA).* From the programmer's perspective the CWA means that only known *positive* facts (*i.e.,* Horn clauses of the form true ⇒ `fact)` must be declared in the program as axioms. From the LP reasoning engine perspective, this means the application of an additional implicit inference rule, the CWA, that concludes that any goal that is neither declared as facts nor derivable from facts using declared rules can be inferred as being false.

In contrast, under the OWA the programmer must explicitly declare as axioms not only all known positive facts, and all known rules with positive conclusions, but also all known negative facts and all known rules with negative conclusions of a given domain. As a result, the set of axioms is much larger. This is also counterintuitive to many programmers since database systems and object-oriented programming and modelling languages *(e.g.,* Java and UML) make the CWA. In the absence of sufficient positive facts and rules and negative facts and rules to either prove or disprove the conjecture given as a goal, an OWA theorem prover does not conclude anything concerning the conjecture. However, an OWA theorem prover is guaranteed to make only logically sound derivations, even if the modeller has forgotten some fact or rule, whereas an LP engine may derive unsound conclusions through the CWA if the programmer has forgotten a positive fact or rule. But forgetting negative facts and rules is more frequent than forgetting positive ones.

FCFOL theorem provers can be used for model checking tasks expressed in Linear Temporal Logic (LTL*)* because any LTL formula can be reformulated into a monadic CFOL formula, *i.e.,* a formula which atomic formulas all have arity one. Proving the satisfiability of a propositional logic formula can be reformulated as a CSP over Boolean variables. SAT solvers can also be used for symbolic model checking by using the encoding of system state transitions and temporal formula over them as a conjunction of Boolean variables used by BDD model checkers. Modern SAT solvers in fact scale-up better than BDD solvers for large sub-classes of model checking tasks. They rely on intelligent global backtracking search algorithms such as CDBJ with unit clause learning to find an assignment of truth value to all variables that satisfy all constraints. Very complex SAT problem instances for which global search is not efficient enough may be solved by heuristic local search.

Using SAT solvers to solve CSP in non-Boolean domains requires encoding variables from these domains as bit strings. For example, strings, integer and floats can be encoded by their ASCII binary code. This turns the SAT formula to prove very large. It thus does not scale-up to large non-Boolean CSP. *Satisfiability Modulo Theories (SMT)* solvers address this limitation by integrating SAT solvers with specialized solvers for non-Boolean domains. A SMT solver interleaves calls to the SAT solver and the non-Boolean solvers during its search for a solution. Those specialized solvers can be implemented either imperatively or logic-based. SMT solvers incorporating solvers for constraint theories representing operations over non-primitive data structures such as arrays, lists and trees can be used for model checking of software manipulating such structures, while generally using much less memory than model checkers based on explicit state space search, like BDD or SAT.

Constraint Logic Programming (CLP) solvers are based on the same multi-solver integration design pattern as SMT solvers. The difference is that a CLP engine uses an LP engine as the basic solver to extend, instead of a SAT solver. A CLP solver is programmed declaratively as a set of CLP rules. They extend LP rules by allowing constraints from other theories than first-order atomic formula tree and list unification theories supported by an LP engine. During search, the CLP engine interleaves calls to the LP engine and to solvers for other theories such as finite enumerations, linear integer and real arithmetic and bit vector operations. An SMT solver incorporating a first-order atomic formula tree unification theory and a list unification theory will not be equivalent to an LP engine because the former reasons under OWA while the later reasons under CWA.

6.2.2. **Categorization of SPL verification task results**

Having categorized the properties of a SPL to verify and the automated reasoning engines to use such verification, we conclude our presentation of the characteristics to consider when comparatively evaluating SPL verification tools by categorizing the results that they can provide as output. It is shown in the class diagram of Figure 6.4.

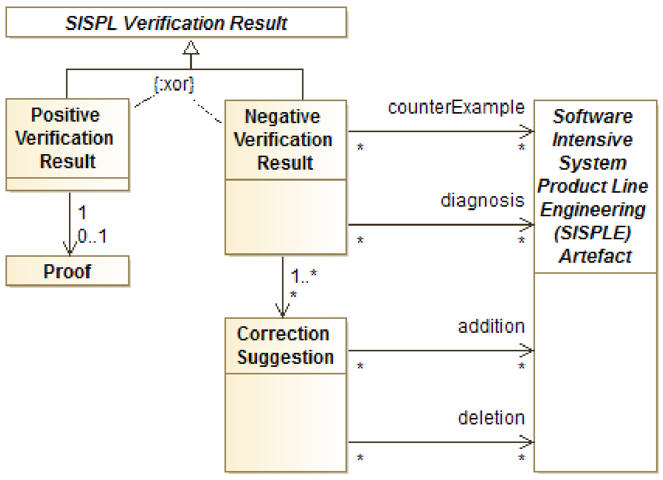
SISPL Verification Result
{xor}
Positive Verification Result
Negative Verification Result
Proof
1
0..1
1..*
*
Correction Suggestion
counterExample
diagnosis
addition
deletion
Software Intensive System Product Line Engineering (SISPLE) Artefact

Figure 6.4: Categorization of SPLE verification results

An SPL verification task result cannot be limited to a simple Boolean variable value. More information must be returned to be helpful to SPL engineers for them to take actions in reaction to the verification result, such as writing qualification documentation or correcting the system design flaw or implementation bug revealed by the verification result.

When the result is positive*, i.e.,* when the reasoning engine has established that the property is verified, the more informative output is a formal proof that it is. For large SPL and complex properties, the automated proofs can be extremely long. In addition, automated reasoners often do not directly reason on the formalism that the engineers used to model the system and specify the property to verify. For example, BDD model checkers and SAT solvers encode states, state transitions, strings and numbers into bit vectors. A raw proof in terms of this binary encoding will hardly be comprehensible by the engineers. Hence, the verification proof presented to the user is rarely a mere log of the inference steps executed by the reasoning engine. In general, it is an interactive data structure with extra information allowing the user to easily browse it, summarize it from various perspectives, zoom in and out of details and so forth. Generating such helpful proofs from the raw inference log of an automated reasoning engine is a human-computer interaction challenge.

When the verification result is negative*, i.e.,* when it was proven false there are three main categories of complementary information that can be returned by a SPL tool, each one providing a richer level of help to the SPL engineers than the previous one. The first level is to provide one or several *counterexample* artefacts violating a universal quantification property over an artefact class. The next level of helpful information is a *diagnosis* that identifies to deep root cause of the property violation. The final level is to provide a *correction suggestion*, i.e., a set of changes the artefacts diagnosed as the root causes of the property violation.

In the area of PL verification, a key problem is that no class of constraint solver scales well on all classes of PL verification problems. A practical industrial strength PL verification service must thus include a variety of solvers allowing its user to choose the best one for any particular class of problem (or even automate the selection). Thus, various solvers will be included, like a CLP solver or an SMT solver and their performance on the PL verification use cases provided by industrial partners will be compared. Further, we will also evaluate soft-constraint optimization techniques such as multi-objective branch and bound and bucket elimination on PL optimization use cases.

It is important to also address key aspects that have so far each achieved only very little interest in the scientific community, but are very important from a practical and industrial perspective. These include: incremental analysis in order to minimize the effort and time required for verification of a software increment and identifying variability smells, i.e., cases which are not errors in a formal sense like inconsistencies, but may point to software correctness issues.

6.3. Verification automation approaches

6.3.1. Verification of *variability model and configuration* properties

Defects in PLE are about undesirable situations, such as anomalies or inconsistencies, both in product line models (PLMs) and in their configuration processes. At the level of PLMs, these defects are usually introduced, involuntarily, when they are being designed due to the fact that it is a non-trivial activity: a single product line can represent billions of products and have thousands of elements that can be specified in the PLM. For instance, adding or removing relations can turn out the whole model inconsistent. That is why quality assurance of PLMs has recently been a prominent topic for researchers and practitioners in the context of product lines. Identification and correction of PLMs defects, is vital for efficient management and exploitation of the product line. Defects that are not identified or not corrected will inevitably spread to the products created from the product line, which can drastically diminish the benefits of the product line approach [Mazo 2011]. Besides, product line modelling is an error-prone activity. Indeed, a product line specification represents not one, but an undefined collection of products that may even fulfil contradictory requirements [Lauenroth et al. 2010]. The aforementioned problems enforce the urgent need of early identification and correction of defects in the context of product lines. This fact implies that identifying the causes of defects and supporting corrections is of paramount importance for assuring quality and getting the promised benefits of PLE. Even if the defects are identified, if they are not corrected, they will inevitably spread to the products created from the product line and negatively affect the evolution of the product line, which can drastically diminish the benefits of the product line.

Proposing repairing alternatives for defects on product line models and their configurations consists in (i) proposing comprehensive explanations of the causes of these defects, and (ii) proposing mechanisms for handling these defects like repair the defect through removal, insertion, and adjustment of constraints, variables or variables' domains or control the defect.

There are in literature several related works that propose automatic identification of defects in product line models [Benavides et al. 2005], [Classen et al. 2010], [Salinesi and Mazo 2012], [Felfernig et al. 2012], [Noorian et al. 2011], [Rincon et al. 2015]. However, just some verification criteria of some modelling languages, and some types of explanations have been studied in a unified manner. For instance, [Benavides et al. 2005] propose a propositional logic-based approach to represent and verify some criteria, like void models and false optional features, on feature models [Classen et al. 2010]. There are some works like the ones of [Felfernig et al. 2012] that focus on identification of minimal correction subsets for constraint programs that cannot be simultaneously satisfied; i.e. for a single type of defect on PLMs known as void models. Other works, such as the one of [Noorian et al. 2011], use constraint satisfaction algorithms to identify some minimal correction subsets for three types of defects: void models (models with no products), dead features (features that can never be selected in a configuration process) and false optional features (features that are modeled as optional ones but are part of all the products of the product line) in PLMs. And there are other works, like the ones of [Rincon et al. 2015], where not only defects on PLMs are identified, but also some explanations about the causes and the possible corrections of these defects are proposed for some defect types (e.g., void feature models).

Regarding the verification of the conformance of product line models relative to their meta-models, [Mazo et al. 2011] propose a conformance checking approach in which they transform PLMs expressed as feature models into constraint logic programs and use a collection of nine rules to check the conformance of feature models with respects to the feature language meta-model. However, even if the approach correctly detects non-conformity defects and detects the cause of these defects, the approach does not consider other languages than feature models, neither the explanation about why these defects occur or how to correct them, nor the defects that can appear in configurations of product line models.

Concerning configuration verification, using the AHEAD Tool Suite, [Batory 2005] relies on propositional logic SAT solving. When a user selects and unselects features, the PLM at hand is automatically evaluated using SAT solvers and logic truth maintenance system algorithms. This evaluation determines whether the selection made by the user causes inconsistencies between the configuration and the PLM. [Sun et al. 2005] use the Alloy Analyzer, a formal method workbench relying on SAT solving, to verify whether a specific configuration is consistent (with relation to the corresponding PLM) or not, to extract the parts of the PLM that cause the inconsistency, and to provide guidance to improve/fix the configuration. [Osman et al. 2009] uses first order logic to represent PLMs and to define a set of rules based on constraint dependencies. When users select features of the PLM, this approach automatically validates the rules using Prolog and it detects the rules that were broken and the characteristics selected that break the rules. [Wang et al. 2007] translate PLMs and their constraints into the ontology language OWL DL. Using FaCT++ as reasoning engine, they automatically determine whether a given configuration, represented in OWL DL is consistent with the corresponding PLM. [White et al. 2010] present an approach that receives as input an inconsistent configuration and a set of constraints of a PLM, and they transform this input into a constraint satisfaction problem (CSP). Then, they use a CSP solver to obtain a minimal set of features to change, in order to make the inconsistent configuration consistent. These works focus on identifying inconsistencies in configurations created from PLMs, but without giving any explanation about the causes and possible corrections of these defects to users.

6.3.2. Verification of SPL *requirements* properties

For most systems engineering companies a major challenge is to manage and guarantee the quality of the system requirement specifications to deliver the final products with the desired quality.
The Knowledge-Based Requirements Engineering approach [Góngora et al. 2017] has gained ground to address this challenge. This is achieved through the use of a domain ontology to provide sources of knowledge for the different stages needed to check and enhance the quality of a requirements specification. An ontology for Requirements Engineering purposes is formed by the following layers, each layer feeds from the inner ones as shown in Figure 6.5:

- **Vocabulary Layer:** Valid terms, forbidden terms, other NL terms, Syntactic clustering types, everything as concepts.

- **Conceptual model Layer:** Relationships among concepts (hierarchies, associations, synonyms…) as well as semantic clusters and relationship types
- **Patterns Layer:** Grammars or sequential restrictions to identify different requirements typologies. Aimed to producing formal representation out of requirements statements.
- **Patterns Formalization Layer:** Definition of the formal representation to formalize requirements statements by pattern matching.
- **Inference Rules Layer:** For decision making (e.g. consistency, completeness)

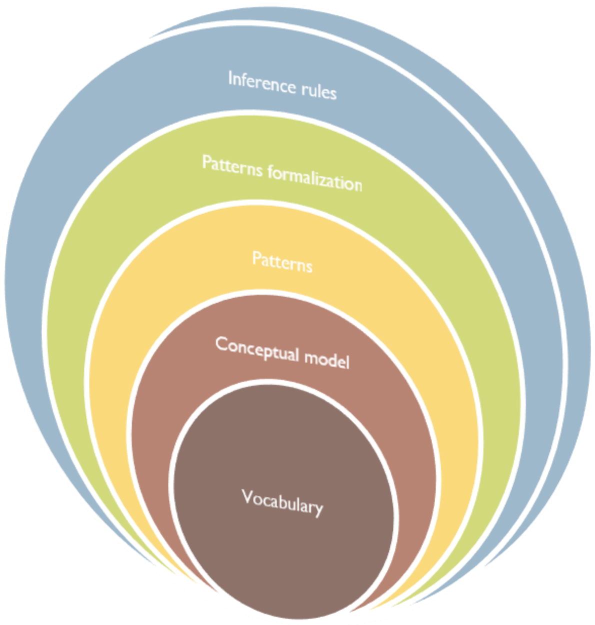

Figure 6.5: Ontology structure the SKB - System Knowledge Base

With this structure, there are several sources of knowledge that an ontology is capable to provide for quality checking and improvement, for example:

- **Lexical analysis**: related to the content of the vocabulary layer of the ontology in terms of valid vs invalid words, domain concepts…
- **Syntactic rules**: to identify the proper tense or voice in the requirements.
- **Boilerplates**: to identify the agreed upon structure for every different type of requirement.
- **Patterns**: to provide the proper formalization for a requirement once it has been matched with a boilerplate.

These sources of knowledge are used to assess the quality of the requirement specification based on the Correctness-Consistency-Completeness (CCC) approach. This approach is based on the following sub-items as illustrated in Figure 6.6:

- **Correctness:** mainly from an individual point of view, requirement by requirement as isolated chunks of information. Thus, each requirement is analysed for non-ambiguity, uniqueness, verifiability, multiple needs in the same requirement…
- **Consistency:** looking for non-consistent information among a set of requirements
- **Completeness:** trying to answer the question: are all the needs properly addressed in the requirements specification?

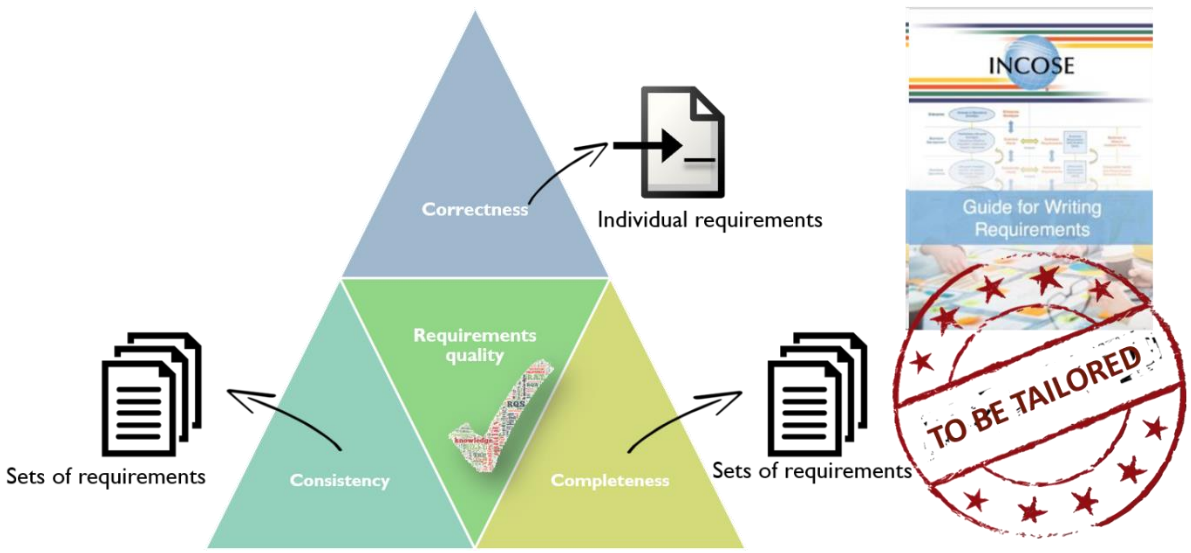

Figure 6.6: CCC appliance (e.g., INCOSE Guide for Writing Requirements)

This approach can be applied to check that the analysed requirements specification quality is guaranteed regarding to any guideline or standard chosen (e.g., tailoring for the INCOSE Guide for Writing Requirements).

The application of the Knowledge-Based Requirements Engineering to the SPL Verification arises using the SPL technology at the Systems Engineering scope. This responds to the need of checking and enhancing the quality of the system requirement specification. Also, keeping in mind the GIGO concept (Garbage In, Garbage Out), the Knowledge-Based Requirement Engineering has an application to the SPL extraction in order to guarantee that the quality of the requirement specification samples, that shall be used to extract the commonality and variability, is the proper one.

6.3.3. **Verification of SPL *functional* properties**

This section describes state-of-the-art tools for verifying SPL functional properties, i.e. properties describing *what* a system does, abstracting from how it does it. For example, that a server always eventually sends a response to every request would be classified as a functional property. In contrast, a non-functional property could specify how fast the response will be served (within what time bounds), or how much memory or energy this is allowed to consume. The rest of this section will focus on actual tools designed to be used for verifying SPL properties and not on approaches that describe smaller extensions of already established verification tools not explicitly designed for verifying SPLs
The work in [Thüm et al. 2014] describes an approach for variability-aware verification of Java programs along with a compatible tool called FeatureIDE. This tool is an extension of another tool called FeatureHouse [Apel et al. 2013] that supports the composition of Java and C source code unit assets realizing features of a variability model. FeatureIDE extends FeatureHouse with the ability to verify properties with respect to variability using theorem proving with KeY [Beckert et al. 2007] and model checking with Java PathFinder (JPF) [Havelund and Pressburger 2000]. KeY considers dynamic logic, an extension of classical propositional and predicate logic with e.g. operators expressing modality, and supports both interactive (i.e. by hand) and fully automated

correctness proofs. JPF is an extensional state space search model checker. A tool that further builds on FeatureIDE is SPLverifier, which is described in [Apel et al. 2011b].

ProVeLines, described in [Cordy et al. 2013], is another variability-aware verification tool. The verification approach is based on *Feature Transition Systems (FTS)* [Classen et al. 2010] a variability-aware extension of transition systems that represents the behaviour of all the products in a single compact model. The tool allows verification of such models against properties expressed in LTL. In addition to LTL, ProVeLines also allows to represent and check feature formulae expressed as: binary decision diagrams (BDDs), boolean formula in conjunctive normal form (to be solved by a SAT solver), and abstract syntax tree. Notably, in contrast to [Thüm et al. 2014c] and [Apel et al 2011], this tool verifies properties of an abstract model, rather than software code. This means that the verification approach and tool are not limited to a certain programming language, and to verifying software since a considered FTS can represent HW and also an entire system.

Another approach that uses FTS to represent the behaviour of a family of products is presented in [Beek et al. 2017]. Unlike in [Cordy et al. 2013], the properties to be verified over the FTS are specified in feature μ-calculus [Beek et al. 2016], which embeds feature expressions into the logic thus allowing the specification of properties that should hold for specific products or a subset of all products. The approach is supported by the general-purpose toolset mCRL2.

Work in [Beek et al. 2012] introduces the VMC tool for specification and verification of families of products represented as *Modal Transition Systems (MTS).* MTS allow expressing possible and required transitions while additional transition constraints, e.g. alternative or excludes, are done using an action-based branching-time temporal logic MHML [Asirelli et al. 2010]. The MHML can also be used to specify the properties to be verified over the MTS which removes the need to use a separate logic for property specification.

The work in [Schaefer et al. 2011] addresses the problem of model checking all products of a software product line (SPL) against temporal properties specified in a fragment of LTL. Scalability of verification is achieved by exploiting hierarchical variability models (HVM) of the SPL to decompose the temporal property of the SPL into properties of the variation points of the HVM. This allows compositional verification to be applied. The work extends and adapts ProMoVer, a previously developed compositional model checker for program models extracted from Java bytecode.

6.3.4. Verification of SPL *safety* properties

In accordance with safety standards such as ISO 26262 and IEC 61508, a safety property of a system is a property that mitigates or prevents hazards, i.e. top-level system failures that constitute potential sources of physical injury or damage to the health of persons/environment. Such standards advocate that the requirements of a system, i.e. intended system properties, are structured in a hierarchical manner, where top-level requirements (goals) are directly linked to relevant hazards and lower-level requirements are linked between hierarchical levels. The intended relation between requirement levels is that of refinement, i.e. that the fulfilment of higher-level requirements can be inferred from fulfilment of lower-level requirements. A top-level requirement is a safety requirement if it is linked to a hazard. A lower-level requirement is a safety requirement if it is indirectly or directly linked to a top-level safety requirement. Thus, whether a property is safety-related or not cannot be distinguished by its formal representation, but rather how it traces to other properties. This also means that whether a property is safety-related or not has no direct influence on how such a property is formally verified.

6.3.5. Verification of SPL *performance* properties

In the area of system design, system simulation has been established as a key method for the validation and verification of functional and non-functional requirements. System simulation is utilized to economize difficult, time consuming, or impossible analyses based on physical prototypes. Additionally, it tackles the system verification problem from another perspective using an instance-oriented verification approach to verify a single product variant of the SPL. In contrast to formal verification, system simulation is not inherently complete but always depends on simulation models.

Simulations models (also known as *virtual prototypes (VPs))* are the basis for simulation-based system verification. In contrast to real physical prototypes that can be seen as early product variants of the SPL, VPs provide better insights into the system under test and enable a better controllability during verification and validation tasks. The fields of applications of VPs cover the well-established register transfer level (RTL) and gate level simulations used for functional verification of integrated circuits (ICs). Important representatives are VHDL [IEEE1364] and Verilog [IEEE1076]. Another approach that gains more and more importance over the last decades is electronic system level (ESL) simulation. This simulation model often contains both the digital as well as the analog system parts. The focus is on higher abstraction levels and a model of the entire system, rather than isolated ICs. Instead of pure hardware description languages, high-level languages such as C, C++ or MATLAB are used. Additionally, there are languages that originate from hardware specification that evolved into ESL languages. A well know example is the language SystemC that provides semantic similarities to VHDL but evolved to an ESL language by introducing concepts of transaction level modelling (TLM). High-level modelling languages are often used for software development. It is possible to use actual software prototypes for the creation of the VP. This reduces the effort to create the VP, while also increasing the accuracy of the system simulation. Such VPs can be used to verify functional, timing or power requirements as presented in [Zimmermann et al. 2012].

The simulation is composed of the Design Under Test (DUT) and the related test bench. A virtual prototype based verification run covers mainly a single system instance (product variant) and a dedicated stimulus. Therefore, only a finite subset of all possible inputs, system states and system configuration can be tested. The proof of the system correctness is therefore incomplete and only proof of the presence of errors is possible but not its absence. Repeated simulation runs with multiple stimuli and system configurations or random deviations are executed to reach a general assessment of the system. Simulation models often provide a modular structure to foster the reuse of different simulation parts. Another aspect is the use of constraint-based random stimulus generation to test the DUT with different inputs. Often, coverage analysis is used to get a quantitative measure about the completeness of the executed analyses. One metric for example, measures the amount of executed functionality to determine the quality of the input stimuli and therefore the test bench.

The event-driven simulation language SystemC enables the modelling of complex, electronic systems at different levels of abstraction. SystemC is a class library for C++, which enables the modelling of concurrent hardware and software. SystemC extends C++ with mechanisms to model synchronization, concurrency and inter-process communication. A core concept is modelling of closed simulation entities (sc_module) and a hierarchical structure of these. The TLM extension fosters the modelling of communication without the specific protocol or hardware implementation. By providing a standardized interface, it increases the interoperability of simulation models. SystemC offers the possibility to integrate realization artefacts such as software prototypes that are based on C/C++ or compatible programming languages. Another benefit is the object-oriented structure that easily enables the creation of simulation libraries.

In the context of Product Line Verification, *virtual prototypes* (abstraction models) provide a promising approach, because they enable the verification and comparison of different system instances (product variants). Using SystemC as a modelling language, it is possible to support different abstraction models to verify software and hardware properties of the whole system. In accordance to the models, functional, and non-functional requirements can be verified. Therefore, it is, besides functional verification, best suited for performance verification, such as response time or bus load verification. The main challenge of this methodology is the handling of the variability through the generation of the DUT instance and the stimulus generation.

Virtual prototypes provide a good synergy with the formal verification approach described before. The basis of formal verification is a variability model specified with UML, SysML or other languages. Additionally, there are SIS Component Artefacts specified e.g., in a programming language such as C/C++ which are the realization artefacts and therefore the models used for formal verification correspond to these artefacts. These SIS Component Artefacts make up a simulation library that is used to generate the virtual prototype. By using this configurable set of artefacts to form a simulation library the correspondence between the realization artefacts and the simulation model is guaranteed. Using the variability model to assemble and parameterise the VP guarantees maximum reuse of existing models. The basis for virtual prototype-based verification differs from formal verification, but by reusing existing models for configuration, a good synergy is created. Similar to formal verification, the goal of virtual prototypes is to verify different SPL properties. Instead of using an automated reasoning engine, the virtual prototype is used and the verification is always based on dedicated execution traces. While formal verification directly works on system models/abstractions, the virtual prototype is used to evaluate different application scenarios/instances and verification is done on these traces. Execution traces of a virtual prototype run are verified using formal methods such as model checking. Another approach is to use LTL assertions to verify the expected behaviour during virtual prototype simulation (online verification).

6.3.6. Fault / complexity analysis of SPLs: Metrics, configuration mismatches, formal concept analysis

Several analysis approaches have been developed to measure the complexity of SPLs or to detect faults based on family-based [Thüm et al. 2012] analysis techniques. Metrics for software product lines can be distinguished into metrics for measuring the complexity of *variability models* and metrics for measuring the complexity of *code* and will be disgusted below in detail. Family-based fault detection will be discussed in the end of this section.
Metrics on variability models are based on the measurement of features, constraints, depth of tree [Bagheri et al. 2011, Mefteh et al. 2015], number of valid configurations [Bagheri et al. 2011, Mann et al. 2011], model complexity, and ratios between those elements. Besides the measurement of the total number of features [Apel et al. 2011, Bagheri et al. 2011, Berger and Guo 2014, Maâzoun et al. 2016, Mann et al. 2011, Mefteh et al. 2015, Nadi et al. 2014, Passos et al. 2011] in a feature model, there are also metrics to measure only the top level and leaf features

[Bagheri et al. 2011, Berger ang Guo 2014, Maâzoun et al. 2016, Mefteh et al. 2015], the number of configurable features [Leitner et al. 2012, Mann et al. 2011], features involved in cross-tree constraints [Sánchez et al. 2014], the number of independent configurable features [Mann et al. 2011], sometimes referred to as atomic sets [Zhang et al. 2004], or the number of variation points in a feature model [Mann et al. 2011, Sánchez et al. 2014]. Typed configuration languages offer the opportunity to measure distinct values for features of a certain type [Berger and Guo 2014], e.g., Boolean vs. Numeric features. Besides the measurement of the total number of cross-tree constraints [Berger and Guo 2014, Maâzoun et al. 2016, Passos et al. 2011], there exist measures based on feature groups [Berger and Guo 2014] or number of constraints by expressiveness [Passos et al. 2011]. Complexity of feature models are typically defined through adoption of classical cyclomatic complexity metric to feature models [Bagheri et al. 2011, Maâzoun et al. 2016, Mefteh et al. 2015, Sánchez et al. 2014], cross-tree constraint ratios [Bagheri et al. 2011, Berger and Guo 2014, Sánchez et al. 2014], the connection density of features [Bagheri et al. 2011, Maâzoun et al. 2016, Mefteh et al. 2015, Sánchez et al. 2014], or the ratio of optional features [Bagheri et al. 2011, Maâzoun et al. 2016, Mann et al. 2011, Mefteh et al. 2015]. Textual configuration languages offer additionally the opportunity to measure the ratio of typed or attributed features [Berger and Guo 2014]. However, while there exist already a variety of metrics to measure characteristics of complete variability models, there exist only very few metrics to measure characteristics of single features [Mann et al. 2011, Sánchez et al. 2014].
In the context of SPLs, code metrics must consider variability implementation mechanisms in order to operate on the complete platform instead of single instances only. Different metrics have been developed, which consider variability. Basic metrics are designed to measure size aspects of the implementation. These are lines of feature code [Bagheri et al. 2011, Couto et al. 2011, Hunsen et al. 2016, Liebig et al. 2010, Zhang et al. 2012, Zhang et al. 2013, Lopez-Herrejon et al. 2008], while other metrics are specifically designed for the compositional-based approach of feature-oriented programming [Abílio et al. 2016, Abílio et al. 2015, Macia et al. 2010].

Even if there are a lot of variability-aware metrics designed and used to measure artefacts of software product lines, only few of them have been evaluated with respect to their usefulness for characterizing qualitative characteristics of an implementation. Most studies combined several metrics to detect code smells in feature-oriented code [Abílio et al. 2015, Macia et al. 2010, Vale et al. 2015, Vale et al. 2015b] or analyse whether different metrics measure independent characteristics [Apel et al. 2011, Bagheri et al. 2011, Berger and Guo 2014, Liebig et al. 2010]. Bagheri and Gasevic study the ability of metrics models to measure the maintainability of feature models [Bagheri et al. 2011]. Sánchez et al. use metrics on feature models for test case prioritization while testing with sample products [Sánchez et al. 2014]. Finally, we are aware of only one study, which uses variability-aware code metrics to study the correlation between variability and vulnerabilities in software systems [Liebig et al. 2010]. Even if first studies have shown the potential of metrics for asset verification in some use cases, there is more research needed to facilitate useful usage for practitioners and industry.

Family-based approaches can be used complementary to metrics on software product line assets to detect failures, which may only exist in certain products. Prominent approaches are the analysis of dead code and unused features [Tartler et al. 2011], variability-aware type checking and control-flow analysis [Kästner et al. 2012], the detection of configuration mismatches [El-Sharkawy et al. 2017], or formal concept analysis [Lüdemann et al. 2016]. These techniques often combine the information from the feature model with variability information from code assets for the early detection of failures. However, especially when working with legacy systems, a feature model is not always explicitly available.

6.3.7. **Evolutionary analysis**

Approaches for evolutionary SPL analysis/verification should focus, in particular, on the changes introduced to a SPL. For example, these changes can be related to the variability model, the source code, the build process, or any combination of them. Some approaches introduce concepts which partially address this issue:

- For efficient *feature model analysis* in evolving SPLs, Schröter et al propose in [Schroeter et al. 2016] feature model interfaces, which only contain relevant features for a composition of two feature models. As the number of relevant features in an interface is typically smaller than in the original feature model, the effort for verifying the composed feature model can be reduced. In particular, if a change does not affect the features of an interface, there is no need to re-verify the composed feature model. Nieke et al. introduce in [Nieke et al. 2016] their concept of Temporal Feature Models (TFMs) and corresponding evolution operations to avoid changes, which will break specific configurations of selected features. Further, Botterweck et al. discuss in [Botterweck 2009] their EvoFM approach as a basis for analysing the impact of evolution on the variability model. While all three approaches focus on feature models, which typically consist of pure Boolean variability, other approaches might aim at such support for variability models of higher expressiveness, e.g. including numerical values and operations in constraints.
- Ribeiro et al. present in [Ribeiro et al. 2011] an approach for the automatic impact detection of feature changes. We share the intent to use change information reengineered from code with information derived from the variability model to aid in the correct evolution of product lines. However, the authors mainly focus on *code and code dependencies* while the analysis of the underlying variability model is out of scope. Further, this approach relies on the application of emergent interfaces [Ribeiro et al. 2010], which limits the approach to SPLs supporting this concept.
- The *combined analysis* of the impact of feature changes is the subject to the work by Ziegler et al. [Ziegler et al. 2016] and Tartler et al. [Tartler et al. 2014]. The former derives traceability links among configuration options in the variability model and the code assets. Their objective is to identify the affected source code files when changing an option in the variability model. The latter identifies all modified files of a SPL patch and derives a set of configurations, which have to be checked for validity of these modifications. While the derivation of the impact of arbitrary changes to SPL assets is important, the approach by Ziegler et al. targets the Linux kernel with its specific characteristics. Tartler et al. require the availability of a build system to check if affected configurations still yield the desired products.

The individual capabilities of the concepts introduced above form a basis for the development of an integrated and comprehensive evolutionary SPL analysis approach in REVaMP². We aim at reusing, combining and extending these concepts for the requirements imposed by the project context.

6.4. Open challenges in product line verification

In the area of PL verification, a key problem is that no class of constraint solver scales well on all classes of PL verification problems. A practical industrial strength PL verification service must thus include a variety of solvers allowing its user to choose the best one for any particular class of problem (or even automate the selection). Thus, various solvers should be included, like a CLP solver or an SMT solver and their performance on the PL verification use cases provided by industrial partners should be compared. We might be inspired by the Betty framework [Segura et al. 2012], Betty is a framework supporting the benchmarking and testing on the analyses of feature models and a similar benchmarking frameworks might be provided for SPL verification tasks. Further, we should also evaluate soft constraint optimization techniques such as multi-objective branch and bound and bucket elimination on PL optimization use cases.
Some verification aspects that have so far each achieved only very little interest in the scientific community, but are very important from a practical and industrial perspective. These include: incremental analysis in order to minimize the effort and time required for verification of a software increment and identifying variability smells, i.e., cases which are not errors in a formal sense like inconsistencies, but may point to software (configuration) issues. For some examples of the initial attempts to deal with these cases, we reference to El-Sharkawy et al. 2017 [El-Sharkawy et al. 2017].

7. Tool integration

7.1. Introduction

Tool integration is an ongoing topic for model-based system development since the late 80s. According to Wassermann et al. [Wassermann 1990] five dimensions are described that need to be addressed for tool integration. Table 7.1 summarizes these dimensions.

Integration Dimension	Description
Platform Integration	A common framework and abstraction layer is needed in order to provide network or operating system transparency.
Data Integration	Data exchange between different tools or the definition of relationships among data objects produced by different tools has to be enabled and ensured.
Presentation Integration	To provide a common look and feel the same presentational elements have to be used, regardless of the concrete tool functionality.
Control Integration	This kind of integration allows coordinating the control flow between tools. It is also the ability of tools to perform notifications to other tools as well as to run other tools.
Process Integration	Process integration is concerned with the orchestration of the various activities and the coordination of the interaction with the humans executing the activities within the processes.

Table 7.1: Integration Dimensions

A general architecture addressing the various aspects of tool integration is shown in Figure 7.1.

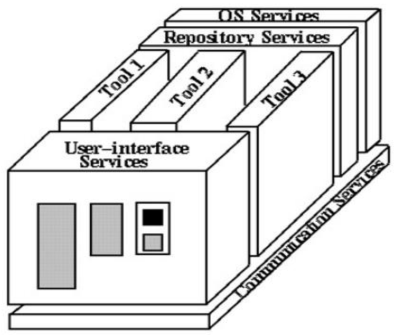

Figure 7.1: Toaster Model [NIST 1993]

The toaster model introduces different types of services to addresses tool integration aspects:

- **OS Services,** for Platform integration
- **Repository Services,** Data integration
- **User-Interface Services,** for Presentation integration
- **Communication Services,** for Control and Process integration

7.2. Characteristics to comparatively evaluate Tool Integration approaches

Type: The first characteristic to compare tool integration approaches is to determine the type of the approach.

Integration Dimension: Which kind of integration according to Wassermann et al. is addressed by the Methodology or Tool.

Implementation: Check whether there are any restrictions regarding the implementation of an integration, e.g., the implementation language (C/C++, C#, Java, etc.).

Core integration concept: How are new tools and services integrated in a tool-chain and how can existing tools and services be maintained or exchanged.

Data exchange concept: How does the data exchange concept look like.

- Internal model(s): each tool to be integrated needs to conform to the exact elements of that models
- Meta-Model: each tool to be integrated needs to refine (some of) the elements of that model

Restrictions/Limitations: What are the boundaries of instantiating the integration approach or applying the integration tool?

7.3. Tool Integration approaches

In the context of data and presentation integration for systems engineering process, it is necessary to provide the proper mechanisms to: 1) represent under a common and shared data model any piece of knowledge, and 2) provide advanced retrieval mechanisms elevating the meaning of information resources from text-based descriptions to concept-based ones.

7.3.1. Generic Tool Integration standards

7.3.1.1. OSLC

In the context of software and system artefacts reuse, the Open Services for Lifecycle Collaboration (OSLC) initiative [Ryman et al. 2013] is a joint effort between academia and industry to boost data sharing and interoperability among applications by applying the Linked Data principles [Bizer et al. 2009]: "*1) Use URIs as names for things. 2) Use HTTP URIs so that people can look up those names. 3) When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL) and 4) Include links to other URIs, so that they can discover more things*". Led by the OASIS OSLC working group, OSLC is based on a set of specifications that take advantage of web-based standards such as the Resource Description Framework (RDF) (Hayes 2004) and the Hypertext Transfer Protocol (HTTP) to share data under a common data model (RDF) and protocol (HTTP). Every OSLC specification defines a *shape* for a particular type of resource. For instance, requirements, changes, test cases, models (the OSLC-MBSE specification Model-Based Systems Engineering by the Object Management Group) or estimation and measurement metrics, to name a few, have already a defined shape (also called OSLC Resource Shape).

Thus, tools for supporting Application Life-cycle Management (ALM) or Product Life-cycle Management (PLM) have now an agreement on what data must be shared, and how. In the knowledge management framework, the *Assets Management* and the *Tracked Resource Set* are the most convenient specifications for the purpose of managing artefacts. However, there are many artefacts generated during the development lifecycle, which may not fit to existing shapes or standard vocabularies. Simulation models, business rules or physical circuits are examples of potential artefacts whose OSLC resource shape is not defined yet. Furthermore, some common and useful services such as indexing, naming, retrieval, quality assessment, visualization or traceability must be provided by all tool vendors, creating a tangled environment of query languages, interfaces, formats and protocols. Figure 7.2 gives an overview about OSLC.

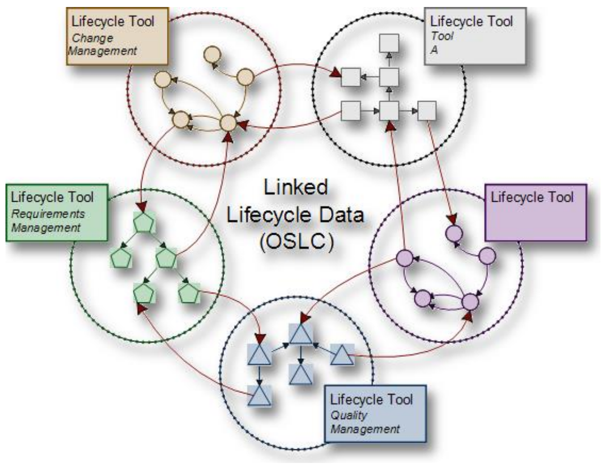

Figure 7.2: OSLC Overview

Therefore, one of the current trends in systems and software development lies in boosting interoperability and collaboration through the sharing of existing artefacts under common data models, formats and protocols. In this context, OSLC is becoming a collaborative software ecosystem [Manikas et al. 2013] for SPLs through the definition of data shapes that serve as a contract to get access to information resources through HTTP-based services.

In particular, the Representational State Transfer (REST) software architecture style is used to manage information resources that are publicly represented and exchanged in RDF. Obviously, OSLC represents a big step towards the integration and interoperability between the agents involved in the development lifecycle.

However, RDF has also been demonstrated [Powers 2003] to contain some restrictions to represent certain knowledge features such as N-ary relationships [Noy 2006], practical issues dealing with reification [Nguyen 2014] and blank nodes [Mallea et al. 2014]. On the other hand, some common services such as indexing, retrieval or quality assessment of any kind of information resource are restricted to the internal storage and the query capabilities offered by each particular tool (usually a SPARQL interface).

7.3.1.2. Semantic Technologies and Semantic Web

The Semantic Web, coined by Tim Berners-Lee [Berners-Lee et al. 2001] proposed a commitment from both academia and industrial areas with the objective of elevating the abstraction level of web information resources. The Resource Description Framework (RDF) [Hayes 2004], based on a graph model, and the Web Ontology Language (OWL) [Hitzler et al. 2009], designed to formalize, model, and share domain knowledge, are the two main ingredients to reuse information and data in a knowledge-based realm. Thus, data, information and knowledge can be easily represented, shared, exchanged and linked to other knowledge bases through the use of Uniform Resource Identifiers (URIs), more specifically HTTP-URIs. As a practical view of the Semantic Web, the Linked Data initiative emerges to create a large and distributed database on the Web by reusing existing and standard protocols. In order to reach this major objective, the publication of information and data under a common data model (RDF) with a specific formal query language (SPARQL) [Bizer and Cyganiak 2009, Hogan et al. 2012] provides the required building blocks to turn the Web of Documents into a real database or Web of Data. In this context, a large body of work can be found in different domains such as Geography, Bibliography, e-Government, e-Tourism or e-Health, all of them having common needs, such as: interoperability among tools, different schemes or data models, or cross-cutting services (index and search).

On the other hand, in recent times we have seen the deployment of service-oriented computing [Krafzig et al. 2005] as a new environment to enable the reuse of software in organizations (process integration). In general, a service oriented architecture comprises an infrastructure (e.g. Enterprise Service Bus) in which services (e.g. software as web services) are deployed under a certain set of policies. A composite application is then implemented by means of a coordinated collection of invocations (e.g. Business Process Execution Language). In this context, Enterprise Integration Patterns (EAI) [Hohpe and Woolf 2004] have played a key role to ease the collaboration among services. Furthermore, existing W3C recommendations such as the Web Services Description Language (WSDL)[1] or the Simple Object Access Protocol (SOAP)[2] have improved interoperability through a clear definition of the input-/output-interface of a service and communication protocol.

In order to improve the capabilities of this type of web services, semantics was applied to ease some tasks such as discovery, selection, composition, orchestration, grounding and automatic invocation of web services. The Web Services Modelling Ontology (WSMO) [Roman et al. 2005] represented the main effort to define and to implement semantic web services using formal ontologies. OWL-S (Semantic Markup for Web Services), SA-WSDL (Semantic Annotations for WSDL) or WSDL-S (Web Service Semantics) were other approaches to annotate web services, by merging ontologies and standardizing data models in the web services realm.

However, these semantics-based efforts did not reach the expected outcome of automatically enabling enterprise services collaboration. Formal ontologies were used to model data and logical restrictions that were validated by formal reasoning methods implemented in semantic web reasoners. Although this approach was theoretically very promising since it included consistency checking or type inference, the reality proved that the supreme effort to create formal ontologies in different domains, to make them interoperable at a semantic level, and to provide functionalities

[1] https://www.w3.org/TR/wsdl/
[2] https://www.w3.org/TR/soap/

such as data validation, was not efficient. More specifically, it was demonstrated [Rodríguez et al. 2012] that, in most of cases, data validation, data lifting and data lowering processes were enough to provide an interoperable environment.

That is why the approach based on the W3C recommendations, WSDL+SOAP, fulfilled most of these requirements with a huge industrial and technological support. However, the lack of agreement on the schemas to be shared (any service provider offered their own schema) and the use of a restricted data model such as XML was still present with the result of preventing a paradigm shift.

In the specific case of software engineering and reuse, the application of semantics-based technologies has also been focused on the creation of OWL ontologies [Castañeda et al. 2010] to support requirements elicitation, and to model development processes [Kossmann et al. 2008] or Model Driven Architecture [Gaševic et al. 2006], to name just a few. These works leverage ontologies to formally design a meta-model and to meet the requirements of knowledge-based development processes.

Taking advantage of the Linked Data principles and Web standards and protocols, the OSLC effort (described in previous subsection) emerges to create a family of web-based specifications for products, services and tools that support all the phases of the software lifecycle. To do so, OSLC defines several specifications based on the aforementioned Linked Data principles. Similar to OSLC, Agosense Symphony[3] offers an integration platform for application and product lifecycle management, covering all stages and processes in a development lifecycle. It represents a service-based solution widely used in the industry due to the possibility of connecting existing tools. WSO2[4] is another middleware platform for service-oriented computing based on standards for business process modelling and management. However, it does not offer standard input-/output-interfaces based on lightweight data models and software architectures such as RDF and REST. Other industry platforms such as PTC Integrity[5], Siemens Team Center[6], IBM Jazz Platform[7] or HP PLM[8] are now offering OSLC interfaces for different types of artefacts.

However, data exchange does not necessarily imply knowledge management. From service providers to data items, a knowledge strategy is required to really represent, store and search software artefacts metadata and contents. In this light, the OSLC initiative is currently following this approach, having impact on the main players of the software and systems industry. Nevertheless, it only covers a restricted type of artefacts and some cross-cutting and basic services for reuse, such as indexing or retrieval, must be provided by all third-parties.

Lastly, a system and software knowledge repository to implement a real knowledge strategy for software reuse should be based on the following three requirements:

1. A language for representing any artefact's metadata and contents.
2. A system for indexing and retrieval.
3. A standard input/output interface (data shape+REST+RDF) to share and exchange artefact metadata and contents.

---

[3] http://www.agosense.com/english/products/agosensesymphony/agosensesymphony
[4] http://wso2.com/
[5] http://www.ptc.com/application-lifecycle-management/integrity
[6] http://www.plm.automation.siemens.com/en_us/products/teamcenter/
[7] https://jazz.net/
[8] http://www8.hp.com/us/en/business-services/it-services.html?compURI=1830395

7.3.1.3. Formal Ontologies Vs Data Shapes

In the early days of the Semantic Web, formal ontologies [Benjamins et al. 1998] designed in RDFS (Resource Description Framework Schema) or OWL were the key technologies to model and share knowledge. From upper ontologies such as DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) or SUMO (The Suggested Upper Merged Ontology) to specific vocabularies such FOAF (Friend Of A Friend) or SKOS (Simple Knowledge Organization System), the process to share knowledge consisted in designing a formal ontology for a particular domain and populate data (instances) for that domain. Although the complete reuse of existing ontologies was expected, the reality demonstrated that every party willing to share knowledge and data would create its own ontologies. Thus, the main idea behind web ontologies was partially broken since just a few concepts were really reused.

Once the Linked Data initiative emerged to unleash the power of existing databases, a huge part of the Semantic Web community realized that a formal ontology was not completely necessary to exchange data and knowledge. Taking into account that ontologies were still present, these efforts were based on validating data consistency [Baclawski et al. 2002] through the execution of procedures such as: 1) reasoning processes to check consistency, and 2) rules, mainly in SWRL (Semantic Web Rule Language) or SPARQL [Bizer and Cyganiak 2009, Hogan et al. 2012]. These procedures are not recommended, due to performance issues, when a huge number of instances are available. As a new evolution, then, the community realized that ontology-based reasoning was not the most appropriate method for data validation.

That is why in recent times the RDF community has seen an emerging interest to manage and validate RDF datasets according to different shapes and schemes, see Table 7.1. New specifications and methods for data validation are being designed to turn reasoning-based validation into a kind of grammar-based validation. These methods take inspiration from existing approaches in other contexts such as DTD (Document Type Definition), XML-Schema or Relax NG (REgular LAnguage for XML Next Generation) for XML, or DDL (Data Definition Language) for SQL (Structured Query Language).

The W3C launched (2014) the RDF Data Shapes Working group and the ShEX (Shape Expressions) language [Boneva et al. 2014, Cyganiak and Reynolds 2014, Gayo et al. 2015]. Both are formal languages for expressing constraints on RDF graphs including cardinality constraints as well as logical connectives for disjunction and polymorphism. As other examples of data validation, OSLC Resource Shapes [Ryman et al. 2013], Dublin Core Description Set Profiles [Coyle and Baker 2013], and RDF Unit [Kontokostas et al. 2014] are also constraint languages for domain specific RDF resources.

Following a more classical approach for RDF data validation, the Pellet Integrity Constraints is an extension of the existing semantic web reasoner [Sirin et al. 2007] that interprets OWL ontologies under the Closed World Assumption with the aim of detecting constraint violations in RDF data. These restrictions are also automatically translated into SPARQL queries. This approach has been implemented on top of the Stardog[9] database, enabling users to write constraints in SPARQL, SWRL or as OWL axioms. Finally, the SPIN language [Knublauch et al. 2011] also makes use of SPARQL (mainly its syntax) to define constraints on RDF-based data that can be

[9] http://stardog.com/

executed by systems supporting SPIN (SPARQL Inferencing Notation), such as the TopBraid's tool-chain.

Process	Type	Creation	Scope	References
Consistency check	Vocabulary-based	Semantic Web reasoner	RDF datasets	[Baclawski et al. 2002]
Data validation (integrity)	Query-based	Hand-made RDF templates	RDF datasets	[Bizer and Cyganiak 2009]
	Vocabulary-based	Hand-made	RDF datasets	[Hogan et al. 2012]
	Vocabulary-based	Hand-made or automatically generated by an OSLC API	OSLC Resource Shape	[Ryman et al. 2013]
	Vocabulary-based	Hand-made	Dublin Core Description Set Profiles	[Coyle and Baker 2013]
	Query-based	RDF Unit (test creation)	RDF datasets	[Kontokostas et al. 2014]
	Query-based (generated from ShEX expressions)	Automatic generation of SPARQL queries	RDF datasets	[Boneva et al. 2014, Cyganiak and Reynolds 2014, Gayo et al. 2015, Alvarez-Rodríguez et al. 2013]
	Vocabulary and Query-based	Automatic generation of SPARQL queries	OWL and RDF under Closed World Assumption	[Sirin et al. 2007]
	Query-based	SPIN language + SPARQL queries	RDF datasets	[Knublauch et al. 2011]

Table 7.2: Summary of the main approaches to validate RDF-encoded data

In conclusion, the relevance of data validation to exchange RDF-encoded data is clear. RDF Data Shapes in its different flavours, such as OSLC Resource Shapes, are becoming the cornerstone for boosting interoperability among agents. It is also clear that ontologies are becoming less important although a combined approach (data shapes and a formal ontology) can provide important benefits in terms of data validation and knowledge inference (if needed). In the context of software reuse, as it has been previously outlined, software artefacts must take advantage of new technologies to enable practitioners the automatic processing of exchanged data.

On the other hand, one of the cornerstones to provide data and knowledge management services for systems development lies in the selection of an adequate knowledge representation paradigm. After a long time of research [Hull and King 1987], this problem still persists, since the choice of a suitable representation format (and syntax) can be reached in several ways.

Different types of knowledge require different types of representation [Groza et al. 2009], including in some cases inference capabilities. In this light, expressions, rule-based systems, regular grammars, semantic networks, object-oriented representations, frames, intelligent agents, or case-based models, to name a few, are some of the main approaches to information and knowledge modelling.

According to the current context for software reuse, the next knowledge representation paradigms (focusing on web-oriented technologies) have been selected for comparison:

- The Resource Description Framework [Hayes 2004] (RDF) is a framework for representing resource information in the Web using a directed graph data model. The core structure of the abstract syntax is a set of triples, each consisting of a subject, a predicate and an object. A set of such triples is called an RDF graph. An RDF graph can be visualized as a nodes and directed-arcs diagram, in which each triple is represented as a node-arc-node link. RDF has been used as the underlying data model for building RDFS/OWL ontologies, gaining momentum in the web-based environment due to the explosion of the Semantic Web and Linked Data initiatives that aim to represent and exchange data (and knowledge) between agents and services under the web-based protocols.
- RDF Schema (RDFS) [Brickley and Guha 2014] provides a data-modelling vocabulary for RDF data. It can be seen as a first try to support the creation of formal and simple ontologies with RDF syntax. RDFS is a formal and simple ontology language in which it is possible to define class and property hierarchies, as well as domain and range constraints for properties. One of the benefits of this property-centric approach is that it allows anyone to extend the description of existing resources.
- OWL (Web Ontology Language) [Hitzler et al. 2009] is an ontology language for capturing meaningful generalizations about data in the Web. It includes additional constructors for building richer class and property descriptions (vocabulary) and new axioms (constraints), along with a formal semantics. OWL 1.1 consists of three sub-languages with different levels of expressivity: 1) OWL Lite, 2) OWL DL (Description Logic) and 3) OWL Full.
- The OWL 2.0 [Hitzler et al. 2009] family defines three different profiles: OWL 2 EL (Expressions Language), OWL 2 QL (Query Language) and OWL RL (Rule Language). These profiles can be seen as a syntactic restriction of the *OWL 2 Structural Specification* and more restrictive than OWL DL. The use of profiles is motivated by the needs of different computational processes. OWL EL is designed for enabling reasoning tasks in polynomial time. The main aim of OWL 2 QL is to enable conjunctive queries to be answered in LogSpace using standard relational database technology. Finally, OWL 2 RL is intended to provide a polynomial time reasoning algorithm using rule-extended database technologies operating directly on RDF triples. In conclusion, OWL 2.0 adds new functionalities regarding OWL 1.x. Most of them are syntactic sugar but others offer new expressivity (Hitzler et al. 2009): keys, property chains, richer datatypes, data ranges, qualified cardinality restrictions, asymmetric, reflexive, and disjoint properties; and enhanced annotation capabilities.

- RIF Core (Rule Interchange Format) [Boley et al. 2013] comprises a set of dialects to create a standard for exchanging rules among rule systems, in particular among Web rule engines. RIF was designed for exchanging rather than developing a single one-fits-all rule language.
    - RIF dialects fall into three broad categories: first-order logic, logic-programming, and action rules. The family of dialects comprises: 1) logic-based dialects (RIF-BLD) including languages that employ some kind of logic such as First Order Logic (usually restricted to Horn Logic) or non-first-order logics; 2) rules-with-actions (RIF-PRD) dialects comprising rule systems such as Jess, Drools and JRules as well as event-condition-action rules such as Reaction RuleML and XChange. RIF also defines compatibility with OWL and serialization using RDF.
- The SRL (System Representation Language, former RSHP [Llorens et al. 2004] [Díaz et al. 2005]) is based on the idea that whatever information can be described as a group of relationships between concepts. Therefore, the leading element of an information unit is the relationship. For example, Entity/Relationship data models are certainly represented as relationships between entity types; software object models can also be represented as relationships among objects or classes; in the process modelling area, processes can be represented as causal/sequential relationships among sub-processes. Moreover, UML (Unified Modeling Language) or SysML (Systems Modeling Language) meta-models can also be modelled as a set of relationships between meta-model elements.
    - SRL also includes a repository model to store information and relationships with the aim of reusing all kind of knowledge chunks. Furthermore, free text information can certainly be represented as relationships between terms by means of the same structure. Indeed, to represent human language text, a set of well-constructed sentences, including the subject+verb+predicate (SVP) should be used. The SVP structure can then be considered as a relationship typed V between the S and the predicated P. More specifically, the SRL formal representation model is based on the following principles:
    - The main description element is the relationship since it is the element in charge of linking knowledge elements.
    - A Knowledge Element (KE) is an atomic knowledge component that appears into an artefact and that is linked by one or more relationships with other KEs to build information. It is defined by a concept, and it can also be an artefact (an information container found inside a wider artefact). A concept is represented by a normalized term (a keyword coming from a controlled vocabulary, or domain). Artefacts are knowledge containers of KEs and their relationships.
    - In SRL, the simple representation model for describing the content of whatever artefact type (requirements, risks, models, tests, maps, text docs or source code) should be:
        - SRL representation for artefact α = $i_{\alpha} = \{(RSHP_1), (RSHP_2), \ldots, (RSHP_n)\}$, where every single SRL is called SRL-description and must be described using KE.
        - One important consequence of this representation model is that there is no restriction to represent a particular type of knowledge. Furthermore, SRL has been used as the underlying information model to build general-purpose indexing and retrieval systems, domain representation models (Díaz et al. 2005), approaches for quality assessment of requirements and knowledge management tools such as knowledgeMANAGER [The Reuse Company Inc. 2014] .

A plethora of other knowledge representation mechanisms and paradigms can be found as it is presented below. However, there is only a subset that satisfy: 1) a language for representing any artefact metadata and contents; 2) a system for indexing and retrieval and 3) a standard input/output interface (data shape+REST+RDF) to share and exchange artefact metadata and contents.

- The SBVR (Semantics of Business Vocabulary and Rules). This is an OMG standard to define the basis for formal and detailed natural language declarative description of a complex entity.
- The Ontology Definition Meta-model (ODM). It is an OMG standard for knowledge representation, conceptual modelling, formal taxonomy development and ontology definition. It enables the use of a variety of enterprise models as starting points for ontology development through mappings to UML and MOF. ODM-based ontologies can be used to support: 1) interchange of knowledge; 2) representation of knowledge in ontologies and knowledge bases; and 3) specification of expressions that are the input to, or output from, inference engines.
- The Reusable Asset Specification (RAS), an OMG standard that addresses the engineering elements of reuse. It attempts to reduce the friction associated with reuse transactions through consistent, standard packaging.

In order to select the proper mechanism for knowledge representation of system and software artefacts, the following points must be considered:

- RDF is based on a directed graph and can only represent binary relationships (unless reification and blank nodes are used). As a representation mechanism, RDF presents some restrictions that have been outlined in several works [Powers 2003]. For instance, N-ary relationships [Noy and Rector 2006], practical issues dealing with reification [Nguyen et al. 2014] and blank nodes [Mallea et al. 2011] are well-known RDF characteristics that do not match the needs of a complete framework for knowledge representation. Furthermore, RDF is built on two main concepts: resources and literals. However, a literal value cannot be used as the subject of an RDF triple. Although this issue can be overcome using a blank node (or even reification) and the property *rdf:value*, it adds extra complexity for RDF users. Finally, RDF has been designed to represent logical statements, constraining also the possibility of representing other widely used paradigms such as objects or entity-relationships models. Due to these facts, it seems clear that RDF can be used for exchanging data, but it is not the best candidate for knowledge representation.
- RDFS is a good candidate for modelling lightweight formal ontologies including some interesting capabilities close to object-oriented models. It can be serialized as RDF, but this issue can also be a disadvantage due to expressivity restrictions of RDF. Again, RDFS has been designed for expressing logical statements that describe web resources, so its use for other types of information is not advisable.
- Building on the previous discussion, OWL presents a family of logic dialects for knowledge representation. It is based on strong logic formalisms such as Description Logic or F-Logic. It was also designed for asserting facts about web resources although it can be used as a general logic framework for any type of knowledge. One of the main advantages of OWL is the possibility of performing reasoning processes to check consistency or infer types. However, reasoning can be considered harmful in terms of performance and most often it is not necessary when data is exchanged. Also, OWL is not the best candidate for data validation, a key process in knowledge exchange.

- RIF Core and the family of RIF dialects have been included in this comparison due to the fact that most domain knowledge is embedded in rules. Nevertheless, RIF was not designed for data validation and its acceptance is still low (just a few tools export RIF and even fewer are capable of importing RIF files). On the other hand, RIF makes use of the web infrastructure to exchange rules, which also means that this environment is a very good candidate to exchange data, information and knowledge.
- SRL, based on relationships, allows domain experts to create relationships among terms, concepts or even artefacts (containers). It provides a framework for knowledge representation with capabilities for expressing any kind of cardinality and N-ary relationships. SRL is based on undirected property graphs, enhancing expressivity. Although it has not been directly designed for data validation, its meta-model allows the possibility of checking cardinality, value, domain and range restrictions. One of the strong points of SRL is the native support of a tool such as knowledgeMANAGER and the possibility of automatically providing semantic indexing and retrieval mechanisms. Both have generated a strong acceptance in industry for requirements authoring and quality checking [Génova et al. 2013]. As minor drawbacks, this tool was not conceived to be used in a web environment and it is not a standard; nevertheless, it can export/import RDFS and OWL ontologies.
- Finally, as a general comment, there is also a lack of tools working natively on RDF. Generally speaking, RDF was conceived to exchange information over the web. Although some RDF repositories can provide capabilities for indexing and searching RDF resources through a SPARQL interface, experience has demonstrated that most often RDF is translated into the native data model of a tool.

Since formal ontologies and reasoning processes are not completely necessary and, instead, data validation is a key aspect for boosting interoperability, it also seems clear that SRL fits perfectly to the major objective of knowledge representation. However, we note that the use of formal RDFS or OWL ontologies is not incompatible with SRL, but possible and enriching. RDFS and OWL are languages for building domain vocabularies, while SRL is already a domain vocabulary for knowledge representation, so that it is possible to define SRL through a formal RDFS or OWL ontology.

7.3.2. **Variability-specific Tool Integration standards**

7.3.2.1. The Common Variability Language (CVL)

There are two different ways of specifying variability: as feature models that are independent of any design or implementation model; or as variability models related to a base model [Haugen et al. 2008]. Regarding the second option, two main approaches should be considered: (1) Annotating the base model by means of extensions to the modelling language (e.g. UML profiles); or (2) making separate variability models that apply to the base model. The Common Variability Language (CVL) is an example of the second approach. The history of CVL and the last version of the language (OMG Revised Submission) is available[10] and SINTEF continued working from CVL into BVR (Base Variability Resolution) [Vasilevskiy et al. 2015].

The CVL provides a generic and separate approach for modelling variability in models defined by a Domain Specific Language (DSL). CVL can be applied to models in any DSL defined by a meta-model that conforms with the Meta Object Facility (MOF). Figure 7.3 shows an overview of the

[10] http://variabilitymodeling.org

CVL. First, CVL allows to specify variants of a base model (in any DSL conforming to MOF), to specify resolution models (specifying the set of choices of variability that wants to be included in the resulting model) and to execute those resolutions to obtain resolved models by replacing variable parts of the base model with alternative model replacements found in a library model.

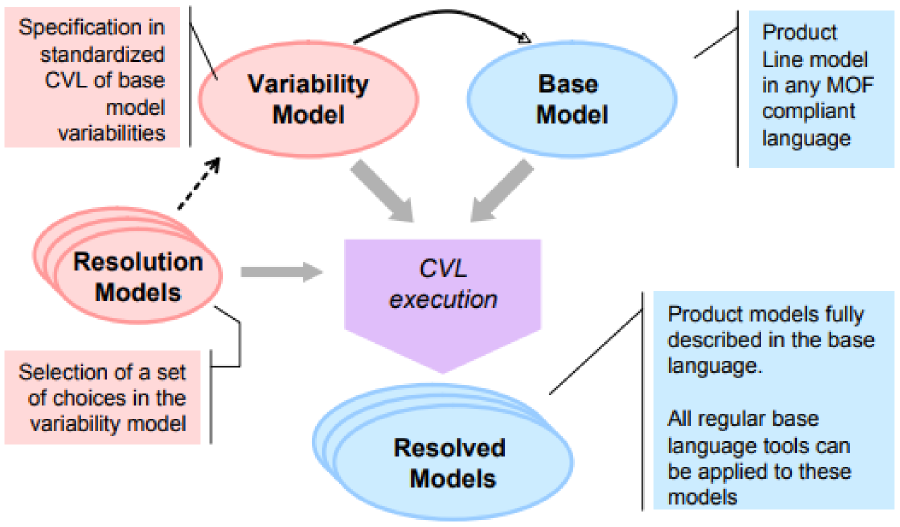
Specification in standardized CVL of base model variabilities
Variability Model
Base Model
Product Line model in any MOF compliant language
Resolution Models
CVL execution
Selection of a set of choices in the variability model
Resolved Models
Product models fully described in the base language.
All regular base language tools can be applied to these models

Figure 7.3: Common Variability Language as specified in the Request for Proposals [OMG-CVL 2012]

Variability in CVL is specified through two conceptual layers:
- feature specification layer where variability can be specified following a feature model syntax
- product realization layer where variability specified in terms of features is linked to the actual models in terms of placements, replacements and substitutions

In the context of Software Product Lines, the feature specification layer expresses high level features that the user would like to include, similar to feature diagrams. The concepts of CVL (e.g. type, composite variability, constraint and iterator) can be used to mimic feature diagrams (e.g. mandatory/optional feature, feature dependencies, XOR/OR cardinality) [Svendsen et al. 2010]. The product realization layer further defines low level operations necessary to transform the base model to a resolved product model, including details on how the transformation should be executed.

The base model is a model described by a given DSL which serves as a base for different variants defined over it. In CVL the elements of the base model that are subject to variations are the placement fragments (hereinafter placements). A placement can be any element or set of elements that is subject to variation. To define alternatives for a placement we use a replacement library, which is a model described in the same DSL as the base model that will serve as basis to define alternatives for a placement. Each one of the alternatives for a placement is a replacement fragment (hereinafter replacement). Similar to placements, a replacement can be any element or set of elements that can be used as variation for a replacement.

In CVL, variants of the base model are specified by means of fragment substitutions. Each substitution references to a placement and a replacement and includes the information necessary to substitute the placement by the replacement. In other words, each placement and replacement is defined along with its boundaries, which indicate what is inside or outside each fragment (placement or replacement) in terms of references among other elements of the model. Then, the substitution is defined with the information of how to link the boundaries of the placement with the boundaries of the replacement. When a substitution is materialized (selected in a resolution model and executed), the base model (with placements substituted by replacements) continues to conform to the same meta-model.

7.3.2.2. Variability Exchange Language (VEL)

The purpose of the Variability Exchange Language (VEL) [Schulze et al. 2015] is to support the information exchange among variant management tools on the one hand and system development tools on the other hand. The essential tasks of a variant management tool are to represent and analyse the variability of a system abstractly and to define system configurations by selecting the desired system features. A system development tool captures information of a specific kind, such as requirements, architecture, component design, or tests. In order to support the development of variable systems, development tools either have to offer the capability to express and deal with variability directly, or an additional piece of software like an add-on must be provided that adds this capability to the development tool.

To interconnect variant management with systems development, the information exchange among the corresponding tools must be established. A variant management tool must be able to read or extract the variability from a development tool and to pass a configuration, i.e., a set of selected system features, to the development tool. Up to now, the interfaces that support this information exchange are built for each development tool anew. With a standardized Variability Exchange Language, a common interface can be defined that is implemented by the development tools and used by the variant management tools. The integration of variant management tools with systems development tools via this interface enables a continuous development process for variable systems and supports a flexible usage of tools for this process.

Using VEL the following types of variability information can be described:

- Structural Variation Points: These are variation points defining structural variability, i.e., they point to assets which are not necessarily part of each variant. A structural variation point can be optional (an associated asset either exists or exists not in a variant) or alternative (one of a specific set of variations will be selected, each associated with an asset that can be part of a variant).
- Parameter Variation Point: These are variation points defining a variable parameter value (e.g. a number). The value can be calculated by an expression, or there is a set of alternative values to choose from.

Each variant of a variation point (one for optional variation points, many for alternative variation points) has an expression. Existence expressions (for variations of structural variation points and alternative values of parameter variation points) will return a Boolean value, defining the selection state. Otherwise, expressions for calculations return a specific value for a given parameter. The language of the expression is not defined in VEL, so any complex language can be supported.

Variation Points support also hierarchies to represent nested asset structures and among variations dependencies can be defined. Next to the descriptions of the variation points, VEL can also describe variation point configuration, i.e., a selection of a variation for each variation point, defining a specific variant.

7.3.3. **Tool Integration Frameworks**

7.3.3.1. ModelBus

ModelBus[11] is a flexible and highly customizable Open Source model driven tool integration framework for engineering processes that allows building a seamlessly integrated tool environment. ModelBus has a bus like communication architecture. Using some integrated basic core services, e.g. the Model Storage and Notification, it is very easy to add further tools, due to documented service interfaces and well establishes standards.
It is possible to use heterogeneous tools for the whole lifecycle. Suitable adapters integrate these tools into ModelBus by translating the tool syntax into one understood by ModelBus. This allows communication between these heterogeneous tools and ModelBus. If one tool is successfully integrated into the ModelBus the functionality of this tool is available for other services and vice versa. COTS (Commercial of the shelf) tools can be integrated using these adapters.

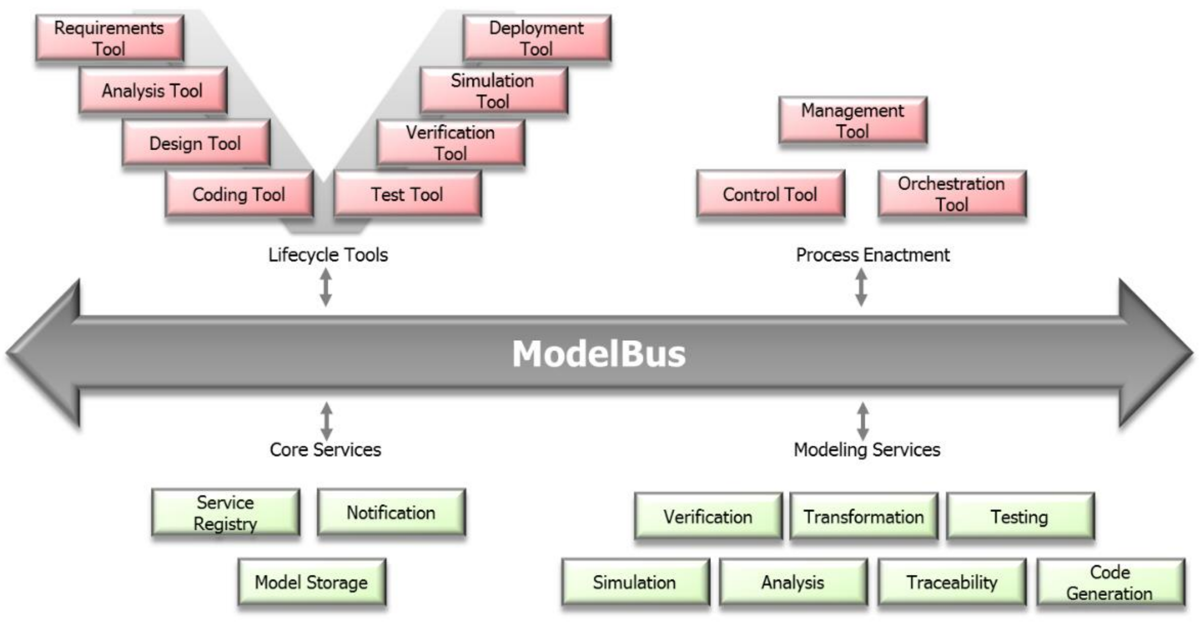

Figure 7.4.: ModelBus

ModelBus is based on SOA principles and supports the automation of development tasks to reduce manual developer effort. For example, it is possible to create single tasks as services, associate them as a new service by other services as a concatenation of services and execute them. This could be done either by manual commands or other services.
ModelBus is usable in any kind of development environment and is based on industrial standards (e.g., UML, BPMN, BPEL, MOF EMF) and has an internal model repository to document check-ins, partial check-out, merges of models. Additionally, it has an Eclipse integration, ModelBus TeamProvider, to support use this technology in Eclipse based tools. Figure 7.4 shows a ModelBus scheme. Furthermore, ModelBus works with the concept of freedom of data. This means that data created in or imported into ModelBus should be suitable for the whole development process in the ModelBus context.

[11] https://www.modelbus.org/

7.3.3.2. IBM Jazz

The IBM Jazz platform[12] is an integration based on open flexible services and internet architecture. The Jazz platform provides the infrastructure for the integration of several types of lifecycle tools. The platform consists of an architecture and set of application frameworks and toolkits as shown in Figure 7.5. The current Jazz platform supports a number of IBM Rational products (all individually licensed) with the application frameworks and toolkits used by IBM developers. The long-term vision is for non-IBM developers to develop tool integrations to support more integration services.

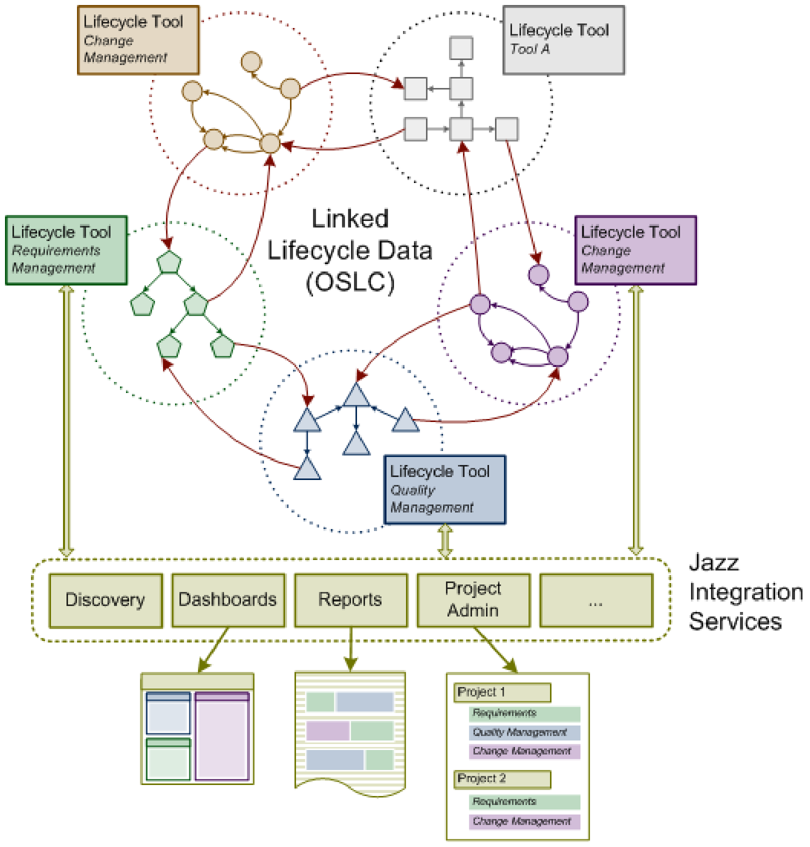

Figure 7.5: IBM Jazz Platform

The Jazz architecture uses two key concepts:

- Linked lifecycle data. This allows data from multiple lifecycle tools (such as requirements, change requests, test plans, codes) to be linked in some way.
- Integration services; providing cross-cutting capability that all lifecycle tools can use (e.g. user identify management, project admin, lifecycle query, dashboards); enabling a coherent set of tools that work well together.

The Open Services for Lifecycle Collaboration (OSLC) initiative uses the linked data approach. OSLC supports the notions of Providers and Consumers. An OSLC Provider is a tool that "owns" lifecycle data and exposes that data to other tools as described in its corresponding OSLC

[12] https://jazz.net

specification. An OSLC Consumer is a tool that accesses another tool's data through its corresponding OSLC-specified interface.

The OSLC community has published specifications for different lifecycles (such as requirements, change requests, test plans, codes), these identify a set of standard interfaces and methods. Using this data tools can establish links to data managed by other tools, possibly hosted on a different platform. The data is exposed using RESTful web services.

The linked data approach assumes that each tool is responsible for managing its data and only references to fractions of the tool internal data are given to other tools via the exposed interfaces. Since access to the data is via web services, this has some implications on the availability of data. To summarize, the IBM Jazz platform is a concrete implementation of the core OSLC principles, with data integration central to the tool integration approach; each tool retains ownership of its own data. A centralized and externalized data repository is not part of the architectural design. Only via specific interfaces as defined by the OSLC specifications are subsets of the tool data made public. Control integration is not provided as a core capability of the Jazz platform, but could be introduced with the integration of a specific tool (e.g. Rational Build Forge). IBM Jazz has a particular approach to presentation integration as it allows the rendering of data located in remote tools within the user interface of that tool. This works for specific subsets of data.

The usage of Jazz in custom-made development environments is currently not easy to achieve, only a limited number of tools implement one of the OSLC specifications. It is also worth noting that the OSLC specifications are still evolving. They are designed to be extendable so more tool data can be exposed if required.

7.3.3.3. Eclipse

Eclipse is a component based architecture, which is easily extensible through plugins and extension points. At its core, there is Equinox which is an implementation of the OSGi core framework specification. The Eclipse community, with the support of the Eclipse Foundation, provides a set of pre-packaged plugins which provide integrated development environments (IDEs), targeting different development or usage needs. Using the same Eclipse core, we can find IDEs covering the development needs of Java, C/C++, JavaEE, Scout, Domain-Specific Languages, Modelling, Rich Client Platforms, Remote Applications Platforms, Testing, Reporting, Parallel Applications or for Mobile Applications.

The Eclipse community is very active, and features and plugins for different purposes extend the Eclipse functionalities in very different ways. For example, the Eclipse marketplace https://marketplace.eclipse.org/ gathers more than 26 million solutions. Being able to easily integrate different features and tools in the same extensible platform makes Eclipse a widely adopted integration platform. The integration can be just the fact that the tools run in the same framework or more complex relations among tools could exist if they use the pre-defined extension points that the plugins expose.

Eclipse Lyo is a SDK to help the Eclipse community adopt OSLC and one example of the many plugins that are available for the Eclipse Platform. It is a model-based source code generator, which represents a tool-chain as a model. This way, the development can be abstracted from technical OSLC details. The generator produces code that bridges a source application and other

OSLC clients/servers. The OSLC part of the generated code is fully functional, only the application part has to be extended manually to fit the needs.

Lyo's tool-chain model is backed by the Eclipse Modeling Framework (EMF). The OSLC EMF model is split into three viewpoints: A *Domain Specification View* defines resource types with properties and relationships. The *Toolchain View* allocates a set of resources to a specific tool. Finally, internal details of the tool interface are modelled by the *Adapter Interface View*. Those three model views are used to generate code that is compliant to the OSLC4J SDK. Generated code is independent of the initial models and the Lyo platform. An incremental work process with multiple model refinements is supported.
The EMF base provides Model-Driven Software Development advantages such as validation or model-to-model transformation (e.g. to enhance or extend a source model). Lyo gains further benefits from the Eclipse Platform, for instance graphical editors to deal with OSLC models.

Code generated by Lyo acts as an adapter to a source application. So, modelled resources and services represent business logic and data of the server application, but to actually access internal application data a mapping between the adapter and the source application has to be implemented. For this purpose, the Lyo code generator creates required method stubs.

7.3.3.4. Bottom-Up Technologies for Reuse

Bottom-Up Technologies for Reuse [Martinez et al. 2015, Martinez et al. 2017] is a built-on Eclipse framework where different tools, techniques and algorithms can be integrated for relevant activities in SPL extraction through extension points. The adapters are also an important concept in BUT4Reuse enabling the support of different artefact types. The adapters extension points define how a given artefact type can be represented in an abstraction based on elements. This intermediate representation is used for the different analysis of the software variants towards the extraction of an SPL. More information can be found in Section 4.3.2.

7.3.3.5. KernelHaven

KernelHaven [KernelHaven 2017] is developed by the University of Hildesheim to simplify the development of research prototypes in the domain of analyses of software product lines by providing a reusable platform. Its plug-in architecture facilitates the integration of new analysis concepts and the novel combination of existing data extractors. KernelHaven provides capabilities to extract three different types of information from product line assets: variability models, build information, and code. This is illustrated in Figure 7.6. The data extractors are based on powerful research prototypes like Undertaker [Undertaker 2017], TypeChef [TypeChef 2017], KconfigReader [KconfigReader 2017], or KbuildMiner [KBuildMiner 2017] and can be flexibly exchanged and combined with each other. The correct use of data extractors is specified for the desired analysis in human readable form through a configuration file to improve the understandability of conducted experiments for external reviewers and to support reproducibility. While KernelHaven serves as an integration platform to simplify the development of analysis tools, it may itself be integrated into a bigger context like an integrated tool-chain.

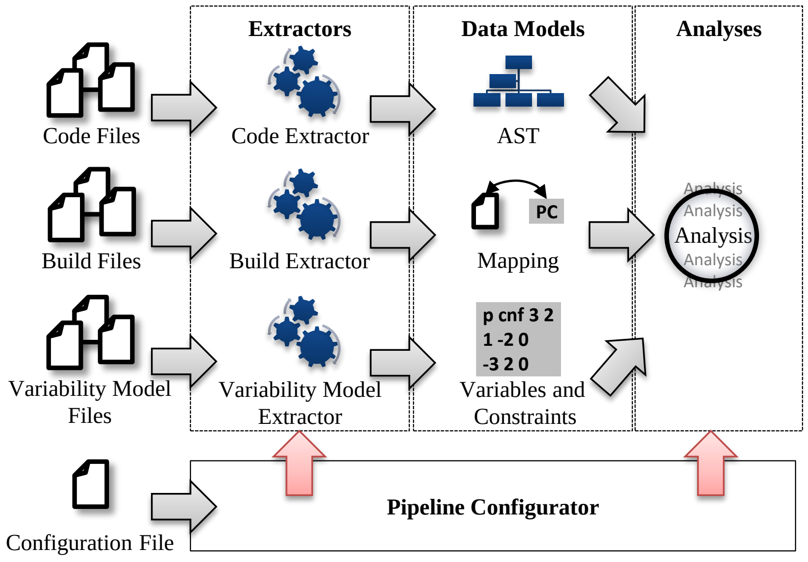

Figure 7.6: Architecture of KernelHaven

7.4. Open challenges in Tool Integration

7.4.1. Easy integration/exchange of algorithms, tools and services

Some tools and services provide similar information, but differ in the details and granularity of this information, like extracting code blocks from C preprocessor-annotated code [Undertaker 2017] against an entire abstract syntax tree including macro-expansion [TypeChef 2017]; tool integration needs to support the re-configuration of a tool-chain, e.g., to adapt to new requirements or to enhance existing tool support, in an easy-to-use manner.

The integration of a new algorithm or the exchange of an existing algorithm typically requires re-compilation of the affected tool or service; such integration or exchange needs to be supported in a way that avoids re-compilation but utilizes the re-configuration-mechanism of the tool-chain.

7.4.2. Development of compatible interfaces

Existing tools and services typically provide and require their individual format for data exchange, like IVML [IVML 2015] in EASy-Producer [EASy-Producer 2016] and VEL [VEL 2015] in pure::variants [PureVariants 2006] for variability models; a common interface for different input/output-formats needs to be available for different types of artefacts.

7.4.3. Development of common data models

Existing tools and services typically rely on their individual data model for storing and processing data; tool integration needs to offer a common data model, which, in particular, allows the exchange of information between different tools and services.

7.4.4. Integration needs to support the efficient computation of metrics

Software metrics are well established in software engineering to measure properties of software. In a study of variability-aware implementation metrics, we discovered that most suitable metrics are developed from scratch instead of adapting well established metrics for the needs of SPLs [El-Sharkawy et al. 2017b]. The state of the art in variability-aware implementation metrics miss to provide concepts and evaluations whether lifting classical software metrics may be beneficial to software product line engineering.

Software metrics that have been designed to analyse software product lines usually take only a certain aspect of variability information into account. For instance, some metrics operate on variability models only [Bagheri et al. 2011, Berger and Guo 2014, Maâzoun et al. 2016, Mann et al. 2011], while other metrics measure different aspects of variation points and ignore the rest of the code [Hunsen et al. 2016, Liebig et al. 2010, Zhang et al. 2012]. We did not find any complexity metric combining different information sources, like variability model, variation points, and code [El-Sharkawy et al. 2017b]. Such metrics need the combination of different information sources or parsers to gather the required information.

While the state of the art often makes use of metrics that have been designed to analyse software product lines, there exist only very few contributions that evaluate their ability to draw conclusions about qualitative aspects of the analysed artefacts. Positive examples are evaluated correlations between metrics for variability models and maintainability [Bagheri et al. 2011] and correlation between measures of configuration complexity in code artefacts and vulnerabilities [Ferreira et al. 2016]. However, due to insufficient amount of evaluations, it remains unclear to what extent metrics can contribute to a better understanding of product line artefacts in general.

7.4.5. **Evolution**

A particular focus of round-trip SPLE is on asset co-evolution automation technologies, which will consider different program versions as well as their differences for the automatic support of the synchronized realization of changes between different asset class pairs; in round-trip SPLE the resulting algorithms, tools, and services need to be seamlessly integrated in the tool-chain of the project to provide such automatic support, like automated integration of deviating product and the derivation of SPLs.

As part of the development of asset verification automation technologies, incremental analysis algorithms, tools, and services need to be seamlessly integrated in the tool-chain of round-trip SPLE to provide fast turnaround time in agile settings for asset verification.

8. Projects related to REVaMP2

8.1. ESAPS / Café / Families

From 1999 to 2005 there were three consecutive research projects on SPLE[13]. Each project represented an effort from 286 to 330 person-years and they involved from 6 to 8 countries and from 20 to 22 partners. These projects created the comprehensive foundations of product line engineering as it is typically depicted today, and the series of Product-Families-Engineering Workshops which later evolved into the International Software Product Line Conference (SPLC). The research and industrial transfer results of the projects have been documented in a series of books [Pohl et al. 2005, Käkölä and Dueñas 2006, Van der Linden et al. 2007]. These projects were not formed in a vacuum, but built on the predecessor projects ARES [Jazayeri et al. 2000] and PRAISE and a range of previous work in companies like Philips, Siemens, Nokia, Thales, etc. as well as work from the US. However, they were the first projects of such significant size on product line engineering and enlarged the focus from pure technical topics like architectures to encompass all facets of the BAPO model (Business, Architecture, Process, Organization) [Jazayeri et al. 2000].

Within this series of projects, ESAPS had the task of laying a unified foundation on product line engineering research, bringing together many of the initial industrial attempts on this field. It established basic terminology (e.g., initially starting with the product family term, later switching to the term of product line), and basic methods for many problems in product line research such as scoping, domain analysis, architecture design, implementation methods, testing, etc.

The Café-project built on these foundations and enriched the set of available methods and tools for supporting product line engineering. It also had a stronger focus on covering business and organization aspects than ESAPS had. Also, while ESAPS particularly focused on the domain engineering perspective, the Café-project increasingly added the application engineering perspective.

Finally, the Families project focused on more comprehensively addressing all four BAPO dimensions and created outreach deliverables like [Pohl et al. 2005, Käkölä and Dueñas 2006, Van der Linden et al. 2007] and extensions of the CMMI for product lines (to address the process dimension). Overall the focus was on disseminating the product line approach to a wider audience.

Supporting product line evolution was among the topics addressed in these projects right from the start of ESAPS [Van der Linden 2002]. However, the main work was on establishing the foundations for supporting SPL adoption, less on evolution. Also, the evolution-oriented topics and approaches researched in these projects were mostly different from the focus of REVaMP^2 as the research focussed mainly on methodological research and less on tooling. For example, within this project different practical approaches (and problems) of product line evolution were systematically analysed [Riva and del Rosso 2003], particularly also focussing on architecture-level evolution problems.

[13] https://itea3.org/project/esaps.html
https://itea3.org/project/cafe.html
https://itea3.org/project/families.html

Another avenue of research on SPL evolution that was addressed in these projects was supporting the traceability in SPL development. Again, this was mainly done from a methodological perspective, less from a tooling and implementation perspective. Even in the cases where reengineering was addressed, the focus was mostly on organizational aspects rather than supporting the human developer through the automation of tasks.

8.2. MoSiS

Model-driven development of highly configurable embedded Software-intensive Systems (MoSiS) is an ITEA 2 project 2007 – 2010 involving 12 partners from 5 countries[14]. The MoSiS project developed the Common Variability Language (CVL) that was the base for an ongoing standardization in the Object Management Group (OMG). This project is relevant as it prepared a variability-specific tool integration standard which we presented in Section X.

8.3. VARIES (VARiability In safety-critical Embedded Systems)

VARIES was an industry driven three-year research project (2012-2015) in the ARTEMIS programme on the topic of variability in safety critical embedded systems[15]. 23 Organisations from 7 countries were involved. VARIES recognizes that embedded systems are rarely conceived from scratch. It often is an evolving process in which variants are added and removed when variability drivers trigger new variability decisions. VARIES address the following variability paradox: how can we ensure that the benefits of introducing more variability in the product portfolio outweighs the cost of more complexity caused by variability? The costs are not only engineering costs but also long-term maintenance costs.

The VARIES project had three key objectives: 1) Enable companies to make an informed decision on variability use in safety-critical embedded systems. 2) Provide effective variability architectures and approaches for safety-critical embedded systems, and 3) Offer consistent, integrated and continuous variability management over the entire product life cycle. In this context, there were two major lines of investigation, converging into a single platform: Methodologies to handle variability in the product portfolio based on variability drivers and instruments that enable companies to manage their variability challenges.

VARIES did an extensive review of the state of the art in methods for capturing and representing variability data and developed new methodologies based on workshops and questionnaires.

Methods derived from theory:

- The Complexity Cube
- Typologies of variability and variability drivers
- Business Strategies
- Domain Framework
- Process based methods (Stage Gate, V-Model, Engineering standards) and Supply Chain Management

[14] https://itea3.org/project/mosis.html
[15] http://www.varies.eu

Empirical methods:

- VARIES Software Product Line Framework
- The Sirris Framework for Identifying Variability Drivers in a Company
- Variability Driver Identification Workshop, an empirical method created in the context of the VARIES project by Sirris, VTT, HiQ and Vlerick
- Vlerick – Barco Questionnaire for identifying the impact of variability drivers on stakeholders

In-depth analysis and synthesis methods:

- Feature Value Analysis
- Cost Effect Analysis
- VariVal (developed by Sirris)
- Product and Process information stored in databases (Product Live Cycle Management, Product Data Management, Enterprise Resource Planning)
- Scope, Commonality and Variability (SCV) analysis

The evolution of the variability inside an SPL can be described by variability drivers that enforce variability decisions. A taxonomy of variability drivers was made. The main categories are Compliance, Technology and Market. 22 Different variability drivers were identified and documented. For each of the drivers, it was described the typical challenges, stakeholders (initiating and affected), impact on the company, influence and management (can it be ignored or changed), effect on variability and the relations with other drivers.

One of the contributions of the project was the “variability driver genome” to allow quick comparison between product lines / portfolios and even across companies. This genome is a matrix representing the incidence of variability drivers over time (rows represent a time period and columns the incidence of a variability driver in that period). The impact of the product portfolio and the architecture behind it has an important effect on competitiveness and efficiency. There are trade-offs between the complexity costs and market opportunities of more or less product variety. It is observed that variability decisions are often taken in an informal decision process and that decision evolves over time. Therefore, it is important to understand the variability decision process well. Figure 8.1 illustrates the workflow of variability drivers.

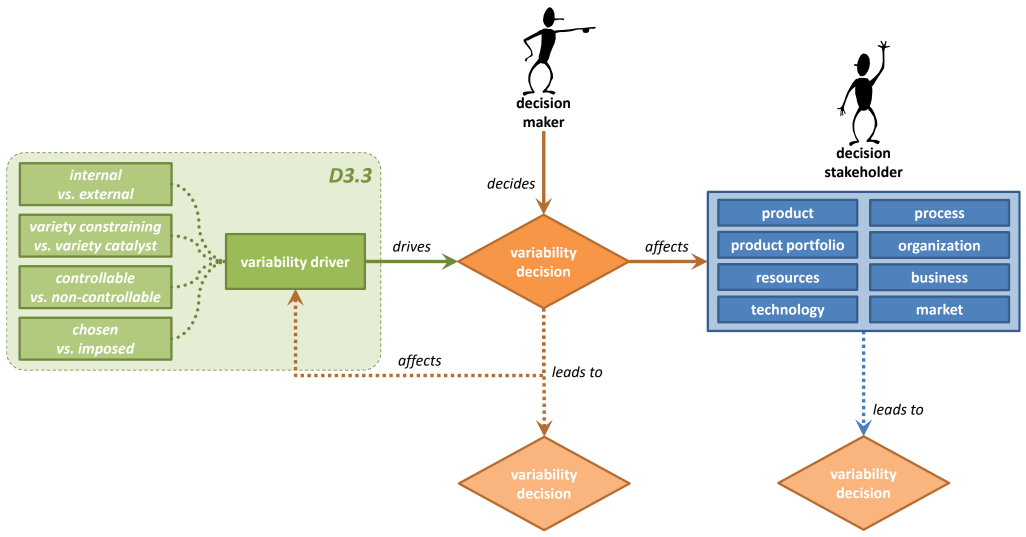

Figure 8.1 Workflow for variability drivers

A typology of variability decisions was made based on the following criteria: internal/external product variety, consequences of the decision on the portfolio, visibility, disciplines and affected stakeholders, impact on resources and profitability. The interactions between variability decisions are described. They can be used as a tool to detect possible forgotten or missing drivers that affect the decisions. Figure 8.2 illustrates the defined stepwise model to coordinate the planning of product portfolio and roadmap.

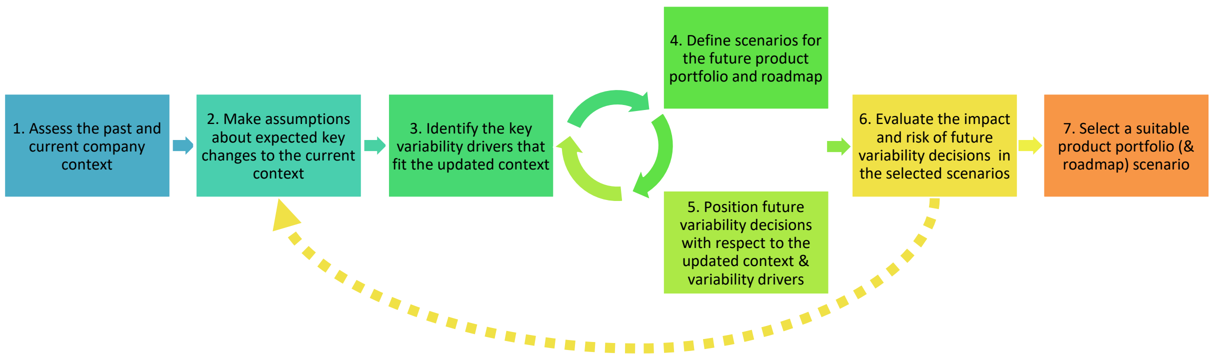
1. Assess the past and current company context
2. Make assumptions about expected key changes to the current context
3. Identify the key variability drivers that fit the updated context
4. Define scenarios for the future product portfolio and roadmap
5. Position future variability decisions with respect to the updated context & variability drivers
6. Evaluate the impact and risk of future variability decisions in the selected scenarios
7. Select a suitable product portfolio (& roadmap) scenario

Figure 8.2: A Model to Coordinate the Planning of Product Portfolio and Roadmap

Finally VARIES methodologies added the concept of the life cycle. Some variability drivers occur more often in certain parts of the life cycle. The concept is added to the variability genome. VARIES defined a glossary with terminology from different perspectives: Business, industrial engineering, Management and Software Engineering.

We discuss below several topics faced in the VARIES project which are of special interest for round-trip SPLE.

Variability Languages: VARIES first choice for a standardized variability language was CVL. OMG did not continue to finalise the standardisation process after a patent threat. Since it became unlikely that CVL would become a standard in the course of VARIES an alternative was developed by SINTEF called BVR (Base Variability Resolution) [Vasilevskiy et al. 2015]. An open source Eclipse based tool supporting BVR has been published. BVR has a smaller and more concise meta-model that fits the needs of the VARIES industrial partners. A concept that cannot be found in CVL and BVR is that of a fixed or variable number of slots that can be occupied by modules that occupy one or more slots.

Product Decision Methods: TECNALIA presents a cascade of Systematic Reuse Strategies. Reuse-Invest helps to decide if it is economically profitable to adopt systematic reuse. Reuse-Check assesses if the organization processes are ready for adopting systematic reuse practices. The next step is the Reuse-Process Assessment Method to see what processes need to be introduced. Finally, a Reuse-Action Plan is created.

VTTs Informal Decision Support Method: Changes in product development often include many risks and difficulties in managing the changes and consequences of significant change decisions. Capturing and analysing significant decisions on product development changes helps manage changes and the architecture of the system-of-interest. Many organisations follow an incremental and iterative development process in product development. Therefore, an incremental and iterative model is needed for managing significant change decisions. The purpose of the change decision support method is to find the risks and managerial and technical consequences of significant change decisions and to improve the transparency of the decisions.

Variability Analysis Challenges: The subjects of the challenges are:

- Testing and quality assurance
- Consistency between problem space and solution space
- Automatic support for commonality and variability analysis with a challenge on Re-Engineering of software product lines:
  - Investigate automatic mechanisms that allow extraction of variability from legacy systems. Different mechanisms are used in this direction, such as clone detection and static code analysis techniques. There is a need for the extraction of knowledge from code/models to aid in the identification of variability, as well as the impact of the complexity of existing software in the Re-engineering of the new product line. The former usually relies on clone detection techniques.
  - Often, software product lines are realized using the branching and merging facilities of version control systems. In such a scenario, VARIES aims to investigate analyses for migration towards explicit variability models, in order to use them pro-actively in version control.
- Impact Analysis

For each of the above challenges, the VARIES project investigated the state-of-the-art and they identified several gaps such as the importance of generation of test cases and test suite optimization, the limitations in handling multiple variants, or (more at technical level) the possibility to use OCL (Object Constraint Language) as entities in CVL and SAT-solvers. Specific in the field of code extraction, the detection of implicit code constraints among features was identified as a problem that remains to be solved.

Variability Analysis Solutions: Several solutions were proposed dealing with different topics that we detail below.
A Lightweight Formal Technique for Qualifying Product Derivation Tools: Variability analysis [VARIES D4.7 2012] addresses the challenge of developing a formally certifiable implementation of variability mechanisms based on BVR and OVM variability modeling languages. A formally defined core language for variability modeling Featherweight VML (FwVML) was developed in the analysis to prove the correctness of product derivation tools for functional safety-critical systems using translation validation methodology.

Consistency Between Problem and Solution Space: The project presents an approach to automatically and accurately extract constraints from existing implementations using static analysis techniques [VARIES D4.7 2012]. Different sources of configuration constraints, including build-time errors and new feature-effect heuristics, are used to automatically extract configuration constraints from C code. This analysis also supports systematic methods of re-engineering product portfolios into product lines and makes it easier to certify the product line when individual products were certified before re-engineering. Proposed are novel scalable extraction strategies based on the structural use of #IFDEF directives, on parser and type errors, and on linker checks.

Clone Detection for Reengineering of Software Product Lines: This analysis developed a prototype tool that can extract the variability of two legacy systems, and create a common code base by introducing variation points in the merged code using #ifdef directives [VARIES D4.7

2012]. It explored the use of clone detection techniques that work on ASTs (Abstract Syntax Tree). This analysis offers important tool support for the methodology of re-engineering legacy systems into a product line.

A Methodology for Migrating a Set of Clones to a Safety Critical Product Line: This analysis explores a systematic approach of migrating a set of clones to a safety critical product line, considering all the necessary steps that are needed to successfully perform the migration [VARIES D4.7 2012]. A special focus is given on verifying and certifying the resulting product line. It also proposes a novel approach on how to manage a set of variants that are developed using clone-and-own in a structured way that allows at a later timer to migrate more efficiently to a product line reducing overall efforts and costs.

Safety-criticality: Safety-critical is one of VARIES focus domains on the variability problems. This was handled in VARIES D4.8 “Challenges and Potentials of Certifying Product Lines” and D4.9 “Approving Product Lines”. The requirements of the functional safety as well as the certification standards and their consequent restriction of modifications are in competition with the efficient handling of variability in product lines.
There are some strategies that can be adopted to facilitate it:

- Reassessment of the modification only. The usage of an explicit and separate variability model to generate all variants ensures that the description of the modifications is complete, isolated and explicit.
- Concept of elements, subsystems and systems – compliant items: when the subsystems (also called Safety Element out of Context) are already certified there is a reduction in time to validate a new system that is composed of those elements. This can also be used to limit the reassessment of a new variant.
- Process certification: in functional safety both the product and the development process are evaluated. If the process is well chosen and described it must not be reassessed for each variant in a product line.

Requirements for the reuse of easements to facilitate approval of product variants and SPLs:

- Early identification of the relevant standards and determination of the proposed application of the product.
- Usage of already approved subsystems.
- Separation of safety and non-safety functions.
- Separation of variable and common part in different models can result in an easier safety assessment.
- Definition of a common set of safety requirements reduces the effort to assess each variant.
- Make the product configurable. A single certification can than cover all variants.
- Multiple certification of variants is often needed as variability and product lines are not well covered in safety standards. A proper design and development process can reduce the effort needed.

The results are the VARIES tools and platform. References can be found on the VARIES website to Plum (Tecnalia), the BVR Tool (SINTEF), pure::variants from pure-systems, Artisan Studio and the Atego Asset Library from Atego (now PTC), CTE XL and Meran from Berner and Mattner, Modelbus and Traceino form Fraunhofer. Public deliverables and extracts from non-public deliverables are made available on the VARIES website http://www.varies.eu/download-media or http://www.varies.eu/results.

8.4. EvoLine

The EvoLine-project [EvoLine 2017] was a bilateral research project between the University of Bremen and the University of Hildesheim, founded by DFG (German Research Foundation) as part of the Priority Programme SPP1593: Design For Future – Managed Software Evolution. The SPP focused on research on the evolution of software. Within the SPP the EvoLine-project focused on the development of methods and tools to support the correct and consistent evolution of SPLs written in C. A particular focus was given to embedded systems and industrial automation systems. The work at the University of Hildesheim in the REVaMP² project can be partially seen as a continuation of much of the research done in EvoLine.

In the embedded domain, variability is typically realized using static pre-processor directives or explicit configuration variables. This complicates analysis and maintenance of programs significantly, as it is not sufficient to perform correct evolution of a single program, but of all possible programs. The relation between configuration options (the so-called variability model) and the program code must be considered. Thus, the continuous evolution of long-living product lines requires to make consistent modification to the code and the variability model to avoid erroneous configurations or non-configurable products. The EvoLine project did research on family-based approaches [Thüm et al. 2012] to identify errors caused by inconsistent evolution.

A central hypothesis of the EvoLine project is that an evolutionary analysis approach will lead to significantly more efficient decision procedures to identify any potential evolution issues in a software product line [Schmid et al. 2015]. The core idea of this approach is to focus on the changes, e.g., introduced by a commit of a software product line repository, to deduce that the evolved product line is consistent and correct if it was before. This idea is in line with the evolutionary (incremental) analysis we aim for in WP6. To provide such an approach, we need to understand the changes typically made to a software product line (e.g. the number of changes to a product line in a certain timeframe, the type and number of affected artefacts, the intensity of changes to variability information in affected artefacts, etc.). Started in the EvoLine project and already extended as part of REVaMP², a study on the evolution of open-source SPLs was conducted [Kroeher et al. 2017], contributing to the understanding of SPL evolution at large and, thus, the REVaMP² project.

Lüdemann et al. studied how variability information can be used to reduce complexity in formal concept analysis [Lüdemann et al. 2016]. Formal concept analysis aims at deriving a lattice that represents the variability of a source file, as a support in reengineering software product lines. However, the lattices constructed by this approach can become very large and complex. In EvoLine, Lüdemann et al. showed how consistency analysis, both within code variability and between code and a variability model, can be used to reduce the complexity of a lattice.

Due to the lack of publicly available software product lines, Linux is often used as a case study in research. Studies have shown that its implementation is comparable with code from industry [Hunsen et al. 2016] and that its variability model written in Kconfig is fundamentally different to academic models, which were used before as case studies to evaluate scientific results [Berger et al. 2013, She et al. 2010]. The imprecise documentation of the variability management concepts and many corner cases complicate proper variability extraction and analysis. In EvoLine, an analysis of the Kconfig semantics was applied to uncover the real semantics of Kconfig and to investigate the accuracy of existing variability extractors [El-Sharkawy et al. 2015a, El-Sharkawy et al. 2015b].

While there already exists much work that defines family-based concepts to detect syntactical defects in product line code, like variability-aware type checking, and control-flow analysis [Kästner et al. 2012], or dead code analysis [Tartler et al. 2011], there exist only little work to provide a semantic analysis. In Evoline, El-Sharkawy et al. provided an approach to analyse the consistency between modelled and coded feature dependencies [El-Sharkawy et al. 2016]. Divergence of them was named mismatched configuration information. As part of the REVaMP²-project, this work might be extended based on the feature effect synthesis [Nadi et al. 2015] to provide a more generalized solution. An empirical study on Linux did show the importance of this analysis as two-thirds of the identified configuration mismatches are related to the inadequate use of the variability management concepts provided and may lead to semantically incorrect products during product derivation [El-Sharkawy et al. 2017].

8.5. AMALTHEA

The ITEA Project 09013 AMALTHEA (2011-2014) is an open source tool platform for engineering embedded multi- and many-core software systems[16]. The platform enables the creation and management of complex tool chains including simulation and validation. As an open platform, proven in the automotive sector by Bosch and their partners, it supports interoperability and extensibility and unifies data exchange in cross-organizational projects. AMALTHEA allows tool vendors, engineering companies and other suppliers in the toolchain to efficiently integrate their products and expertise.

The Eclipse APP4MC platform enables the creation and management of complex tool chains including simulation and validation. The main advantage of AMALTHEA for shared software development is its independence from software architecture and development methods. The hardware model in AMALTHEA provides the options for describing the hardware features in detail including the hardware access latencies, core features and memory. The dependency of a software function on a hardware feature can be easily modelled. Usage of AMALTHEA as a standard exchange format also helps in reducing the integration time. Amalthea model can be shared much before the integration phase, while the actual software is still being developed and validated.

There were some investigations concerning variant handling within the AMALTHEA project. But currently AMALTHEA model is not supporting variant handling and there are no plans to add variation points to the model to avoid complexity. An AMALTHEA model is mainly to describe the dynamic behaviour of an already configured software version. So extracting and describing software variance is not currently supported by the AMALTHEA model.

8.6. CRYSTAL

CRYSTAL (CRitical sYSTem engineering AcceLeration) is an Artemis Joint undertaking project dealing with interoperability specification and reference technology platforms for safety-critical systems[17]. The relation of CRYSTAL to this document is the advances in the verification of

[16] http://www.amalthea-project.org/
[17] http://www.crystal-artemis.eu/

requirements properties (related to Section 6.3), requirements reuse, and their usage of tool integration approaches (Section 7) such as OSLC.

8.7. iFEST

iFEST is an Artemis project about an industrial Framework for Embedded Systems Tools[18]. The project ran from 2010 to 2013 and its main achievements are the specification of an integration framework that extends OSLC and a Model Federation mechanism that can attach different roles to the same engineering artefact. Therefore, this project is related with Section 7 of this state-of-the-art document.

iFEST tested its approach in five industrial use cases with an emphasis on the metrics to analyse the benefits. By the use of integration technologies, a reduction of 20% of the development cost, time to market and the cost of poor quality could be shown. iFEST starts from the observation that engineers are confronted with a lot of unstructured information that is scattered among a lot of tools during the lifecycle of an embedded system. The life cycle can span several decades which increases the concern on vendor locking that in turn discourages tool adoption and usage.

iFEST defines the concepts of “Tool Chain” and “Integration Platform”. The difference are the stakeholders: 1) the industrial organisations that build the embedded systems and 2) the platform vendors. Both have an interest in the combined usage of the tools.

The iFEST Integration Framework consists of:

- The Technological Space on which the tool integration implementation relies (execution support, communication protocols, data exchange formats, …)
- The Adaptor Specifications that describe the data manipulated by and services specified by the tools of a Tool Chain. The specification relies on OSLC.
- Guide lines supporting the writing of specifications and implementing Tool Adaptors
- Technical and non-technical Principles to which specifications and implementations must adhere.

iFEST tried to define a common metamodel that merged all the concepts defined in the integrated tools and use it is as a common format for modes in different languages. It was shown that this can be used as a pivot model that reduces the number of point to point connections among tools. It also reduces the complexity of the transformations through an abstract vocabulary. The model is however too abstract to support all concepts of all tools, the metamodel would become too abstract. The conclusion of iFEST on this topic is that a single metamodel is not manageable.

To solve this issue iFEST proposes an approach based on Model Federation. The concept of Roles is introduced. Artefacts receive a Role based on each tool that manages them and the domain in which it is used. This allows an artefact to have different semantics depending on the context (tool) in which it is used. Depending on the tools, artefacts can refer to multiple “real elements”.

[18] http://www.artemis-ifest.eu/

8.8. EMC2

EMC2 – 'Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments' addresses the Artemis Innovation Pilot Programme: AIPP5: Computing platforms for embedded systems[19]. The objective of EMC2 is to establish Multi-Core technology in all relevant Embedded Systems domains, to enable product innovations and to accelerate their market penetration.

EMC2 is an attempt to summarize and to further develop the results from more than 20 ARTEMIS-research projects in the fields of Safety Critical Systems and Embedded Multi-Core Technology with the involvement of partners of these projects in order to make progress towards market innovation. The project is not monolithic in itself; it also combines 12 networked subprojects.

The following challenges were addressed and solutions developed to overcome:

- Dynamic Adaptability in Open Systems
    - Variable number of control units
    - It is possible to have unknown applications
    - It is possible a full range of dynamic changes
- Utilization of expensive system features only as Service-on-Demand in order to reduce the overall system cost.
- Handling of mixed criticality applications under real-time conditions
- Scalability and utmost flexibility
- Full scale deployment and management of integrated tool chains, through the entire lifecycle

As part of the results, an open tool-chain for seamless tool integration was demonstrated in an industrial environment as a prototype. The benefits of using the open tool-chain were:

- Higher engineering efficiency
- Reduced design time
- Increased quality
- Reduced time to look up information
- Reduced tool switch time

It also provides support for longer time goals such as:

- Extending the life-cycle of system engineering tool chains
- Easy replacement of tools
- Avoidance of vendor "lock-in" situations

The features by which the above results are reached can be identified as:

- Tool transparency
- Traceability of artefacts

By providing a transparent use of the necessary tools (an orchestrator, notification services, traceability services, and version control) based on OSLC and delegated interfaces, the developer is mostly contained in one tool context. In addition, the approach eased the creation of traceability links, provided means to relate to different versions and offered the basis for increased quality designs. A method and a tool for supporting application deployment on either software or hardware technologies was also developed.

[19] https://www.artemis-emc2.eu/

8.9. EternalS and HATS

EternalS, (Trustworthy Eternal Systems via Evolving Software, Data and Knowledge) was a FP7 Coordination and Support Action for three projects LivingKnowledge, HATS, Connect and SecureChange, that ran from 2010 till 2013 [20]. HATS was especially relevant to SPL topics. HATS: (Highly Adaptable and Trustworthy Software using Formal Models) formalises Software Product Line Engineering with a formal language called ABS (Abstract Behavioral Specification) to describe features, components and their instances. HATS handles variability (anticipated) and evolvability (not anticipated).

8.10. INDENICA

INDENICA, Engineering Virtual Domain-Specific Service Platforms was a 36 months SP7 project which started in October 2010 [21]. They worked on topics relevant to REVaMP^2 from which we will highlight the proposal of variability modelling languages and variability implementation languages.

[20] http://www.hats-project.eu
[21] https://sse.uni-hildesheim.de/en/research/projects/indenica/

9. Conclusion

The widespread adoption of Software-Intensive Systems and Services in industry has transformed the engineering process into a more agile, round-trip process. This allows for better use of legacy assets by applying systematic variability management.

In this document, we presented the state-of-the-art on the different aspects of this engineering process that is at the heart of the REVaMP² project, attempting to leverage the solutions that already exist in the literature to improve the competitiveness of software-intensive companies. In this document, we covered topics ranging from variability management and methodologies for round-trip engineering including product line extraction, product line verification, and product line co-evolution. In addition, we identified several tool integration methods that can help achieving a holistic solution to these interconnected challenges. We mentioned other projects that have dealt with similar problems to REVaMP², and recognized their contributions to the state-of-the-art.

In Section 2, we introduced relevant concepts and background information about the state-of-the-art on variability management and product line engineering. Software product line engineering aims at reusing software in a systematic way. This improves the process of software development by managing the variability of software products. In Section 3, we presented Round-trip engineering as a holistic approach to deal with variability management. Round-trip engineering uses synchronized modelling as a way to engineer the development process. This is accomplished by the use of reverse engineering, code-model-synchronization, and code generation. In Section 4, we discussed the state-of-the-art on product line extraction. Extraction helps to turn the legacy assets into reusable artefacts. In Section 5, we delved into the verification process. Formal verification of software-intensive systems ensures the quality of the final software products, and eases the often required product certification. In Section 6 we gave an overview of the state-of-the-art related to co-evolution. Co-evolution has been discussed in several contexts with one common trait: it is performed on two co-evolving streams of data artefacts. This helps in reducing the parallel effort needed to maintain co-evolving artefacts. In Section 7, we identified the solutions and the challenges of tool integration. This helps us in developing a common integrated framework that can be used in the future to create a coherent technical solution to the challenges discussed in previous sections. Finally, in Section 8 we presented the related previous projects and their contributions to the state-of-the-art. For each of the discussed topics, we identified the open challenges that the community faces. By capitalizing the collaboration of academic and industrial partners, we attempt to solve some of these challenges in the context of REVaMP² project.

References

[Abal et al. 2014]	Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, New York, NY, USA, 421-432. DOI=http://dx.doi.org/10.1145/2642937.2642990
[Abílio et al. 2015]	Ramon Abílio, Juliana Padilha, Eduardo Figueiredo, and Heitor Costa. Detecting code smells in software product lines – an exploratory study. In Proceedings of the 2015 12th International Conference on Information Technology - New Generations, ITNG '15, pages 433–438. IEEE Computer Society, 2015.
[Abílio et al. 2016]	Ramon Abilio, Gustavo Vale, Eduardo Figueiredo, and Heitor Costa. Metrics for feature-oriented programming. In Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics, WETSoM '16, pages 36–42. ACM, 2016.
[Al-Hajjaji et al. 2016]	Mustafa Al-Hajjaji, Fabian Benduhn, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Mutation Operators for Preprocessor-Based Variability. In Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '16), Ina Schaefer, Vander Alves, and Eduardo Santana de Almeida (Eds.). ACM, New York, NY, USA, 81-88. DOI=http://dx.doi.org/10.1145/2866614.2866626
[Ali et al. 2009]	M.S. Ali, M.A. Babar, and K. Schmid. A comparative survey of economic models for software product lines. In 35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009, pages 275–278. IEEE Computer Society.
[Al-Kofahi et al. 2015]	Jafar Al-Kofahi, Lisong Guo, Hung Viet Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. Static detection of configuration-dependent bugs in configurable software. In Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15), Vol. 2. IEEE Press, NJ, USA, 795-796.
[Alliance OSGi 2009]	Alliance, OSGi. "OSGi Service Platform–Service Compendium (Release 4, Version 4.1)." Createspace, August (2009).
[Alvarez-Rodríguez et al. 2012]	Alvarez-Rodríguez, Jose María, José Emilio Labra Gayo, Francisco Adolfo Cifuentes Silva, Giner Alor-Hernández, Cuauhtémoc Sánchez, and Jaime Alberto Guzmán Luna. 2012. "Towards a Pan-European E-Procurement Platform to Aggregate, Publish and Search Public Procurement Notices Powered by Linked Open Data: The Moldeas Approach." International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 22 (3): 365–84.
[Alvarez-Rodríguez et al. 2013]	Alvarez-Rodríguez, Jose María, José Emilio Labra-Gayo, and Patricia Ordoñez de Pablos. 2013. "Leveraging Semantics to Represent and Compute Quantitative Indexes: The RDFIndex Approach." In Metadata and Semantics Research, edited by Emmanouel Garoufallou and Jane Greenberg, 390:175–87. Communications in Computer and Information Science. Springer International Publishing. http://dx.doi.org/10.1007/978-3-319-03437-9_19.
[Alvarez-Rodríguez et al. 2015]	Alvarez-Rodríguez, Jose Maria, Juan Llorens, Manuela Alejandres, and Jose Fuentes. 2015. "OSLC-KM: A Knowledge Management Specification for OSLC-Based Resources." INCOSE International Symposium 25 (1): 16–34. doi:10.1002/j.2334-5837.2015.00046.x.
[Andrade et al. 2013]	Rodrigo Andrade, Henrique Rebêlo, Márcio Ribeiro, and Paulo Borba. Aspectj-based idioms for flexible feature binding. In Proceedings of the 2013 VII Brazilian Symposium on Software Components, Architectures and Reuse, SBCARS '13, pages 59–68. IEEE Computer Society, 2013.
[Andreessen 2011]	Andreessen, Marc (20 August 2011). "Why Software is Eating the World". The Wall Street Journal (Life & Culture). Retrieved 7. February 2016, www.wsj.com/articles/SB10001424053111903480904576512250915629460
[Apel et al. 2011]	Sven Apel and Dirk Beyer. Feature cohesion in software product lines: An exploratory study. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 421–430. ACM, 2011.
[Apel et al. 2011b]	Apel, S., Speidel, H., Wendler, P., Rhein, A. v., & Beyer, D. (2011). Detection of Feature Interactions Using Feature-aware Verification. (ss. 372-375). IEEE Computer Society.
[Apel et al. 2013]	Apel, S., Kästner, C., & Lengauer, C. (2013). Language-Independent and Automated Software Composition: The FeatureHouse Experience. IEEE Transactions on Software Engineering, 39(1), 63-79.
[Apel et al. 2016]	Apel, S., Batory, D., Kästner, C., & Saake, G. (2016). Feature-Oriented Software Product Lines. Springer-Verlag Berlin An.

[Asikainen et al. 2003] Asikainen, T., Soininen, T., & Männistö, T. (2003, November). A Koala-based approach for modelling and deploying configurable software product families. In International Workshop on Software Product-Family Engineering (pp. 225-249). Springer, Berlin, Heidelberg.

[Asirelli et al. 2010] Asirelli, P., Ter, M. H., Fantechi, A., & Gnesi, S. (2010). A Logical Framework to Deal with Variability. (ss. 43-58). Springer-Verlag.

[Assunção et al. 2017] Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A. Reengineering legacy applications into software product lines: a systematic mapping. Empirical Software Engineering, 2017

[Atkinson et al. 2002] Atkinson, C. (2002). Component-based product line engineering with UML. Pearson Education.

[Baclawski et al. 2002] Baclawski, Kenneth, Mieczyslaw M. Kokar, Richard J. Waldinger, and Paul A. Kogut. 2002. “Consistency Checking of Semantic Web Ontologies.” In International Semantic Web Conference, 454–59.

[Baclawski et al. 2002] Baclawski, Kenneth, et al. "Extending the Unified Modeling Language for ontology development." Software and Systems Modeling 1.2 (2002): 142-156.

[Bagheri et al. 2011] Ebrahim Bagheri and Dragan Gasevic. Assessing the maintainability of software product line feature models using structural metrics. Software Quality Journal, 19:579–612, Sep 2011.

[Baker et al. 2013] Baker, Thomas, Sean Bechhofer, Antoine Isaac, Alistair Miles, Guus Schreiber, and Ed Summers. 2013. “Key Choices in the Design of Simple Knowledge Organization System (SKOS).” Web Semantics: Science, Services and Agents on the World Wide Web 20 (May): 35–49. doi:10.1016/j.websem.2013.05.001.

[Bashroush et al. 2017] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher and Goetz Botterweck: CASE Tool Support for Variability Management in Software Product Lines, CSUR 2017

[Batista et al. 2008] Batista, T. V., Bastarrica, M. C., Soares, S., & da Silva, L. F. (2008, September). A Marriage of MDD and Early Aspects in Software Product Line Development. In SPLC (2) (pp. 97-103).

[Batory 2005] Don S. Batory: Feature Models, Grammars, and Propositional Formulas. SPLC 2005: 7-20

[Batory et al. 2003] Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. In: Clarke, L.A., Dillon, L., Tichy, W.F. (eds.) Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA. pp. 187–197. IEEE Computer Society (2003)

[Batory et al. 2004] Batory, D. (2004, May). Feature-oriented programming and the AHEAD tool suite. In Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on (pp. 702-703). IEEE.

[Becker et al. 2003] Becker, M. (2003, February). Towards a general model of variability in product families. In Workshop on Software Variability Management. http://www.cs.rug.nl/Research/SE/svm/proceedingsSVM2003Groningen.pdf (pp. 19-27).

[Beckert et al. 2007] Beckert, B., Hahnle, R., & Schmitt, P. H. (2007). Verification of Object-oriented Software: The KeY Approach. Springer-Verlag.

[Beek et al. 2012] Beek, M. H., Mazzanti, F., & Sulova, A. (2012). VMC: A Tool for Product Variability Analysis. i D. Giannakopoulou, & D. Mery (Red.), FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings (ss. 450-454). Springer Berlin Heidelberg.

[Beek et al. 2016] Beek, M. H., Vink, E. P., & Willemse, T. A. (2016). Towards a feature mu-calculus targeting SPL verification. arXiv preprint arXiv:1604.00350.

[Beek et al. 2017] Beek, M. H., Vink, E. P., & Willemse, T. A. (2017). Family-Based Model Checking with mCRL2. (ss. 387-405). Springer-Verlag New York, Inc.

[Benavides et al. 2005] David Benavides, Pablo Trinidad Martín-Arroyo, Antonio Ruiz Cortés (2005) Automated Reasoning on Feature Models. CAiSE 2005: 491-503

[Benjamins et al. 1998] Benjamins, V. Richard, Dieter Fensel, and Asunción Gómez-Pérez. 1998. “Knowledge Management through Ontologies.” In PAKM.

[Berger and Guo 2014] Berger T. and Jianmei Guo. Towards system analysis with variability model metrics. International Workshop on Variability Modelling of Software Intensive Systems, VaMoS ’14, pages 23:1–23:8. ACM, 2014.

[Berger and She 2010] Berger, T., & She, S. (2010). Formal semantics of the CDL language. Technical note, University of Leipzig.

[Berger et al. 2010] Berger, T., S. She, R. Lotufo, K. Czarnecki, and A. Wasowski. 2010. Feature-to-code mapping in two large product lines. In Proceedings of the 14th International Conference on Software Product Lines, pages 498–499, 2010.

[Berger et al. 2010b]	Berger, T., She, S., Lotufo, R., Wąsowski, A., & Czarnecki, K. (2010, September). Variability modeling in the real: a perspective from the operating systems domain. In Proceedings of the IEEE/ACM international conference on Automated software engineering (pp. 73-82). ACM.
[Berger et al. 2013]	Berger, T., Rublack, R., Nair, D., Atlee, J. M., Becker, M., Czarnecki, K., & Wąsowski, A. (2013, January). A survey of variability modeling in industrial practice. In Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems (p. 7). ACM.
[Berger et al. 2014]	Berger, T., Pfeiffer, R. H., Tartler, R., Dienst, S., Czarnecki, K., Wąsowski, A., & She, S. (2014). Variability mechanisms in software ecosystems. Information and Software Technology, 56(11), 1520-1535.
[Berners-Lee 2006]	Berners-Lee, Tim. 2006. Linked Data. http://www.w3.org/DesignIssues/LinkedData.html.
[Berners-Lee et al. 2001]	Berners-Lee, Tim, James Hendler, and Ora Lassila. 2001. "The Semantic Web." Scientific American 284 (5): 34–43.
[Beuche et al. 2016]	Danilo Beuche, Michael Schulze and Maurice Duvigneau. When 150% Is Too Much: Supporting Product Centric Viewpoints In An Industrial Product Line. In 20st International Systems and Software Product Lines Conference SPLC, Beijing, China, 2016, pages 262-269.
[Bizer and Cyganiak 2009]	Bizer, Christian, and Richard Cyganiak. 2009. "Quality-Driven Information Filtering Using the WIQA Policy Framework." Web Semantics: Science, Services and Agents on the World Wide Web 7 (1): 1–10. doi:10.1016/j.websem.2008.02.005.
[Bizer et al. 2009]	Bizer, Christian, Tom Heath, and Tim Berners-Lee. 2009. "Linked Data - The Story So Far:" International Journal on Semantic Web and Information Systems 5 (3): 1–22. doi:10.4018/jswis.2009081901
[Bkak et al. 2010]	Bąk, K., Czarnecki, K., & Wąsowski, A. (2010). Feature and meta-models in Clafer: mixed, specialized, and coupled. SLE, 102-122.
[Boley et al. 2013]	Boley, Harold, Gary Hallmark, Michael Kifer, Adrian Paschke, Axel Polleres, and Dave Reynolds, eds. 2013. RIF Core Dialect (Second Edition). W3C Recommendation.
[Boneva et al. 2014]	Boneva, Iovka, José Emilio Labra Gayo, Samuel Hym, Eric G. Prud'hommeau, Harold R. Solbrig, and Slawek Staworko. 2014. "Validating RDF with Shape Expressions." CoRR abs/1404.1270.
[Bontemps t al. 2004]	Bontemps, Y., Heymans, P., Schobbens, P. Y., & Trigaux, J. C. (2004, August). Semantics of FODA feature diagrams. In Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation–Towards Tool Support (pp. 48-58).
[Borba et al. 2012]	Borba, P.; Teixeira, L. & Gheyi, R. A Theory of Software Product Line Refinement. Theoretical Computer Science, 2012, 455, 2-30
[Bosch 2001]	J. Bosch. Software product lines: Organizational alternatives. In Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Pages 12-19, 2001.
[Botterweck et al. 2009]	G. Botterweck, A. Pleuss, A. Polzer, and S. Kowaleski. Towards Feature-Driven Planning of Product-Line Eovlution. InProceedings of the First International Workshop on Feature-Oriented Software Development, pages 109-116, 2009.
[Brickley et al. 2014]	Brickley, Dan, and R. V. Guha, eds. 2014. RDF Schema 1.1. W3C Recommendation.
[Castañeda et al. 2010]	Castañeda, Verónica, Luciana Ballejos, Laura Caliusco, and Rosa Galli. 2010. "The Use of Ontologies in Requirements Engineering," Global Journal of Researches In Engineering, 10 (6). http://engineeringresearch.org/index.php/GJRE/article/view/76.
[Chen et al. 2011]	Chen, L., & Babar, M. A. (2011). A systematic review of evaluation of variability management approaches in software product lines. Information and Software Technology, 53(4), 344-362.
[Cicchetti et al. 2008]	Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008, September). Automating co-evolution in model-driven engineering. In Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE (pp. 222-231). IEEE.
[Classen et al. 2010]	Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, Jean-François Raskin: Model checking lots of systems: efficient verification of temporal properties in software product lines. ICSE (1) 2010: 335-344
[Classen et al. 2011]	Classen, A., Boucher, Q., & Heymans, P. (2011). A text-based approach to feature modelling: Syntax and semantics of TVL. Science of Computer Programming, 76(12), 1130-1143.
[Clauß et al. 2001]	Clauß, M., & Jena, I. (2001, September). Modeling variability with UML. In GCSE 2001 Young Researchers Workshop.

[Clements et al. 1996]	Clements, P. C. (1996, March). A survey of architecture description languages. In Software Specification and Design, 1996., Proceedings of the 8th International Workshop on (pp. 16-25). IEEE.
[Clements et al. 2001]	Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2001)
[Configit 2017]	https://configit.com/ Last accessed 2017-09-20
[Cordy et al. 2013]	Cordy, M., Classen, A., Heymans, P., Schobbens, P.-Y., & Legay, A. (2013). ProVeLines: A Product Line of Verifiers for Software Product Lines. (ss. 141-146). ACM.
[Couto et al. 2011]	Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering, CSMR ’11, pages 191–200. IEEE Computer Society, 2011.
[Coyle et al. 2013]	Coyle, Karen, and Tom Baker. 2013. “Dublin Core Application Profiles Separating Validation from Semantics.” W3C RDF Validation Discussion. https://www.w3.org/2001/sw/wiki/images/4/4a/RDFVal_Coyle_Baker.pdf.
[Creff et al. 2012]	Creff, S., Champeau, J., Jézéquel, J.-M., Monégier, A., Model-based product line evolution: an incremental growing by extension. 16th International Software Product Line Conference, pp. 107-114, 2012.
[Cyganiak et al. 2014]	Cyganiak, Richard, and Dave Reynolds. 2014. “The RDF Data Cube Vocabulary.” W3C Recommendation. W3C.
[Czarnecki et al. 2000]	Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Boston, MA (2000)
[Czarnecki et al. 2005]	Czarnecki, K., & Antkiewicz, M. (2005, September). Mapping features to models: A template approach based on superimposed variants. In International conference on generative programming and component engineering (pp. 422-437). Springer, Berlin, Heidelberg.
[Czarnecki et al. 2005b]	Czarnecki, K., Antkiewicz, M., Kim, C. H. P., Lau, S., & Pietroszek, K. (2005, October). Model-driven software product lines. In Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (pp. 126-127). ACM.
[Czarnecki et al. 2005c]	Czarnecki, K., Helsen, S., & Eisenecker, U. (2005). Formalizing cardinality-based feature models and their specialization. Software process: Improvement and practice, 10(1), 7-29.
[Czarnecki et al. 2006]	Czarnecki, K., & Pietroszek, K. (2006, October). Verifying feature-based model templates against well-formedness OCL constraints. In Proceedings of the 5th international conference on Generative programming and component engineering (pp. 211-220). ACM.
[Czarnecki et al. 2012]	Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., & Wąsowski, A. (2012, January). Cool features and tough decisions: a comparison of variability modeling approaches. In Proceedings of the sixth international workshop on variability modeling of software-intensive systems (pp. 173-182). ACM.
[Davis et al. 1993]	Davis, Randall, Howard Shrobe, and Peter Szolovits. 1993. “What Is a Knowledge Representation?” AI Magazine 14 (1): 17.
[Demeyer et al. 2002]	Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. Object-Oriented Reengineering Patterns
[Dhungana et al. 2007]	Dhungana, D., Rabiser, R., Grünbacher, P., & Neumayer, T. (2007, November). Integrated tool support for software product line engineering. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (pp. 533-534). ACM.
[Dhungana et al. 2008]	Dhungana, D., & Grünbacher, P. (2008). Understanding Decision-Oriented Variability Modelling. In SPLC (2) (pp. 233-242).
[Dhungana et al. 2010]	Dhungana, D., Heymans, P., & Rabiser, R. (2010, January). A Formal Semantics for Decision-oriented Variability Modeling with DOPLER. In VaMoS (pp. 29-35).
[Dhungana et al. 2010b]	Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T., Structuring the modeling space and supporting evolution in software product line engineering. Journal of Systems and Software, 83(7), 1108-1122, 2010.
[Dietrich et al. 2012]	C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A robust approach for variability extraction from the Linux build system. In Proceedings of the 16th International Software Product Line Conference (SPLC’12), Vol. 1., pages 21–30, 2012.
[Dintzner et al. 2014]	Nicolas Dintzner , Arie Van Deursen , Martin Pinzger, Extracting feature model changes from the Linux kernel using FMDiff, Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems, January 22-24, 2014, Sophia Antipolis, France.

[Dintzner et al. 2017]	Nicolas Dintzner , Arie Deursen , Martin Pinzger, Analysing the Linux kernel feature model changes using FMDiff, Software and Systems Modeling (SoSyM), v.16 n.1, p.55-76, February 2017
[Dubinsky et al. 2013]	Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., & Czarnecki, K. (2013, March). An exploratory study of cloning in industrial software product lines. In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on (pp. 25-34). IEEE.
[EASy-Producer 2016]	EASy-Producer. https://github.com/SSEHUB/EASyProducer Last visisted: 08.09.2017.
[Eichelberger et al. 2015]	Holger Eichelberger, Sascha El-Sharkawy, Christian Kröher, and Klaus Schmid. Integrated Variability Modeling Language: Language Specification Version 1.27, 2015.
[Eichelberger et al. 2015b]	Holger Eichelberger, Klaus Schmid, Mapping the Design-Space of Textual Variability Modeling Languages: A Refined Analysis. International Journal of Software Tools for Technology Transfer, 17(5):559-584, 2015
[Eichelberger et al., 2013]	Holger Eichelberger, Christian Kröher and Klaus Schmid. An Analysis of Variability Modeling Concepts: Expressiveness vs. Analyzability,In John Favaro and Maurizio Morisio, Editor, Proceedings of the 13th International Conference on Software Reuse (ICSR '13) , pp 32-48. Springer, 2013.
[El-Sharkawy et al. 2017]	Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2017. An Empirical Study of Configuration Mismatches in Linux. In 21st International Systems and Software Product Lines Conference. ACM, New York, NY, USA, 10. DOI:http://dx.doi.org/10.1145/3106195.3106208 Accepted.
[El-Sharkawy et al. 2017b]	Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. Implementation Metrics for Software Product Lines - A Systematic Literature Review. Report No. 1/2017, SSE 1/17/E, Online available https://sse.uni-hildesheim.de/en/research/projects/revamp2/spl-metrics/, 2017.
[EMF 2017]	The Eclipse Foundation. Eclipse Modeling Framework. Online at http://www.eclipse.org/modeling/emf/. Last visited 20.07.2017
[EvoLine 2017]	Support for correct Evolution of Software Product Lines, http://www.dfg-spp1593.de/index.php?id=47
[Eysholdt et al. 2009]	Eysholdt, M., Frey, S., & Hasselbring, W. (2009). EMF Ecore based meta model evolution and model co-evolution. Softwaretechnik-Trends, 29(2), 20-21.
[Falleri et al. 2008]	Falleri, J. R., Huchard, M., Lafourcade, M., & Nebut, C. (2008, September). Metamodel matching for automatic model transformation generation. In International Conference on Model Driven Engineering Languages and Systems (pp. 326-340). Springer Berlin Heidelberg.
[FeatureMapper 2013]	Heidenreich, F. Feature Mapper - Mapping Features to Models. Online at http://featuremapper.org/. Last visited 20.07.2017
[Felfernig et al. 2012]	Alexander Felfernig, Monika Schubert, Christoph Zehentner: An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM 26(1): 53-62 (2012)
[Ferreira et al. 2016]	Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do #ifdefs influence the occurrence of vulnerabilities? An empirical study of the linux kernel. In Proceedings of the 20th International Systems and Software Product Line Conference, SPLC ’16, pages 65–73. ACM, 2016.
[Fischer et al. 2015]	Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO tool: extraction and composition for clone-and-own. In Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15), Vol. 2. IEEE Press, Piscataway, NJ, USA, 665-668.
[Font et al. 2015]	Font, J., Arcega, L., Haugen, Ø., and Cetina, C. (2015, July). Building software product lines from conceptualized model patterns. In Proceedings of the 19th International Conference on Software Product Line (pp. 46-55). ACM.
[Font et al. 2015b]	Font, J., Arcega, L., Haugen, Ø., and Cetina, C. (2015, October). Addressing metamodel revisions in model-based software product lines. In ACM SIGPLAN Notices (Vol. 51, No. 3, pp. 161-170). ACM.
[Font et al. 2016]	Jaime Font, Lorena Arcega, Øystein Haugen, and Carlos Cetina. 2016. Feature location in models through a genetic algorithm driven by information retrieval techniques. In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems(MODELS '16). ACM, New York, NY, USA, 272-282. DOI: https://doi.org/10.1145/2976767.2976789
[Font et al. 2017]	Jaime Font, Lorena Arcega, Øystein Haugen, and Carlos Cetina. 2017. Achieving Feature Location in Families of Models through the use of Search-Based Software Engineering. in IEEE Transactions on Evolutionary Computation, vol. PP, no. 99, pp. 1-1. DOI: https://doi.org/10.1109/TEVC.2017.2751100

[Font et al. 2017b]	Font, J., Arcega, L., Haugen, Ø., and Cetina, C. (2017). Leveraging variability modeling to address metamodel revisions in Model-based Software Product Lines. Computer Languages, Systems & Structures, 48
[Garces et al. 2009]	Garcés, K., Jouault, F., Cointe, P., & Bézivin, J., Managing Model Adaptation by Precise Detection of Metamodel Changes. 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA '09), pp.34-49, 2009.
[Gaševic et al. 2006]	Gaševic, Dragan, Vladan Devedžic, Dragan Djuric, and SpringerLink (Online service). 2006. Model Driven Architecture and Ontology Development. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg.
[Gayo et al. 2015]	Gayo, José Emilio Labra, Eric Prud'hommeaux, Iovka Boneva, Slawek Staworko, Harold R. Solbrig, and Samuel Hym. 2015. “Towards an RDF Validation Language Based on Regular Expression Derivatives.” In Proceedings of the Workshops of the EDBT/ICDT 2015 Joint Conference (EDBT/ICDT), Brussels, Belgium, March 27th, 2015., 197–204. http://ceur-ws.org/Vol-1330/paper-32.pdf.
[Génova et al. 2013]	Génova, Gonzalo, José Miguel Fuentes, Juan Llorens Morillo, Omar Hurtado, and Valentin Moreno. 2013. “A Framework to Measure and Improve the Quality of Textual Requirements.” Requir. Eng. 18 (1): 25–41.
[Góngora et al. 2017]	Chalé Góngora, H. G., Llorens, J. and Gallego, E. (2017), Your Wish, My Command – Speeding up Projects in the Transportation Industry Using Ontologies. INCOSE International Symposium, 27: 1070–1086. doi:10.1002/j.2334-5837.2017.00413.x
[Gonzalez et al. 2005]	González-Baixauli, B., Laguna, M. A., & Crespo, Y. (2005, September). Product lines, features, and MDD. In EWMT 2005 workshop.
[Große-Rhode et al. 2015]	Martin Große-Rhode, Michael Himsolt, and Michael Schulze. The Variability Exchange Language Version 1.0, 2015.
[Groza et al. 2009]	Groza, Tudor, Siegfried Handschuh, Tim Clark, S Buckingham Shum, and Anita de Waard. 2009. “A Short Survey of Discourse Representation Models.”
[Gruschko et al. 2007]	Gruschko, B., Kolovos, D., & Paige, R. (2007, March). Towards synchronizing models with evolving metamodels. In Proceedings of the International Workshop on Model-Driven Software Evolution (p. 3). IEEE.
[Haugen et al. 2008]	Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G. K., & Svendsen, A. (2008, September). Adding standardized variability to domain specific languages. In Software Product Line Conference, 2008. 12th International (pp. 139-148). IEEE.
[Hausenblas et al. 2011]	Hausenblas, Michael, Boris Villazón-Terrazas, and Bernadette Hyland. 2011. “GLD Life Cycle.” W3C Government Linked Data Group. W3C.
[Havelund et al. 2000]	Havelund, K., & Pressburger, T. (2000). Model checking JAVA programs using JAVA PathFinder. International Journal on Software Tools for Technology Transfer, 2(4), 366-381.
[Hayes 2004]	Hayes, Patrick. 2004. “RDF Semantics.” World Wide Web Consortium. http://www.w3.org/TR/rdf-mt/.
[Heidenreich et al. 2008]	Heidenreich, F., Kopcsek, J., & Wende, C. (2008, May). FeatureMapper: mapping features to models. In Companion of the 30th international conference on Software engineering (pp. 943-944). ACM.
[Hellebrand et al. 2017]	Robert Hellebrand, Michael Schulze, and Uwe Ryssel. Reverse engineering challenges of the feedback scenario in co-evolving product lines. In Proceedings of the 5th International Workshop on Reverse Variability Engineering (REVE), Sevilla, Spain, 2017, Accepted for Publication.
[Herrmannsdoerfer et al. 2008]	Herrmannsdoerfer, M., Benz, S., & Juergens, E. (2008). Automatability of coupled evolution of metamodels and models in practice. Model Driven Engineering Languages and Systems, 645-659.
[Herrmannsdoerfer et al. 2009]	Herrmannsdoerfer, M., Benz, S., & Juergens, E., COPE - Automating Coupled Evolution of Metamodels and Models. 23rd European Conference on Object-Oriented Programming (ECOOP '09), pp. 52-76, 2009.
[Herrmannsdoerfer et al. 2009b]	Herrmannsdoerfer, M., Ratiu, D., & Wachsmuth, G. (2009). Language Evolution in Practice: The History of GMF. SLE, 9, 3-22.
[Hitzler et al. 2009]	Hitzler, Pascal, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. 2009. “OWL 2 Web Ontology Language Primer.” W3C Recommendation. World Wide Web Consortium. http://www.w3.org/TR/owl2-primer/.
[Hogan et al. 2012]	Hogan, Aidan, Jürgen Umbrich, Andreas Harth, Richard Cyganiak, Axel Polleres, and Stefan Decker. 2012. “An Empirical Survey of Linked Data Conformance.” Web Semantics: Science, Services and Agents on the World Wide Web 14 (July): 14–44. doi:10.1016/j.websem.2012.02.001.

[Hohpe and Woolf 2004]	Hohpe, Gregor, and Bobby Woolf. 2004. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. The Addison-Wesley Signature Series. Boston: Addison-Wesley.
[Horrocks et al. 2005]	Horrocks, Ian, Bijan Parsia, Peter Patel-Schneider, and James Hendler. 2005. “Semantic Web Architecture: Stack or Two Towers?” In Principles and Practice of Semantic Web Reasoning, 37–41. Springer.
[Hotz et al. 2006]	Hotz, L., Wolter, K., Krebs, T., Deelstra, S., Sinnema, M., Nijhuis, J., & MacGregor, J. (2006). Configuration in industrial product families. Ios Press.
[Hubaux et al. 2010]	Hubaux, A., Boucher, Q., Hartmann, H., Michel, R., & Heymans, P. (2010). Evaluating a Textual Feature Modelling Language: Four Industrial Case Studies. SLE, 10, 337-356.
[Hull and King 1987]	Hull, Richard, and Roger King. 1987. “Semantic Database Modeling: Survey, Applications, and Research Issues.” ACM Computing Surveys (CSUR) 19 (3): 201–260.
[Hunsen et al. 2016]	Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor based variability in open-source and industrial software systems: An empirical study. Empirical Softw. Engg., 21:449–482, Apr 2016.
[IEEE1076 2009]	IEEE Standard VHDL Language Reference Manual," in IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002) , vol., no., pp.c1-626, Jan. 26 2009
[IEEE1364 2006]	IEEE Standard for Verilog Hardware Description Language," in IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001) , vol., no., pp.0_1-560, 2006
[Jazayeri et al. 2000]	Jazayeri M, Ran A, van der Linden F (2000) Software architecture for product families: principles and practice. Addison Wesley
[Jia et al. 2011]	Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649-678, Sept.-Oct. 2011. doi: 10.1109/TSE.2010.62
[Käkölä and Dueñas 2006]	Käkölä, Timo, Dueñas, Juan Carlos. Software Product Lines: Research Issues in Engineering and Management, Springer 2006
[Kang et al. 1990]	Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study (No. CMU/SEI-90-TR-21). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
[Kang et al. 2009]	Kang, K. C. (2009). FODA: twenty years of perspective on feature models. the keynote of SPLC.
[Kang et al. 2013]	Kang, K. C., & Lee, H. (2013). Variability modeling. In Systems and Software Variability Management (pp. 25-42). Springer Berlin Heidelberg.
[Kästner et al. 2008]	Christian Kästner, Sven Apel, and Martin Kuhlemann. "Granularity in software product lines." Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. IEEE, 2008.
[Kästner et al. 2011]	Christian Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In Proceedings of the 2011 International Conference on Object Oriented Programming Systems Languages and Applications, pages 805-824, 2011
[Kästner et al. 2012]	Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A Variability-Aware Module System. In Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 773–792, New York, NY: ACM Press, October 2012.
[Kbuild 2017]	Linux Kernel Makefiles. https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt Last visited 14.02.2017.
[KBuildMiner 2017]	KBuildMiner. https://github.com/ckaestne/KBuildMiner Last visited: 03.03.2017.
[Kconfig 2017]	Kconfig Language Specification. https://www.kernel.org/doc/Documentation/kbuild/kcon_g-language.txt Last visited 13.02.2017
[KconfigReader 2017]	KconfigReader. https://github.com/ckaestne/kconfigreader. Last visited 11.07.2017.
[Kenner et al. 2010]	A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Towards Type Checking #ifdef Variability in C. In Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, pages 25-32, 2010.
[KernelHaven 2017]	KernelHaven. https://github.com/KernelHaven/KernelHaven Last visited: 13.07.2017.
[Kim et al. 2005]	Kim, S. D., Min, H. G., Her, J. S., & Chang, S. H. (2005, July). DREAM: A practical product line engineering using model driven architecture. In Information Technology and Applications, 2005. ICITA 2005. Third International Conference on (Vol. 1, pp. 70-75). IEEE.

[Knublauch et al. 2011]	Holger Knublauch, James A. Hendler, and Kingsley Idehen. 2011. “SPIN - Overview and Motivation.” Member Submission. W3C. http://www.w3.org/Submission/spin-overview/.
[Kogut et al. 1994]	Kogut, P., & Clements, P. (1994). Features of architecture description languages. In In Proceedings of the Eighth International Workshop on Software Specification and Design.
[Kontokostas et al. 2014]	Kontokostas, Dimitris, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. 2014. “Test-Driven Evaluation of Linked Data Quality.” In 23rd International World Wide Web Conference, WWW ’14, Seoul, Republic of Korea, April 7-11, 2014, 747–758. doi:10.1145/2566486.2568002.
[Kossmann et al. 2008]	Kossmann, Mario, Richard Wong, Mohammed Odeh, and Andrew Gillies. 2008. “Ontology-Driven Requirements Engineering: Building the OntoREM Meta Model.” In , 1–6. IEEE. doi:10.1109/ICTTA.2008.4530315.
[Krafzig et al. 2005]	Krafzig, Dirk, Karl Banke, and Dirk Slama. 2005. Enterprise SOA: Service-Oriented Architecture Best Practices. Prentice Hall Professional.
[Krishna et al. 2005]	Krishna, A. S., Gokhale, A., Schmidt, D. C., Ranganath, V. P., Hatcliff, J., & Schmidt, D. C. (2005, October). Model-driven middleware specialization techniques for software product-line architectures in distributed real-time and embedded systems. In Proceedings of the MODELS 2005 workshop on MDD for Software Product-lines.
[Kroeher et al. 2017]	Kröher, C. & Schmid, K., (2017). Towards a Better Understanding of Software Product Line Evolution. Softwaretechnik-Trends: Vol. 37, No. 2. Berlin: Gesellschaft für Informatik e.V., Fachgruppe PARS. (S. 40-41).
[Krueger 2001]	C. W. Krueger, Easing the transition to software mass customization, in PFE, 2001.
[Krueger et al. 2007]	Krueger, C. W. (2007, October). Biglever software gears and the 3-tiered spl methodology. In Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion (pp. 844-845). ACM.
[Krüger et al. 2016]	Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich and Gunter Saake. Extracting software product lines: a cost estimation perspective, SPLC (REVE) 2016
[Lapouchnian et al. 2009]	Lapouchnian, A., & Mylopoulos, J. (2009). Modeling domain variability in requirements engineering with contexts. Conceptual Modeling-ER 2009, 115-130.
[Lauenroth et al. 2010]	Lauenroth, K., Metzger, A. and Pohl, K., 2010. Quality assurance in the presence of variability. In Intentional Perspectives on Information Systems Engineering (pp. 319-333). Springer Berlin Heidelberg.
[Lauterbach 2017]	http://www.lauterbach.com/frames.html?home.html
[Leitner et al. 2012]	Andrea Leitner, Reinhold Weiß, and Christian Kreiner. Analyzing the complexity of domain model representations. In Proceedings of the 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems, ECBS ’12, pages 242–248. IEEE Computer Society, 2012.
[Liebig et al. 2010]	Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE ’10, pages 105–114. ACM, 2010.
[Linsbauer et al. 2016]	Lukas Linsbauer , Alexander Egyed , Roberto Erick Lopez-Herrejon, A variability aware configuration management and revision control platform, Proceedings of the 38th International Conference on Software Engineering Companion, May 14-22, 2016, Austin, Texas
[Llorens et al. 2004]	Llorens, Juan, Jorge Morato, and Gonzalo Genova. 2004. “RSHP: An Information Representation Model Based on Relationships.” In Soft Computing in Software Engineering, edited by Ernesto Damiani, Mauro Madravio, and LakhmiC. Jain, 159:221–53. Studies in Fuzziness and Soft Computing. Springer Berlin Heidelberg. http://dx.doi.org/10.1007/978-3-540-44405-3_8.
[Lopez-Herrejon et al. 2008]	Roberto E. Lopez-Herrejon and Salvador Trujillo. How complex is my product line? the case for variation point metrics. In Second International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS’08, pages 97–100, 2008.
[Lopez-Herrejon et al. 2013]	Roberto E. Lopez-Herrejon, Alexander Egyed, Towards interactive visualization support for pairwise testing software product lines. VISSOFT 2013
[Lotufo et al. 2010]	Lotufo R., She S., Berger T., Czarnecki K., Wąsowski A. (2010) Evolution of the Linux Kernel Variability Model. In: Bosch J., Lee J. (eds) Software Product Lines: Going Beyond. SPLC 2010. Lecture Notes in Computer Science, vol 6287. Springer, Berlin, Heidelberg

[Lüdemann et al. 2016]	Dierk Lüdemann, Nazish Asad, Klaus Schmid, Christopher Voges. Understanding Variable Code: Reducing the Complexity by Integrating Variability Information. 2016 IEEE ICSME '16, pages 312 - 322. IEEE. 2016.
[Maâzoun et al. 2016]	Jihen Maâzoun, Nadia Bouassida, and Hanêne Ben-Abdallah. Change impact analysis for software product lines. J. King Saud Univ. Comput. Inf. Sci., 28:364–380, Oct 2016.
[Macia et al. 2010]	Isela Macia, Alessandro Garcia, and Arndt von Staa. Defining and applying detection strategies for aspect-oriented code smells. In Proceedings of the 2010 Brazilian Symposium on Software Engineering, SBES '10, pages 60–69. IEEE Computer Society, 2010.
[Mallea et al. 2011]	Mallea, Alejandro, Marcelo Arenas, Aidan Hogan, and Axel Polleres. 2011. “On Blank Nodes.” In The Semantic Web–ISWC 2011, 421–437. Springer.
[Manickam et al. 2013]	Manickam, P., Sangeetha, S., & Subrahmanya, S. V. (2013). Component-Oriented Development and Assembly: Paradigm, Principles, and Practice using Java. CRC Press.
[Manikas et al. 2013]	Manikas, Konstantinos, and Klaus Marius Hansen. 2013. “Software Ecosystems – A Systematic Literature Review.” Journal of Systems and Software 86 (5): 1294–1306. doi:10.1016/j.jss.2012.12.026.
[Mann et al. 2011]	Stefan Mann and Georg Rock. Control variant-rich models by variability measures. In Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '11, pages 29–38. ACM, 2011.
[Martinez et al. 2014]	J. Martinez, T. Ziadi, J. Klein, and Y. L. Traon, Identifying and visualising commonality and variability in model variants, in ECMFA, 2014.
[Martinez et al. 2014b]	J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyandé, J. Klein, and Y. L. Traon, Feature relations graphs: A visualisation paradigm for feature constraints in software product lines, in VISSOFT, 2014.
[Martinez et al. 2015]	J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, Bottom-up adoption of software product lines: a generic and extensible approach, in SPLC, 2015.
[Martinez et al. 2015b]	J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, Automating the extraction of model-based software product lines from model variants, in ASE, 2015.
[Martinez et al. 2016]	J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, Name Suggestions during Feature Identification: The VariClouds Approach, in SPLC, 2016
[Martinez et al. 2016b]	J. Martinez, T. Ziadi, M. Papadakis, T. F. Bissyandé, J. Klein, and Y. L. Traon, Feature location benchmark for software families using eclipse community releases, in ICSR, 2016.
[Martinez et al. 2016c]	L. Li, J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, Mining Families of Android Applications for Extractive SPL Adoption, in SPLC, 2016.
[Martinez et al. 2017]	J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. L. Traon, Bottom-Up Technologies for Reuse: Automated Extractive Adoption of Software Product Lines, in ICSE, Volume 2, 2017
[Martinez et al. 2017b]	J. Martinez, W. K. Assunção, T. Ziadi: ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In SPLC, Volume 2, 2017
[Mazo 2011]	Raúl Mazo: A Generic Approach for Automated Verification of Product Line Models. Pantheon-Sorbonne University, Paris, France 2011
[Mefteh et al. 2015]	Mariem Mefteh, Nadia Bouassida, and Hanêne Ben-Abdallah. Implementation and evaluation of an approach for extracting feature models from documented uml use case diagrams. In Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15, pages 1602–1609. ACM, 2015
[MOF 2017]	Object Management Group. Meta-Object Facility. Online at http://www.omg.org/mof/. Last visited 20.07.2017.
[Montalvillo and Díaz 2015]	Leticia Montalvillo and Oscar Díaz. 2015. Tuning GitHub for SPL development: branching models & repository operations for product engineers. In Proceedings of the 19th International Conference on Software Product Line (SPLC '15). ACM, New York, NY, USA, 111-120. DOI=http://dx.doi.org/10.1145/2791060.2791083
[Muthig et al. 2002]	Muthig, D., & Atkinson, C. (2002). Model-driven product line architectures. Software Product Lines, 79-90.
[Nadi et al. 2011]	Sarah Nadi and R. Holt. Make it or break it: Mining anomalies from linux kbuild. In Proceedings of the 18th Working Conference on Reverse Engineering, pages 315–324, 2011.
[Nadi et al. 2012]	Sarah Nadi and R. Holt. Mining Kbuild to detect variability anomalies in Linux. In Proceedings of the 16th European Conference on Software Maintenance and Reengineering, pages 107–116, 2012.

[Nadi et al. 2013]	Sarah Nadi, Christian Dietrich, Reinhard Tartler, Richard C. Holt, and Daniel Lohmann. 2013. Linux variability anomalies: what causes them and how do they get fixed?. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13). IEEE Press, Piscataway, NJ, USA, 111-120.
[Nadi et al. 2014]	Sarah Nadi and Ric Holt. The linux kernel: A case study of build system variability. Journal of Software: Evolution and Process, 26:730–746, Aug 2014.
[Narayanan et al. 2009]	Narayanan, A., Levendovszky, T., Balasubramanian, D., Karsai, G., Automatic domain model migration to manage metamodel evolution, Model Driven Engineering Languages and Systems (MoDELS'09), pp. 706-711, 2009.
[Nestor et al. 2008]	Daren Nestor, Ste en Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. 2008. Applying visualisation techniques in software product lines. 4th ACM symposium on Software visualization. ACM, 175–184.
[Neves et al. 2015]	Neves, L.; Borba, P.; Alves, V.; Turnes, L.; Teixeira, L.; Sena, D. & Kulesza, U. Safe Evolution Templates for Software Product Lines. Journal of Systems and Software, 2015, 106, 42-58
[Nguyen et al. 2014]	Nguyen, Vinh, Olivier Bodenreider, and Amit Sheth. 2014. "Don't like RDF Reification?: Making Statements about Statements Using Singleton Property." In , 759–70. ACM Press. doi:10.1145/2566486.2567973.
[Nieke et al. 2016]	M. Nieke, C. Seidl, and S. Schuster. Guaranteeing Configuration Validity in Evolving Software Product Lines. In Proceedings of the 10th International Workshop on Variability Modelling of Software-intensive Systems, pages 73-80, 2016.
[NIST 1993]	NIST Special Publication 500-21 1. Reference Model for Frameworks of Software Engineering Environments (Technical Report ECMA TR/55, 3rd ed.), National Institute of Standards and Technology, August 1993.
[Nonaka et al. 1995]	Nonaka, Ikujiro, and Hirotaka Takeuchi. 1995. The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation. New York: Oxford University Press
[Noorian et al. 2011]	Noorian M, Ensan A, Bagheri E, Boley H, Biletskiy Y. Feature Model Debugging based on Description Logic Reasoning. InDMS 2011 Aug 18 (Vol. 11, pp. 158-164).
[Northrop 2004]	L. Northrop. Software product line adoption roadmap. Technical Report CMU/SEI-2004-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2004.
[Northrop et al. 2009]	L. Northrop, P. C. Clements, et al. A Framework for Software Product Line Practice, Version 5.0. www.sei.cmu.edu/productlines/framework.html, 2009.
[Noy and Rector 2006]	Noy, Natasha, and Alan Rector. 2006. "Defining N-Ary Relations on the Semantic Web." W3C Working Group. http://www.w3.org/TR/swbp-n-aryRelations/.
[OMG-CVL 2012]	OMG. Common variability language (CVL), OMG revised submission 2012. OMG document: ad/2012-08-05, 2012.
[Osman et al. 2009]	Elfaki, A.O., Phon-Amnuaisuk, S. and Ho, C.K., 2009. Using First Order Logic to Validate Feature Model. In VaMoS (pp. 169-172).
[Ottlik et al. 2017]	Ottlik, Sebastian, et al. "Context-sensitive timing automata for fast source level simulation." 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2017.
[Parsai et al. 2017]	Parsai, A., Murgia, A., Demeyer, S.: Littledarwin: a feature-rich and extensible mutation testing framework for large and complex java systems. In: Proceedings of 7th IPM International Conference on Fundamentals of Software Engineering 2017
[Passos et al. 2011]	Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wasowski. A study of non-boolean constraints in variability models of an embedded operating system. In Proceedings of the 15th International Software Product Line Conference, Volume 2, SPLC '11, pages 2:1–2:8. ACM, 2011.
[Passos et al. 2013]	Passos, L., Czarnecki, K., Apel, S., Wasowski, A., Kästner, C., Guo, J., Hunsen, C., Feature-oriented software evolution. 7th International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '13). Article 17, 2013.
[Passos et al. 2016]	Passos, L.; Teixeira, L.; Dintzner, N.; Apel, S.; Wasowski, A.; Czarnecki, K.; Borba, P. & Guo, J. Coevolution of Variability Models and Related Software Artifacts - A Fresh Look at Evolution Patterns in the Linux Kernel. Empirical Software Engineering, 2016, 21, 1744-1793
[Pohl et al. 2005]	Pohl, K., Böckle, G., Van Der Linden, F.: Software product line engineering: foundations, principles, and techniques. Springer-Verlag New York Inc (2005)
[Powers 2003]	Powers, Shelley. 2003. Practical RDF. Beijing ; Sebastopol: O'Reilly.
[PureVariants 2006]	Pure Systems GmbH: Technical white paper variant management with pure::variants. Tech. rep. (2006), http://www.pure-systems.com/fileadmin/downloads/ pv-whitepaper-en-04.pdf

[Reimann et al. 2010]	Reimann, J.; Seifert, M. & Aßmann, U. Role-Based Generic Model Refeactoring. Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II, Springer, Berlin, Heidelberg, 2010, 78-92
[Ribeiro et al. 2010]	M. Ribeiro, H. Pacheco, L. Teixeira, and P. Borba. Emergent Feature Modularization. Proceedings of the International Conference Companion on Object Oriented Programming Systems Languages and Applications, pages 11–18, 2010.
[Ribeiro et al. 2011]	M. Ribeiro, F. Queiroz, P. Borba, T. Toedo, C. Brabrand, and S. Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In Proceedings of the 10th International Conference on Generative Programming and Component Engineering, pages 23-32, 2011.
[Rincon et al. 2015]	L. F. Rincón, Gloria Lucía Giraldo G., Raúl Mazo, Camille Salinesi, Daniel Diaz: Method to Identify Corrections of Defects on Product Line Models. Electr. Notes Theor. Comput. Sci. 314: 61-81 (2015)
[Riva and del Rosso 2003]	C. Riva and C. Del Rosso: Experiences with software product family evolution. International Workshop on Principles of Software Evolution 2003
[Rodríguez et al. 2012]	Rodríguez, Miguel García, Jose María Alvarez-Rodríguez, Diego Berrueta Muñoz, Luis Polo Paredes, José Emilio Labra Gayo, and Patricia Ordóñez de Pablos. 2012. “Towards a Practical Solution for Data Grounding in a Semantic Web Services Environment.” J. UCS (JUCS) 18 (11): 1576–97.
[Roman et al. 2005]	Roman, Dumitru, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. 2005. “Web Service Modeling Ontology.” Applied Ontology 1 (1): 77–106.
[Romero et al. 2013]	Romero, D.; Urli, S.; Quinton, C.; Blay-Fornarino, M.; Collet, P.; Duchien, L. & Mosser, S.SPLEMMA: A Generic Framework for Controlled-Evolution of Software Product Lines. Proceedings of the 17th International Software Product Line Conference co-located workshops, ACM, 2013, 59-66
[Rose et al. 2009]	Rose, L. M., Paige, R. F., Kolovos, D. S., & Polack, F. A. (2009, October). An analysis of approaches to model migration. In Proc. Joint MoDSE-MCCM Workshop (pp. 6-15).
[Rose et al. 2010]	Rose, L., Kolovos, D., Paige, R., & Polack, F., Model Migration with Epsilon Flock. Third International Conference on Model Transformation (ICMT '10), pp. 184-198, 2010.
[Ryman et al. 2013]	Ryman, Arthur G., Arnaud Le Hors, and Steve Speicher. 2013. “OSLC Resource Shape: A Language for Defining Constraints on Linked Data.” In LDOW
[Salinesi and Mazo 2012]	Camille Salinesi, Raúl Mazo. Defects in Product Line Models and how to Identify them. Abdelrahman Elfaki. Software Product Line - Advanced Topic, InTech editions, pp.50, 2012, 978-953-51-0436-0.
[Sampaio et al. 2016]	Sampaio, G.; Borba, P. & Teixeira, L. Partially Safe Evolution of Software Product Lines. Proceedings of the 20th International Systems and Software Product Line Conference, 2016, 124-133
[Sánchez et al. 2014]	Ana B. Sánchez, Sergio Segura, and Antonio Ruiz-Cortés. A comparison of test case prioritization criteria for software product lines. In Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation, ICST ’14, pages 41–50. IEEE Computer Society, 2014.
[SAT 2015]	Satisfiability Suggested Format. ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi, 1993. Last visited 04.03.2015.
[Schaefer et al. 2011]	Schaefer, I., Gurov, D., & Soleimanifard, S. (2011). Compositional Algorithmic Verification of Software Product Lines. (ss. 184-203). Springer-Verlag.
[Schmid 2000]	Klaus Schmid. Scoping Software Product Lines - An Analysis of an Emerging Technology. In Patrick Donohoe, Editor, Proceedings of the 1st Software Product Line Conference (SPLC 1) - Software Product Lines: Experience and Research Directions , pp. 513-532. Kluwer, 2000.
[Schmid et al. 2004]	Klaus Schmid, Isabel John. A Customizable Approach to Full Lifecycle Variability Management. Science of Computer Programming, 53(3):259-284,2004
[Schmid et al. 2015]	Schmid, K., & Eichelberger, H. (2015, July). EASy-Producer: from product lines to variability-rich software ecosystems. In Proceedings of the 19th International Conference on Software Product Line (pp. 390-391). ACM.
[Schmidt et al. 2005]	Schmidt, D., Nechypurenko, A., & Wuchner, E. (2005, October). MDD for Software Product-Lines: Fact or Fiction. In Models 2005 Workshop (Vol. 9).
[Schobbens et al. 2007]	Schobbens, P. Y., Heymans, P., Trigaux, J. C., & Bontemps, Y. (2007). Generic semantics of feature diagrams. Computer Networks, 51(2), 456-479.

[Schroeter et al. 2016]	R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In Proceedings of the 38th International Conference on Software Engineering, pages 667-678, 2016.
[Schulze et al. 2015]	Michael Schulze and Robert Hellebrand. Variability Exchange Language - A Generic Exchange Format for Variability Data. In Software Engineering (Workshops), Dresden, Germany, 2015, pages 37-46.
[Schulze et al. 2016]	Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. Aligning Coevolving Artifacts Between Software Product Lines and Products. In Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems (VaMOS), Salvador, Brazil, 2016, pages 9-16.
[Schwägerl et al. 2017]	Felix Schwägerl , Bernhard Westfechtel, Perspectives on combining model-driven engineering, software product line engineering, and version control, Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, February 01-03, 2017, Eindhoven, Netherlands
[Segura et al. 2012]	Sergio Segura, José A. Galindo, David Benavides, José Antonio Parejo, Antonio Ruiz Cortés: BeTTy: benchmarking and testing on the automated analysis of feature models. VaMoS 2012:
[Seidl et al. 2012]	Seidl, C.; Heidenreich, F. & Aßmann, U. Co-Evolution of Models and Feature Mapping in Software Product Lines. Proceedings of the 16th International Software Product Line Conference, ACM, 2012, 1, 76-85
[She et al. 2010]	She, S., & Berger, T. (2010). Formal semantics of the Kconfig language. Technical note, University of Waterloo, 24.
[Sinnema et al. 2004]	Sinnema, M., Deelstra, S., Nijhuis, J., & Bosch, J. (2004). Covamof: A framework for modeling variability in software product families. Software product lines, 25-27.
[Sinnema et al. 2007]	Sinnema, M., & Deelstra, S. (2007). Classifying variability modeling techniques. Information and Software Technology, 49(7), 717-739.
[Sirin et al. 2007]	Sirin, Evren, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. 2007. “Pellet: A Practical OWL-DL Reasoner.” J. Web Sem. 5 (2): 51–53. doi:10.1016/j.websem.2007.03.004.
[Sprinkle 2003]	Sprinkle, J. M., & Karsai, G. (2003). Metamodel driven model migration(Doctoral dissertation, Vanderbilt University).
[Sprinkle 2004]	Sprinkle, J., & Karsai, G. (2004). A domain-specific visual language for domain model evolution. Journal of Visual Languages & Computing, 15(3), 291-307.
[Stoiber et al. 2007]	Stoiber, R., Meier, S., & Glinz, M. (2007, October). Visualizing product line domain variability by aspect-oriented modeling. In Requirements Engineering Visualization, 2007. REV 2007. Second International Workshop on (pp. 8-8). IEEE.
[Sun et al. 2005]	Sun, J., Zhang, H., Fang, Y. and Wang, L.H., 2005, June. Formal semantics and verification for feature modeling. In Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on (pp. 303-312). IEEE.
[Svahnberg et al. 1999]	Svahnberg, M. & Bosch, J. Evolution in Software Product Lines: Two Cases. Journal of Software Maintenance: Research and Practice, 1999, 11, 391-422
[Svendsen et al. 2010]	Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., & Olsen, G. (2010). Developing a software product line for train control: A case study of cvl. Software Product Lines: Going Beyond, 106-120.
[Tartler et al. 2011]	Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time–Configurable System Software: Facing the Linux 10,000 Feature Problem. In 6th Conference on Computer Systems. ACM, New York, NY, USA, 47–60.
[Tartler et al. 2014]	R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. In Proceedings of the 2014 USENIX Annual Technical Conference, pages 421-432, 2014.
[The Reuse Company Inc. 2014]	The Reuse Company Inc. 2014. “knowlegeMANAGER (KM).” Industry website. knowledgeMANAGER. http://www.reusecompany.com/knowledgemanager.
[Thüm et al. 2012]	Thüm, T., Sven Apel, Christian Kästner, Martin Kuhlemann, Ina Schaefer, and Gunter Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, University of Magdeburg, 2012.
[Thüm et al. 2014]	Thüm, Thomas, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. “A Classification and Survey of Analysis Strategies for Software Product Lines.” ACM Computing Surveys 47 (1): 1–45. doi:10.1145/2580950.
[Thüm et al. 2014b]	Thüm, T., Kaustner, C., Benduhn, F., Meinicke, J., Saake, G., & Leich, T. (2014). FeatureIDE: An extensible framework for feature-oriented software development. Science of Computer Programming, 79, 70-85.

[Thüm et al. 2014c]	Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., Rhein, A. v., & Saake, G. (2014). Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. (ss. 177-186). ACM.
[Torres et al. 2010]	Mário Torres, Uirá Kulesza, Matheus Sousa, Thais Batista, Leopoldo Teixeira, Paulo Borba, Elder Cirilo, Carlos Lucena, Rosana Braga, and Paulo Masiero. Assessment of product derivation tools in the evolution of software product lines: An empirical study. In Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development, FOSD ’10, pages 10–17. ACM, 2010.
[TypeChef 2017]	TypeChef. https://ckaestne.github.io/TypeChef/. Last visited 11.07.2017.
[Undertaker 2017]	CADOS Undertaker. http://vamos.informatik.uni-erlangen.de/trac/undertaker. Last visited 11.07.2017
[Vale et al. 2015]	Gustavo Vale, Danyllo Albuquerque, Eduardo Figueiredo, and Alessandro Garcia. Defining metric thresholds for software product lines: A comparative study. In Proceedings of the 19th International Conference on Software Product Line, SPLC ’15, pages 176–185. ACM, 2015.
[Vale et al. 2015b]	Gustavo Andrade Do Vale and Eduardo Magno Lages Figueiredo. A method to derive metric thresholds for software product lines. In Proceedings of the 2015 29th Brazilian Symposium on Software Engineering, SBES ’15, pages 110–119. IEEE Computer Society, 2015.
[Van der Linden 2002]	F. van der Linden: Software product families in Europe: the Esaps & Cafe projects. IEEE software
[Van der Linden et al. 2007]	F. van der Linden, K. Schmid, E. Rommes. Software Product Lines in Action. Springer. 2007
[Van Ommering et al. 2000]	Van Ommering, R., Van Der Linden, F., Kramer, J., & Magee, J. (2000). The Koala component model for consumer electronics software. Computer, 33(3), 78-85.
[VARIES D4.7 2012]	Raghava Rao Mukkamala (2012), VARiability In safety-critical Embedded Systems D4.7 Variability Analysis Solutions, http://www.varies.eu/wp-content/uploads/sites/8/2013/05/VARIES_D4.7_v01_PU_FINAL.pdf
[Vasilevskiy et al. 2015]	A. Vasilevskiy, Ø. Haugen, F. Chauvel, M. Johansen, and D. Shimbara. (2015) The BVR tool bundle to support product line engineering. SPLC, page 380-384. ACM
[Vasilevskiy et al. 2016]	Vasilevskiy, A., F. Chauvel and Ø. Haugen (2016). Toward Robust Product Realisation in Software Product Lines. SPLC '16 Proceedings of the 20th International Systems and Software Product Line Conference, ACM Digital Library: 369.
[Voelter et al. 2013]	Voelter, M. (2013). Language and IDE Modularization and Composition with MPS. In Generative and transformational techniques in software engineering IV (pp. 383-430). Springer Berlin Heidelberg.
[Wachsmuth et al. 2007]	Wachsmuth, G. (2007, July). Metamodel adaptation and model co-adaptation. In ECOOP (Vol. 7, pp. 600-624).
[Wang et al. 2007]	Wang, C., Xiong, M., Zhou, Q. and Yu, Y., 2007, June. Panto: A portable natural language interface to ontologies. In European Semantic Web Conference (pp. 473-487). Springer, Berlin, Heidelberg.
[Wasowski et al. 2004]	Wasowski, A. (2004, January). Automatic generation of program families by model restrictions. In SPLC (Vol. 3154, pp. 73-89).
[Wassermann, 1990]	Wasserman A.I. (1990) Tool integration in software engineering environments. Lecture Notes in Computer Science, vol 467. Springer, Berlin, Heidelberg (https://link.springer.com/chapter/10.1007/3-540-53452-0_38)
[White et al. 2008]	White, J., & Schmidt, D. C. (2008). Model-driven product-line architectures for mobile devices. IFAC Proceedings Volumes, 41(2), 9296-9301.
[White et al. 2010]	White, J., Benavides, D., Schmidt, D.C., Trinidad, P., Dougherty, B. and Ruiz-Cortes, A., 2010. Automated diagnosis of feature model configurations. Journal of Systems and Software, 83(7), pp.1094-1107.
[Wimmer et al. 2010]	Wimmer, M., Kusel, A., Schoenboeck, J., Retschitzegger, W., Schwinger, W., Kappel, G., On using Inplace Transformations for Model Co-evolution. Proceedings of 2nd International Workshop on Model Transformation with ATL, pages 14, 2010.
[XFeature 2017]	Xfeature, https://www.pnp-software.com/XFeature/; last accessed 2017-09-20
[Xing et al. 2013]	Z. Xing, Y. Xue, and S. Jarzabek. A large scale linux-kernel based benchmark for feature location research. In Proced. of Intern. Conf. on Soft. Eng., ICSE, pages 1311–1314, 2013.
[Xinnian et al. 2017]	Zheng, Xinnian, Lizy K. John, and Andreas Gerstlauer. "LACross: Learning-Based Analytical Cross-Platform Performance and Power Prediction." International Journal of Parallel Programming (2017): 1-27.

[Zhang et al. 2004]	Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In International Conference on Formal Engineering Methods, pages 115–130. Springer, 2004.
[Zhang et al. 2012]	Bo Zhang and Martin Becker. Code-based variability model extraction for software product line improvement. In Proceedings of the 16th International Software Product Line Conference - Volume 2, SPLC ’12, pages 91–98. ACM, 2012.
[Zhang et al. 2013]	Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. Variability evolution and erosion in industrial product lines: A case study. In Proceedings of the 17th International Software Product Line Conference, SPLC ’13, pages 168–177. ACM, 2013.
[Ziegler et al. 2016]	A. Ziegler, V. Rothberg, and D. Lohmann. Analyzing the Impact of Feature Changes in Linux. In Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, pages 25-32, 2016.
[Zimmermann et al. 2012]	J. Zimmermann, S. Stattelmann, A. Viehl, O. Bringmann, and W. Rosenstiel. Model-driven virtual prototyping for real-time simulation of distributed embedded systems. In 7th IEEE International Symposium on Industrial Embedded Systems (SIES), pages 201 - 210, 2012.