Formal Verification and Automated analysis of Software Product Lines

By

Ganesh Khandu Narwane

(Enrollment Number: ENGG01201104001)

Bhabha Atomic Research Centre, Mumbai

A thesis submitted to the

Board of Studies in Engineering Sciences

In partial fulfillment of requirements

for the degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

September, 2018

Homi Bhabha National Institute

Recommendations of the Viva Voce Committee

As members of the Viva Voce Board, we certify that we have read the dissertation prepared by Ganesh Khandu Narwane entitled "Formal Verification and Automated analysis of Software Product Lines" and recommend that it may be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Chairman	Dr. Archana Sharma, BARC	Archana	Date: Archana 25.9.18
Guide	Dr. A.K. Bhattacharjee, BARC	A.K.Bhattacharjee	Date: 28/9/2018
Co-Guide	Prof. Krishna S., IIT Bombay	Krishna	Date: 28/9/2018
Examiner	Prof. Meenakshi D'Souza	D. Meenakshi	Date: 28/9/2018
Member 1	Dr. Gopika Vinod, BARC	Gopika	Date: 28/9/2018
Member 2	Dr. V.H. Patankar, BARC	V.H.Patankar	Date: 28.09.2018
Member 3	Dr. B. Dikshit, BARC	B.Dikshit	Date: 28/9/18

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to HBNI. We hereby certify that, we have read this dissertation prepared under our direction and recommend that it may be accepted as fulfilling the dissertation requirement.

Date: 26/10/2018
Place: Mumbai

Krishna
Co-Guide

A.K.Bhattacharjee
Guide

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfilment of requirements for an advanced degree at Homi Bhabha National Institute (HBNI) and is deposited in the Library to be made available to borrowers under rules of the HBNI.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Competent Authority of HBNI when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Ganesh Khandu Narwane

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree/diploma at this or any other Institution/University.

Ganesh Khandu Narwane

List of publications arising from the thesis

Journals

1. *Design Variability Verification and Compositional Modeling in Real Time Software Product Lines*, Ganesh Khandu Narwane, Shankara Narayanan Krishna, Jean-Vivien Millo and S Ramesh., Sadhana journal, 2018. (Accepted and to appear)

2. *A Cost Effective Approach for Analyzing Software Product Lines*, Ganesh Khandu Narwane, Shankara Narayanan Krishna, Anup Kumar Bhattacharjee, Lecture Notes in Computer Science, 2014, vol 8337. Springer, ISBN 978-3-319-04482-8, 5–22.

3. *Traceability analyses between features and assets in software product lines*, Ganesh Khandu Narwane, José A. Galindo,Shankara Narayanan Krishna, David Benavides, Jean-Vivien Millo and S Ramesh., Entropy Journal, 2016, Volume 18, doi: `http://dx.doi.org/10.3390/e18080269`.

Conferences

1. *Compositional modeling and analysis of automotive feature product lines*, Shankara Narayanan Krishna, Ganesh Khandu Narwane, S. Ramesh, Ashutosh Trivedi, DAC, 2015, doi: `http://doi.acm.org/10.1145/2744769.2747928`, 57:1–57:6.

2. *Compositional Verification of Software Product Lines*, Jean-Vivien Millo, S. Ramesh, Shankara Narayanan Krishna, Ganesh Khandu Narwane, IFM,

2013, doi: http://dx.doi.org/10.1007/978-3-642-38613-8_8, 109–123.

3. *Tracing SPLs precisely and efficiently*, Swarup Mohalik, S. Ramesh, Jean-Vivien Millo, Shankara Narayanan Krishna, Ganesh Khandu Narwane, SPLC, 2012, doi: http://doi.acm.org/10.1145/2362536.2362562, 186–195.

Others

1. *Automated Analysis Operations in Software Product Lines*, Ganesh Khandu Narwane, A. K. Bhattacharjee and Krishna S., SACI, 2014, Symposium on Advances in Control and Instrumentation.

Dedicated to my Family, Teachers and Friends

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my guide Dr. A. K. Bhattacharjee and Prof. Krishna S. for there invaluable guidance, suggestions and encouragement during the course of this work.

I am indebted to Prof. Ashutosh Trivedi, Prof. S. Akshay, Dr. Ramchandra and Dr. Shetal for the discussions and suggestions during the work. I am grateful to my doctoral committee members Dr. Kallol Roy, Dr. Archana Shrama, Dr. Gopika Vinod, Dr. V. H. Patankar and Dr. B. Dikshit for their valuable suggestions.

I thanks my colleagues and friends Amol, Prateek, Ajith, Hrishi, Devendra, Kakasaheb and Chandrakant for their help and support. Finally, I thanks my parents for their love and encouragement.

Ganesh Khandu Narwane

SYNOPSIS

In a Software Product Line (SPL), the central notion of implementability provides the requisite connection between specifications and their implementations, leading to the definition of products. While it appears to be a simple extension of the traceability relation between components and features, it involves several subtle issues that were overlooked in the existing literature. In the current research, a precise and formal definition of implementability over a fairly expressive traceability relation is been introduced. The consequent definition of products in the given SPL naturally entails a set of useful analysis problems that are either refinements of known problems or are completely novel. This proposal also introduce, a new approach to solve these analysis problems by encoding them as Quantified Boolean Formulae (QBF) and solving them through Quantified Satisfiability (QSAT) solvers. QBF can represent more complex analysis operations, which cannot be represented by using propositional formulae. The methodology scales much better than the SAT-based solutions hinted in the literature and were demonstrated through a tool called SPLAnE (SPL Analysis Engine) on a large set of SPL models. Now a days, most of the systems are composed of software-implemented components often interacting with physical subsystems under real-time constraints. The developed components are managed as a Software Product Lines (SPLs) to derive the variants products as per the requirements. Variability is a central to SPL and it is observed that variability is expressed differently at different levels of abstraction during the various phases of development, like requirements, design and implementation. A natural problem in this context is the conformance of variability information expressed at different levels. Unlike many existing approaches to SPL modeling, our work does not assume a single

global view of variation points, even within the same level of abstraction. In our view, an SPL is a concurrent composition of features, where each feature exhibits independent variability. This enables incremental addition of variability. Design variability verification, in particular, checks whether the variability expressed at the design level conforms to that at the requirement level. This work proposes a novel notion called design variability verification applicable to real-time Software Product Lines (SPL). This work introduce and study a formal model of such feature product lines capable of capturing variability and real-time behavior. The notion of conformance to relate the variability at different levels of abstraction and propose a compositional method of verifying conformance of multiple features is define. The procedure is compositional in the sense that the verification of an entire SPL consisting of multiple features is reduced to the verification of the individual features. Feature level verification essentially involves standard model checking while, in the second step, a Quantified Boolean Formula (QBF) is synthesized and solved. The method has been implemented and demonstrated in a tool SPLEnD (SPL Engine for Design Verification) on a couple of fairly large case studies. SPLEnD uses SPIN tool for the feature level conformance, while the state of the art QBF solver CirQit is used for the SPL level conformance. SPLEnD easily handles the evolution of SPL by addition of new features and modification of existing features.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software Product Line Engineering (SPLE) is a software development paradigm supporting the joint design of closely-related software *products* in an efficient and cost-effective manner. The starting point of a Software Product Line (SPL) is the *scope*, which defines all of the possible features of the products in the SPL. The scope is said to define the *problem space* of the SPL, describing the expectations and objectives of the product line. The description is typically organized as a feature model [1] that expresses the variability of the SPL in terms of relations or constraints (exclusion, requires dependency) between the features and defines all of the possible products in the product line.

An important step in SPLE is the development of *core assets*, a collection of reusable artifacts. The core assets contains the components, and we use the term *component* to represent any artifacts that contribute to product development, like code, design, documents, test plan, hardware, etc. A component is an abstract concept of any assets used in products. The core assets define the *solution space* of the SPL and are developed to meet the expectations outlined in the problem

space [2]. They are developed for systematic reuse across the different products in the SPL [3, 18]. The variability in core assets across the components is represented by a component model. The components in a component model may also have exclusions and require dependency constraints, similarly to feature models. Given the problem and solution spaces for an SPL, as defined by the scope and the core assets, the next important step is *traceability*, which involves relating the elements (features, core assets) at these two levels [2].

The focus of this work is formal modeling and analysis of traceability in an SPL. There are many relationships possible; one of the most useful and natural one is the *implementability* relation that associates each feature in the scope with a set of core assets that are required for implementing the feature(s) [4]. The integration of variabilities of the problem space and the solution space using traceability relation is proposed in this work. For example, one could be interested in checking whether every product in the problem space has a correspondence in the solution space, i.e., every product represented in the feature model can be implemented using the existing assets considering the implementability relation. Another example is the property to check every asset of an SPL that needs to be maintained not only because it is involved in some implementations, but the asset is the only option.

Let us consider an example from the cloud computing domain. The company offers a service to rent computers on a cloud with different possible software configurations using Linux-based distributions. In the back-end, instead of providing physical machines, the company provides virtual machines with some software package installed on them. Thus, the configuration of machines can be generated *on demand* according to the needs of the users. In order to improve the speed of

the creation of a new machine, there are pre-configured machines ready to launch.

In this example, the possible configurations offered to the users define the problem space. The set of available Linux packages implementing the features is the core assets. The pre-configured machines can be seen as another set of assets (limited, but available immediately).

The following are some examples of relevant analyses that could arise in this example:

– Check if at least one of the pre-configured machines covers the needs of a new user configuration.

– Check if at least one of the pre-configured machines realizes (exactly) the needs of a new user configuration.

– Check if there are dead packages, i.e., packages that cannot be in any of the virtual machines.

In the literature, formal modeling and analysis of variability at the feature model level have been studied extensively, and several efficient tools have been built to carry out the analyses [5, 19]. The main idea behind all of these works is that the variability analysis can be reduced to constraints and variables modeling the feature level variability [20, 2, 21, 22, 23, 24, 25, 26, 27]. While there are several recent works on traceability, most of them have confined themselves to an informal treatment [28, 29, 30, 31]. Some works have chosen a formal approach for representing the traceability and configuration of features [6].

In the past, most of the work [5, 19] has encoded variability analysis operations in propositional formulae. There are various SAT solvers, like SAT4j [32], or MiniSAT [33], or PicoSAT [34], which can be used to check the satisfiability of

a propositional formula. We propose a novel approach for modeling traceability and other notions relating features and core assets using the Quantified Boolean Formula (QBF). QBF is a generalization of SAT Boolean formulae in which the variables may be universally and existentially quantified. The Quantified Satisfiability (QSAT) solver is used to check the satisfiability of the QBF. In this work, we make use of the well-known QSAT solver, CirQit [35] and RAReQS-NN [36].

The proposed method has been implemented in a tool that is integrated with the FaMa framework [37]. This tool, called SPLAnE (SPL Analysis Engine) [38], can model feature models, core assets (component models) and a traceability relations. SPLAnE is a feasible solution for the automated analysis of feature models together with asset relations.

1.1 Motivating Example

In this report, we present cloud computing as a product line. The feature model and the component model are used to manage the variability across the scope and core assets, respectively.

Feature Models

Feature models have been used to describe the variant and common parts of the product line since Kang [39] has defined them. The sets of possible valid combinations of those features are represented by using different constraints among features. The feature model in Figure 1.1 represents the features provided by the cloud computing. Two different kinds of relationships are used: (i) hierarchical relationships, which describe the options for variation points within the product

line; and (ii) cross-tree constraints that represent constraints among any features of the feature tree. Different notations have been proposed in the literature [5]; however, most of them share the following relationship flavors:

Four different hierarchical relationships are defined:

- *mandatory:* this relationship refers to features that have to be in the product if its parent feature is in the product. Note that a root feature is always mandatory in feature models.

- *optional:* this relationship states that a child feature is an option if its parent feature is included in the product.

- *alternative:* it relates a parent feature and a set of child features. Concretely, it means that exactly one child feature has to be in the product if the parent feature is included.

- *or:* this relationship refers to the selection of at least one feature among a group of child features, having a similar meaning to the logical OR.

Later, two kinds of cross-tree relationships are used:

- *requires:* this relationship implies that if the origin feature is in the product, then the destination feature should be included.

- *excludes:* this relationship between two features implies that, only one of the feature can be present in a product.

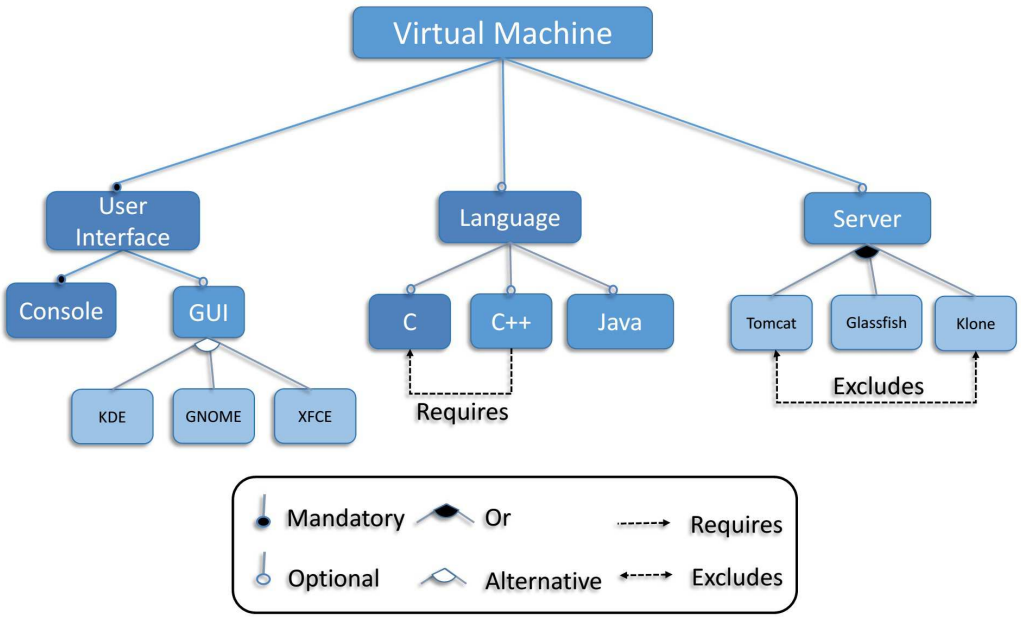

Figure 1.1: Feature model: virtual machine.

Cloud computing technology provides ready to use infrastructure for the clients. The cloud system reduces the cost of maintaining the hardware and software, and also reduces the time to build the infrastructure on the client side. The client pays only for the hardware and software used based on the duration. The feature model for cloud computing is shown in Figure 1.1. The root feature $VirtualMachine$ is a mandatory feature by default. The mandatory relationship is present between the feature $VirtualMachine$ and $UserInterface$, so the feature $UserInterface$ has to be present if feature $VirtualMachine$ is present in the product. The optional relationship is present between the feature *Language* and $VirtualMachine$, so it is optional to have feature *Language* in the products. The feature GUI has alternative relationship with its child features $\{KDE, GNOME, XFCE\}$. Hence, if feature GUI is selected in a product, then only one of its child feature has to be present

in that product. The feature *Server* has *or* relationship with its child features $\{Tomcat, Glassfish, Klone\}$. Hence, if feature *Server* is selected in a product, then at least one of its child feature has to be present in that product. The feature *C++ requires* the feature *C* to be present in a product. The presence of feature *Tomcat* in a product does not allow the feature *Klone* and vice versa. The client can request for a system with the set of features called a $specification$. The minimum set of features in the specification should contain features $\{$ $VirtualMachine$, $UserInterface$, $Console$ $\}$ as they are mandatory features. We can term this as *commonality* across all the products of an SPL. The specification $F = \{$ $VirtualMachine$, $UserInterface$, $Console$, GUI, KDE, $Language$, C $\}$ is valid for the creation of a virtual machine because it satisfies all the constraints in the feature model. The specification $F = \{$ $VirtualMachine$, $UserInterface$, $Console$, GUI, KDE, $Language$, C++$\}$ is not valid because it contains a feature *C++* so it is necessary to select feature *C*.

Component model: Similar to a feature model, same notations can be used to represent variability amongst the components present in *core assets* of an SPL, we call it a Component Model (CM). The variability amongst the components can also be represented by any other models like the Orthogonal Variability Model (OVM), Varied Feature Diagram (VFD) and Free Feature Diagrams (FFDs) [6, 40]. The component model in Figure 1.2 represents the resources available to create a virtual machine. The cloud computing technology will create a virtual machine that contains a set of components required to implement the features present in the client $specification$. Such set of components is called an $implementation$. The implementation $C = \{$*LinuxCore*, *IUser*, *IConsole*, *Terminal*, *ILanguage*, *C-lang*, *c-lib* $\}$ is valid because it satisfies all the constraints on the component

model, so a virtual machine can be created with these components. The implementation $C = \{LinuxCore, IUser, IConsole, ILanguage, C\text{-}lang, c\text{-}lib\}$ is invalid, because the component *Terminal* or *XTerminal* or both are required to satisfy the component model constraints.

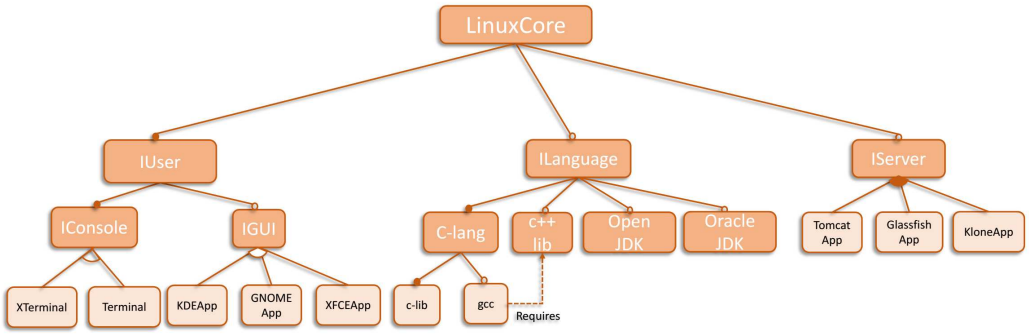

Figure 1.2: Component model: Linux virtual machine-based system.

Table 1.1 shows the traceability relation between the features and the components. The entry in the row of feature *Glassfish* means the component *GlassfishApp* implements the feature *Glassfish*. Similarly, the feature *Console* can be implemented by the set of components: $\{IConsole, XTerminal\}$ or $\{IConsole, Terminal\}$. For each feature in the client specification, the traceability relation gives the required sets of components. In the feature model, the effective features are only the leaf features. The traceability of a parent feature like the feature *GUI* can be implemented by the set of components: $\{IGUI\}$. The feature *GUI* can be abstracted by eliminating all of its child features $\{KDE, GNOME, XFCE\}$; this allows to analyze the SPL at a higher level of abstraction.

Figure 1.3 shows the four preconfigured virtual machines. The preconfigured machines show the set of components from the component model shown in Figure 1.2.

Software product lines can contain a large set of different products. Therefore, facing the complexity of the feature models that represent the products within an SPL is hard. To help in such a difficult task, researchers rely on the computer-aided extraction of information from feature models. This extraction is usually known as the automated analysis in the area. To reason about those models, the relationships existing in the feature model are processed through a CSP (Constraint Satisfaction Problem), SAT, BDD (Binary Decision Diagrams) solver or a specific algorithm. Later, the operation is used to extract specific information from the model. An SPL with twelve leaf features can result in a search space of 2^{12} possible products. Analysis of such a huge search space is a non-trivial task. Some interesting analyses that could performed in this scenario of a Virtual Machine Product Line (VMPL) are as follows:

1. Check if at least one of the pre-configured machines covers the needs of a new user configuration: In VMPL, there is always a need to check the existence of any virtual machine as per the given user specification. For example, the specification F=$\{VirtualMachine, UserInterface, Console, GUI, GNOME\}$ should be first analyzed to check the existence of any implementation that implements F. The implementation C=$\{LinuxCore, IUser, IConsole, Terminal, IGUI, GNOMEApp, IServer, TomcatApp\}$ (equivalent to preconfigured Virtual Machine 2 in Figure 1.3) provides all the features in the specification F, it means that there exists a pre-configured machine which covers the user specification F.

2. Check if at least one of the pre-configured machines realizes (exactly) the needs of a new user configuration: Multiple *implementations* may *cover* a given user specification F. We can analyze the VMPL to find the realized

implementation for the user specification. For example, the implementation C=$\{LinuxCore, IUser, IConsole, Terminal, IGUI, GNOMEApp\}$ (equivalent to preconfigured Virtual Machine 3 in Figure 1.3) exactly provides all the feature in the specification F.

3. Check if there are dead packages: Actual VMPLs contain a huge number of components for Linux systems. The components that are not present in any of the products are termed as dead elements in the product line. In the given VMPL, none of the components is dead.

Feature	Components	Feature	Components
$VirtualMachine$	$\{\{LinuxCore\}\}$	f_1	$\{\{c_1\}\}$
$UserInterface$	$\{\{IUser\}\}$	f_2	$\{\{c_2\}\}$
$Language$	$\{ILanguage\}$	f_8	$\{\{c_{14}\}\}$
$Server$	$\{\{IServer\}\}$	f_{12}	$\{\{c_{10}\}\}$
$Console$	$\{\{IConsole, XTerminal\}, \{IConsole, Terminal\}\}$	f_3	$\{\{c_3, c_4\}, \{c_3, c_5\}\}$
GUI	$\{\{IGUI\}\}$	f_4	$\{\{c_6\}\}$
KDE	$\{\{KDEApp\}\}$	f_5	$\{\{c_7\}\}$
$GNOME$	$\{\{GNOMEApp\}\}$	f_6	$\{\{c_8\}\}$
$XFCE$	$\{\{XFCEApp\}\}$	f_7	$\{\{c_9\}\}$
C	$\{\{C\text{-}lang, c\text{-}lib\}\}$	f_9	$\{\{c_{15}, c_{16}, c_{17}\}\}$
$C++$	$\{\{C\text{-}lang, c\text{-}lib, gcc, c{+}{+}\ lib\}\}$	f_{10}	$\{\{c_{15}, c_{16}, c_{17}, c_{20}\}\}$
$Java$	$\{\{OpenJDK\}, \{OracleJDK\}\}$	f_{11}	$\{\{c_{18}\}, \{c_{19}\}\}$
$Tomcat$	$\{\{TomcatApp\}\}$	f_{13}	$\{\{c_{11}\}\}$
$Glassfish$	$\{\{GlassfishApp\}\}$	f_{14}	$\{\{c_{12}\}\}$
$Klone$	$\{\{KloneApp\}\}$	f_{15}	$\{\{c_{13}\}\}$

Table 1.1: Traceability relation for the virtual machine.

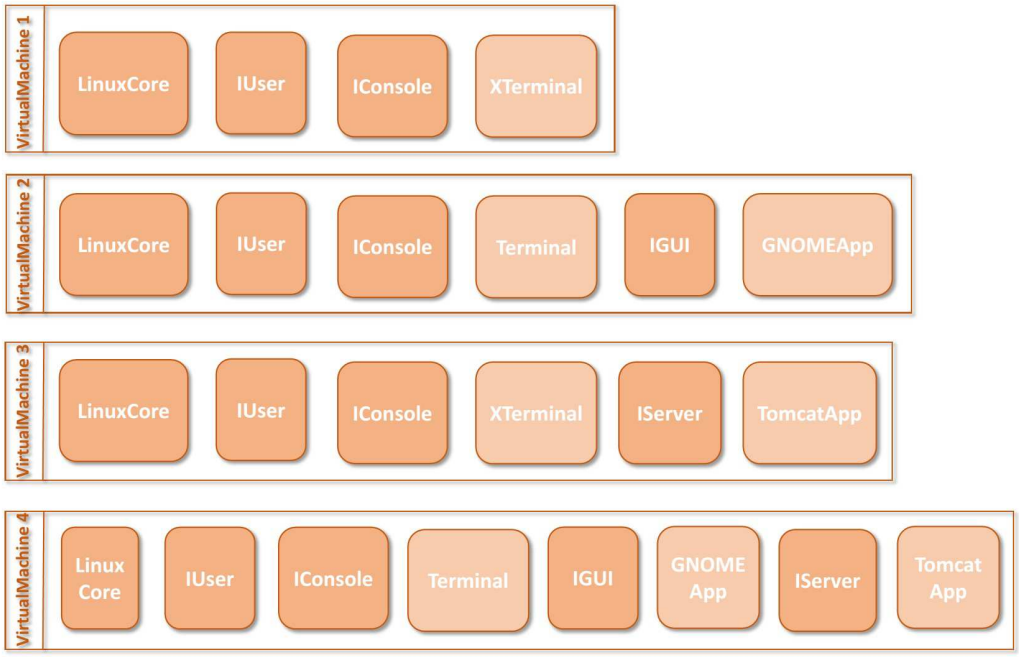

Figure 1.3: Preconfigured virtual machines.

1.1.1 Contribution

The following summarizes the contributions towards formalizing and analyzing SPL models efficiently:

- A simple and abstract set-theoretic formal semantics of SPL with variability and traceability constraints is proposed.
- A number of new analysis problems, useful for relating the features and core assets in an SPL, are described.
- Quantified Boolean Formulae (QBFs) are proposed as a natural and efficient way of modeling these problems. The evidence of the scalability of QSAT

for the analysis problems in large SPLs (compared to SAT) is also provided.

– We present a tool named SPLAnE that enables SPL developers to perform existing operations in the literature over feature diagrams [5] and many new operations proposed in this report. It also allows one to perform analysis operations on a component model and SPL model. We used the FaMa framework to develop SPLAnE that makes it flexible to extend with new analyses of specific needs.

– We experimented on our approach with a number of models, i.e.: (i) real and large Debian models; (ii) randomly-generated SPL models from ten features to twenty thousand features with different levels of cross-tree constraints; and (iii) SPLOT repository models. The experimental results also give the comparison across two QSAT solvers (CirQit and RaReQS) and three SAT solvers (Sat4j, PicoSAT and MiniSAT).

– An example from the *cloud computing* domain is presented to motivate the practical usefulness of the proposed approach.

1.2 SPL with behavioral Features

Large industrial software are often developed as *Software Product Line* (SPL) with a common core set of features which are developed once and reused across all the products of the *family*. The products in an SPL differ on a small set of features which are specified using *variation points*. A central aspect of SPLs is variability modeling and analysis, many early works on SPL modeling focused on this. One of the most prominent models for capturing variability is *feature diagrams* [41].

Feature Diagrams (as well as other variability description formalisms), assume a global view of SPL as they start with a complete list of features and the variation points using a single vocabulary. All the subsequent SPL assets, like requirement documents, design models, source codes, test cases, documentations, share the same definition and vocabulary as presented by [7] and [8]. However, the assumption of a single homogeneous and transverse view of variability description across the entire life cycle of software development seems to impose a waterfall model of development which is limited and inflexible. We give a few instances of the limitations.

First, SPL developers naturally tend to use different representations and vocabulary of variability at different stages of development: at the requirement level, a more abstract and intuitive description of variation points are used, while at the design level, the efficiency of implementation of variation points is of primary concern. For example, consider the case of an automotive SPL, where one variation point is the region of sales (eg. Asia Pacific, Europe, North America, South America and Africa). At the requirement level, this variation point is expressed directly as an enumeration variable assuming one value for every region. Whereas, at the design level, the variation point is expressed using two Boolean variables; by setting the values of the Boolean variables appropriately, the behavior specific to one of the four regions is selected at the time of deployment.

Second, SPLs often evolve during their long lifetime as new features and variabilities are added, removed, or combined during the evolution. A unique and transverse view of the variability requires to maintain the global consistency. A simple change such as combining two variation points into one or splitting one in many requires to propagate this change in every asset impacted at every level

of abstraction (from requirement document to test case) and there is not yet automatic method to perform these changes automatically and consistently. Here one can cite preliminary attempt in this direction by [9].

Third, the end user companies use off-the-shelf components (hardware or software) to implement their features. For example, the Adaptive Cruise Controller (ACC) feature of an automotive SPL can be implemented by either an off-the-shelf ACC component or a combination of off-the-shelf sub-components such as the tracking system (using radar), maximal authorized speed detection system (using GPS map information), and an in-house sub-component depending upon the company specific philosophy. In both cases, there is a distinction (at least in the vocabulary) between the variability as it is expressed in the SPL where company specific philosophy (and keyword) would be respected and the variability offered by the off-the-shelf component that is as generic as possible even though it might be domain specific. The management of this distinction in a single homogeneous and transverse view of the variability would impose a very rigid development methodology.

Fourth, the features are not Boolean anymore thanks to Cordy et. al. [10]. It is not enough to say that a product line of personal computers offers printing facilities. One would like to specify, whether it is $2D$ or $3D$ printing, *black and white* or *color*, $A2$, $A3$, or $A4$. One can add sub-features in the feature diagram but such addition tends to overload the feature diagram and increase the number of cross-tree constraints. Instead, it is natural to consider that a feature has parameters.

In order to overcome the above limitation, we propose a SPL modeling framework that does not start with a global view of variability. The framework allows different vocabularies of variability for different levels of abstraction: the

requirement models can use one vocabulary for expressing variability while the design models can employ a different variability vocabulary. Further, the proposed framework is *compositional* as it allows addition of variabilities as new feature are added. The constraints relating variabilities in different functions can be added along with them. Further, the framework supports both Boolean and non Boolean variabilities with multiple attributes.

One important problem with the use of different variability description at different levels of abstraction is to maintain the consistency across the levels. Consider the automotive SPL described earlier, where one variation point at the requirement level is the region of sale. The same variability is described at the design level as a pair of boolean values. For consistency, we need to check, for example, the behavior of the product when the variation point assumes the value *European market*, at the requirement level, matches with that of the design with the variability expressed at the design level as a pair of boolean values $(true, false)$. In SPL based on a single view of variability, such a mapping is trivial and no special checks are required.

We refer to this problem, generically as *variability verification* and when the consistency of variabilities expressed at the design and requirement levels, we call this *design variability verification*. We present here a formal methods based solution to this problem. This proposed solution is based upon a simple notion of consistency of varaibilities at the design and requirement levels: For every variability at the design level (or loosely, for every product at the design level), there is a varaibility at the requirement level (again loosely, a product at the requiremen level), such that the behavior of the design corresponds to that of the requirement.

For the formalization, the proposed framework extends the standard finite state

machine model, which we call *Finite State Machines with Variability*, in short, *FSMv*. The behavior and variability of a feature at the requirement and design level can be modeled using FSMv. We define a conformance relation between FSMvs to relate the requirement and design models. This relation is based upon the standard language containment of state machines.

One unique feature of FSMv is that it provides a compositional operator for composing the feature state machines. The variability information in a state machine is not derived from a single global variability description and hence incremental addition of features and variabilities is possible using this operator.

One challenge in the variability verification is the analysis complexity: the number of products is exponential in the number of variation points and hence product centric analyses are not scalable. We propose a compositional approach in which every feature of the SPL is first analyzed independently; the per-feature analysis results are then combined to get the analysis result for the whole SPL. The proposed verification approach exploits the compositional structure of the FSMv models to contain the analysis complexity.

The variability verification is to be contrasted with the many recent works on SPL behavior verification by [12] and [13]. The latter works focus on property based verification of SPLs which involve checking whether a property holds for all products in a SPL or if not which products do not satisfy the property. In contrast, the variability verification is concerned with the correspondence of design level variability with that of requirements.

Figure 5.2 summarizes the proposed approach. It shows an SPL composed of features f_1 to f_n. Each feature has an FSMv model of its requirements (called FSMr) and an FSMv model derived from its design (called FSMd). The proposed

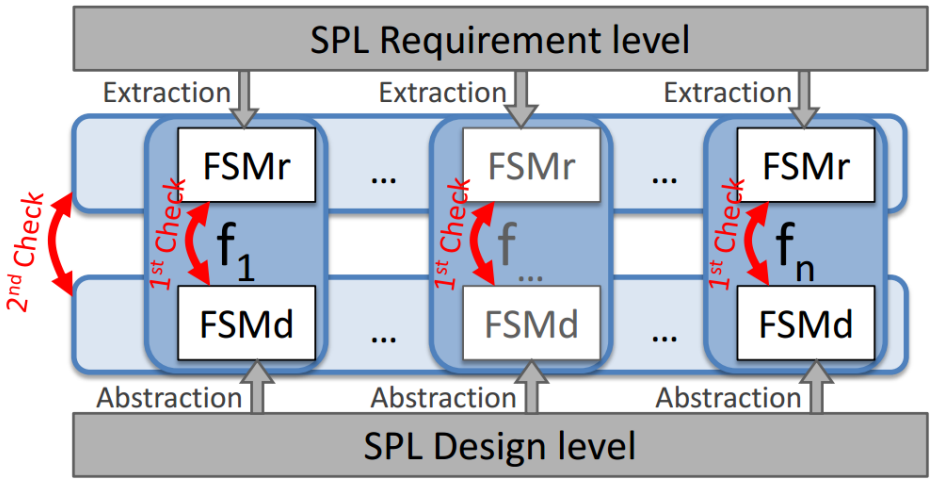

Figure 1.4: The proposed verification approach.

analysis method checks whether the FSMd of every feature conforms to its FSMr (1^{st} check). The output of this first step is a conformance relation Φ_i between each pair of $FSMr_i$ and $FSMd_i$. The obtained conformance relations $\Phi_1, \ldots, \Phi_n$ are then used to check whether the actual behavior of the entire SPL conforms to the expected one (2^{nd} check). The 2^{nd} check is done by synthesizing a Quantified Boolean Formula (QBF) and answering its satisfiability. There is no need to build the entire behavioral model of the SPL in the second step. We have built a prototype tool SPLEnD based upon this approach. This tool performs the first check using SPIN ([14]) while the well-known QBF SAT solver CirQit ([15]) is used for the second step. We have experimented with the tool using modest industrial size examples with very encouraging results.

1.3 SPL with timed behavioral Features

Automotive systems are undoubtedly becoming one of the most complex consumer electronic systems with more and more critical functionality being realized in software. The need for variability among various subclasses of related products adds another dimension to this complexity—it is typical for large automotive manufacturers to build and sell millions of vehicles world-wide with different vehicle types, brands, and options under varying regulatory and legal requirements. In order to cope up with complexity and variability at such scale, automotive manufacturers are turning toward *product line engineering approach* [11]. Product line engineering approach is a framework to develop products family rather than individual products. The key idea is to develop and maintain common requirement and design artifacts for functions and subsystems with well defined variation points and constraints among these variation points. At the time of deployment, the desired variants fulfilling the constraints can be chosen to arrive at a single product. The electronic and software subsystems in a modern vehicle perform a variety of functions which can be abstractly viewed as a collection of *features*. Each feature, typically, is a distinct functionality often visible to the user of the vehicle and may involve repeatedly reading a set of sensors, performing some control computations, and producing some outputs on a set of actuators.

FSMv, however, can not satisfactorily model features expressing real-time constraints or hybrid behavior consisting of interplay between discrete switching with the continuous dynamics of the physical environment. Following Alur-Dill timed automata [16] approach for specifying real-time systems and hybrid automata approach [17] for modeling hybrid behaviors, we introduce generalizations of timed and hybrid automata with variability. We refer to these two models

as timed state machine with variability (TSMv) and hybrid state machine with variability (HSMv). These models can also be considered as FSMv augmented with continuous variables whose dynamics in each state is given via ordinary differential equations. We wish to highlight that while for timed automata a number of verification problems—including reachability—are decidable, for very modest subclasses of hybrid automata simple reachability problems are undecidable. However, there are mature and efficient tools—such as UPPAAL (`www.uppaal.org`) and PHAVER (`http://www-verimag.imag.fr/~frehse/phaver_web/`)—to reason with timed and hybrid automata. Due to space limitation, we introduce TSMv in detail, while only sketch the HSMv model emphasizing the key difference from TSMv.

We introduce a compositional operator to combine multiple features into a feature product line. Each product is a collection of features respecting the given constraints. The features being the atomic objects in any composition, the models capturing them are typically small in the number of discrete/continuous variables and states. Our verification approach is a compositional one: we first verify the design conformance of individual features to their respective requirements. Thanks to the manageable sizes of the TSMv and HSMv at the feature level, this check is relatively economical. We then put together the results obtained at the feature level, and synthesize a quantified boolean formula (QBF) whose satisfiability ensures the conformance of the entire system of feature designs against the requirements.

Chapter 2

Related Works

2.1 Related Work

Automated Analysis of Feature Models: The automated analysis of feature models has been around for more than 20 years [5]. Up to 30 different analysis operations have been presented. However, there is a lack of support for implementation assets and their relations with the variability management. In this report, we extend the variability management analysis with the automated analysis of feature models among the implementation of the different features. White et al. [42] presented an approach to automate the configuration in SPLs by transforming feature model and configurations in Constraint Satisfaction Problem (CSP). The CSP is used to diagnose errors in the selected features. In the case of invalid configuration, it repairs the selected features. Authors also verified their approach on the feature models in the range of 100 to 5000 features. Bagheri et al. shows the approach to construct feature models with its constraints using a propositional formulae [43]. They also explained the formalism to configure a semi-automated

feature model. Soltani et al. gives the configurations process based on artificial intelligence planning technique to derive product from a feature model automatically according to the stakeholders requirements, were the stakeholders may have diverse business and limited resources [44].

Traceability in SPLs: While there is a fairly large body of work in the literature on different facets of SPL, in the following we mention only those which address traceability as a primary aspect. Four important characteristics of a variability model, namely, consistency, visualization, scalability and traceability are defined in [21]. A variability management model that focuses on the traceability aspect of the notion of problem and solution spaces is presented in [2]. Anquetil et al. [20] formalize the traceability relations across the problem and solution space and also across domain and product engineering. In [24], the notion of *product maps* is defined which is a matrix giving the relation between features and products. Consistency analysis of product maps is presented in [25]. Zhu et al. [27] define a traceability relation from requirements to features and also from features to architectures, with consistency analysis. Reference [26] represents a method to identify the traceability between feature model and architecture model. Czarnecki's work [3, 22, 23] on giving semantics to features in feature models by mapping them to other models has been found useful at the requirements level. However, none of the works mentioned above present the formal approach for analyses operations, nor it address the role of traceability in the implementability aspect of SPLs.

Implementation derivation: Borba et al. [45] build on the idea of automatic generation of products from assets by relying on feature diagrams and configuration knowledge (CK) [3]. A CK relates features to assets specifying that as-

sets implement possible feature combinations. The reference [45] lays theoretical foundations on refining and evolving SPLs. The notion of traceability in [45] is general; however, unlike the reference [45], the focus of our report is on the implementability of SPLs.

Template-based traceability: In [22, 23], the authors propose a template-based approach for mapping feature models to annotated models expressed in Unified Modeling Language (UML) or a domain-specific modeling language. Based on a particular configuration of features, an instance of the template is created by evaluating presence conditions in the model. The reference [23] gives a verification procedure which establishes that no ill-formed template instances will be produced given a correct configuration of the feature model. The procedure takes a feature model and an annotated template, which is an instance of a class model (like UML) and a set of OCL (Object Constraint Language) rules. The rules are written with respect to the class model, and each OCL constraint is an invariant on some class c. The final verification is done by checking the validity of a propositional formula. Our notion of traceability is more general than instantiating a template based on the presence of a set of features; moreover, our analysis operations require an encoding into QSAT and we have experimental evidence to suggest that the QSAT encoding performs well over SAT-based procedures (Figure 3.6).

Variability Management: The report that is the closest to our work is that by Metzger et al. [6] and deserves a detailed comparison. In this report, *PL variability* refers to the variations in the features of the system and *software variability* refers to the variations among the software system artifacts. In our report, we follow a different terminology to bring out the product line hierarchy clearly

(shown in Figure 2.1): a scope consists of all the features, a variant specification (referred to as just "specification") is a subset of features, product line specification (*PL specification*) is a set of variant specifications. On the other hand, core assets comprise all the components, a variant implementation (referred to as just "implementation") is a subset of components, product line implementation (*PL implementation*) is a set of variant implementations. The PL variability of Metzger et al. is analogous to PL specifications and software variability is analogous to PL implementations. In Metzger et al., PL variability is represented as OVM (Orthogonal Variability Model) and software variability is represented as FD (Feature Diagrams). In our report, we give set-theoretic semantics to SPLs in lieu of the visually appealing notations such as FD, VFD and OVM. The advantage is that in these semantics the core concepts, analysis problems, and the solution methods can be expressed in a clearer and more concise manner.

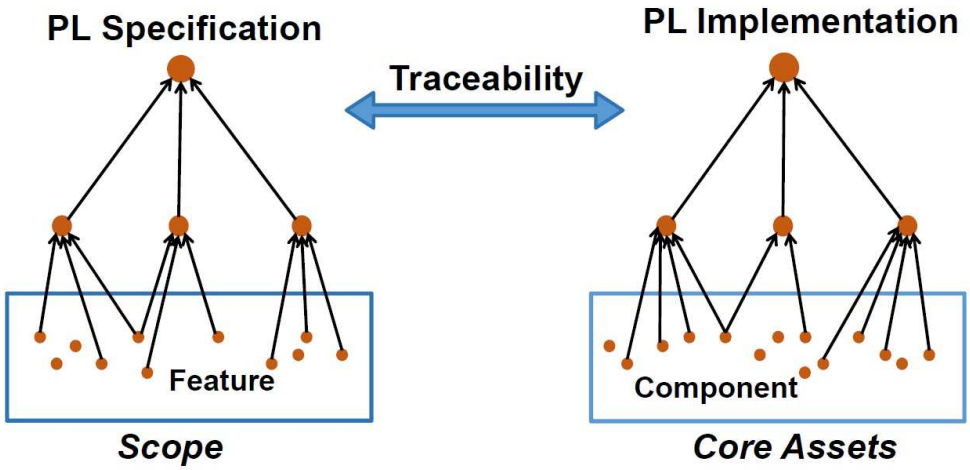

Figure 2.1: Product line hierarchy.

The traceability among PL and software variability is represented in Metzger

et al. using X-links. One type of X-links is of the form $f \Leftrightarrow V_1 \vee V_2 \vee \vee \ldots V_n$ which says a feature f is present iff at least one of the variations V_i is present in the software variability. However, it cannot capture the fact that a feature may be implemented by different sets of software artifacts which may require constraints of the form $f \Leftrightarrow (c_{11} \wedge c_{12} \wedge c_{13}) \vee (c_{21} \wedge c_{22} \ldots) \vee \ldots$. The other type of traceability constraints suggested in Metzger et al. is simple propositional formulae. However, not all propositional constraints provide the intuitive and strong implementability relations between the implementations and specifications. The definition of traceability in our report captures the above-mentioned class of constraints and is used to define a reasonable notion of a relation between implementations and specifications.

Marcilio et al. presented the experimental results to prove the SAT-based approach to analyze a variability models like feature model is easy [46]. Authors have used feature models with maximum of 10,000 features. To increase the hardness, feature models where added with 10%, 20% and 30% of cross-tree constraints (CTC). They found that realistic feature models are not difficult for SAT solvers.

Steven et al. present the heuristic to generate a feature model from an existing system using reverse engineering [47]. They used three real system as Linux, eCos and FreeBSD. The Linux has a variability model represented by Kconfig Language and eCos has Component Definition Language, where as FreeBSD has a list of features. The approach was able to successfully generate the feature model from Linux, eCos and FreeBSD systems.

Mikolás et al. presents a meta model which is mechanically formalized from the feature models in the literature [48]. This meta model is used for feature

modeling and reasoning about it. Larger size SPLs development involves manipulating many parts in FMs. The report [49] propose an compositional approach to develop complex SPLs with the help of complimentary operators like aggregate, merge and slice. Along with reasoning, the report present methods for correction of anomalies, update and extraction and reconciliation of FMs.

The SAT-based definition of products in Metzger et al. allows causally unrelated components and features as products of the SPL. At other times, it is too restrictive in that it does not allow additional components in an implementation which do not provide any feature, but are forced to be with other components because of, say, packaging restrictions. It seems necessary to strike the right balance between the strictness of X-links and the general propositional constraints for a reasonable definition of implementability. This is provided by the definition of the relation *Covers* in our report.

Metzger et al. propose a number of analysis problems; in the terminology of that report, they are *realizability, internal competition, usefulness, flexibility* and *common and dead elements*. We have redefined these in our report from the perspective of the new *implements* relation. Moreover, we have described some new and useful SPL analysis problems (*superfluous, redundancy, critical component, extraneous features*). In Metzger et al., it was noted that the satisfiability-based formulation needed to enumerate and check all the implementations and specifications in order to solve certain analysis problems. Hence, the cumulative complexity of satisfiability checking may be prohibitive for large SPLs. The QSAT based formulation proposed in our report obviates this problem and gives efficient solution methods scalable to large, real-life case studies. Figure 3.6 gives a comparison of SAT and QSAT approaches for the analysis operation *soundness* and

completeness. The time complexity shown in the figure shows the superiority of the QSAT approach over SAT-based approaches for some analysis problems. On a bigger case study (ESPL in Section 3.3), which had 290 features and an equal number of components, the SAT-based approach failed to solve any of the analysis problems.

2.2 Related work with FSMv and TSMv

Feature Based Analysis : [50] explore feature-aware verification to automatically detect feature interactions in a software product line. A language was developed to specify individual features in separate and composable units; based on these feature-local specifications, feature interactions were detected in a product line by either (i) generating all the products and checking them one by one, or (ii) generate one product that contains all the features. The *email* product line with 10 features and 40 products, with 27 feature interactions was checked. [51] presents a programming language oriented approach, that presents a core calculus for feature composition. The features may contain various kinds of software artifacts, like source code in various languages, models and documents. The composition is done uniformly across features with different artifacts in a type-safe way. [52] view features as state machines, and CTL model checking is used to verify properties of individual features. Compositional verification of features is done by checking the consistency of interface labels assigned by the CTL model checking algorithm at the feature level.

Behavioral Conformance : [53] propose the use of modal transition systems (MTS) over labeled transition systems for modeling and analysis of product line architectural behavior. MTS can model optional and required behavior via may and must transitions. A conformance algorithm for MTS is then presented: a fixed point algorithm that computes Cartesian product of states, and eliminates pairs that are invalid according to the relation.

The FTS$^+$ proposed by [12] has some similarities with FSMv, but has a motivational difference. The aim of FTS$^+$ is to model the entire SPL and hence there is a single global machine with a single global vocabulary for expressing variabilities; the variability information represents the presence/absence of features in the SPL. In contrast, our approach is based upon a different view of SPL: a feature with variability is an increment in functionality and an SPL is a collection of features. We use a single FSMv to model a feature and a whole SPL is modeled as a parallel composition of FSMv machines. The difference in viewpoint has another consequence: FTS$^+$ models, since they model the entire SPL, tend to be large and hence has high analysis complexity; some abstraction techniques are hence used ([54]). Whereas, each FSMv models a fraction of functionality and hence can be analyzed easily. [55] use MTS for modeling product behavior and use the logic MHML for model checking. The approaches in [55] as well as [12] use transition systems for expressing system behavior; feature variability constraints are expressed using feature diagrams in [12], while in [55], MHML is used to do this. [12] needs an extra component, a logic for checking properties, while in the case of [55], the MTS+MHML framework is sufficient.

Compositional Verification : [56] propose compositional verification for hierarchical SPLs. Here, Simple Hierarchical Variability Models (SHVM) are used to specify the variability of product artifacts. However, in an SHVM, the number of derivable products is restricted by the fact that there is no means of defining constraints between variation points. [57] uses Event-B composition techniques for feature based product line development. A feature is considered as a basic modular unit in the Rodin tool, and two case studies have been evaluated.

SAT Solving : [58] was the first to propose the use of propositional logic for expressing relationships between requirements in a product line model. Using this, a product line model can be represented as a logical expression; this can be instantiated by the selected requirements. Further, it can be checked if the selected set is valid or not. [59] explores the fundamental connection between feature diagrams, grammars and propositional logic formulas. This connection paved the way for the use of SAT solvers that provide automated support to debug feature models.

Other Approaches and SPL Tools : Many other behavioral models have also been proposed [60, 61, 62, 63] which are usually coupled with a variability model such as OVM [8], the Czarnecki feature model [7], or VPM [64] to attain a fair level of variability expressibility. Unlike all these approaches, FSMv capture the variability in an explicit way which we find more intuitive. The Variation Point Model (VPM) of Hassan Gomaa [64] distinguishes between variability at the requirement and design levels but no design verification approach has been presented. In a recent paper, Jorges *et al.* [65] present a constraint based approach

for variability modeling. Here, architectural as well as behavioral constraints are captured by using temporal logics; synthesis algorithms are then used to compute solutions. Kathrin Berg *et al.* [2] propose a model for variability handling throughout the life cycle of the SPL. Andreas Metzger *et al.* [6] and Riebisch M. *et al.* [31] provide a similar approach but they do not consider the behavioral aspect. In our proposed approach, we extract the relation between requirement and design level variability from a behavioral analysis. [66] present a tool VMC, for the modeling and analysis of product lines. The product family is represented as an MTS, along with extra variability constraints, and all the valid products are automatically generated. The tool implements the algorithm presented in [55]. A demonstration of the main features of VMC can be seen in [67]. Kathi Fisler *et al.* [68] have developed an analysis based on three-valued model checking of automata defined using step-wise refinement. Later on, Jing Liu *et al.* [69] have revisited Fisler's approach to provide a much more efficient method. Recently, Maxime Cordy *et al.* have extended Fisler's approach to LTL formula [70]. Kim Lauenroth *et al.* [71] as well as Andreas Classen *et al.* [12, 54], and Gruler *et al.* [72] have developed model checking methods for SPL behavior. These methods are based on the verification of LTL/CTL/modal μ calculus formula.

All these verification methods assume a global view of variability and hence the representation of variability information is identical in both specification and the design. By contrast, in our work the specification and design involve variability information at different levels of abstraction and hence one needs mapping information between the two levels. Furthermore, our formalism allows incremental addition of functionality and variability and enables compositional verification.

Many other behavioral models have also been proposed [60, 61, 62, 63] which are usually coupled with a variability model such as OVM [8], Czarnecki feature model [7], or VPM [64] to attain a fair level of variability expressible. Unlike all these approaches, FTS$^+$ [12] and FSMv capture in more intuitive way. While our extensions of FSMv handle timed and hybrid systems, the only other product line model known to handle time constraints is the feature timed automata (FTA) introduced in [73]. FTA are an extension of timed automata where the transitions allow featured clock constraints. However, [73] does not use variability in the clock constraints or in the dynamics of the continuous variables unlike TSMv and HSMv.

Chapter 3

SPL Analysis Engine

In this chapter, we give the definitions of traceability and implementation in an SPL. These definitions are illustrated using a virtual machine product line example.

3.1 SPLAnE Framework: Traceability and Implementation

3.1.1 Specification and Implementation

The set of all features found in any of the products in a product line defines the *scope* of the product line. We denote the scope of a product line by $\mathcal{F}$. A scope $\mathcal{F}$ consists of a set of features, denoted by small letters $f_1, f_2 \ldots$. Specifications are subsets of features in the scope and are denoted by $F_1, F_2, \ldots$, with possible subscripts. On the other hand, the collection of components in the product line defines the *core assets* and is denoted as $\mathcal{C}$. Small letters $c_1, c_2 \ldots$, etc. represent

components. Implementations (subsets of components) are denoted by capital letters $C_1, C_2 \ldots$ with possible subscripts. A *Product Line (PL) specification* is a set of *specifications* in an SPL, denoted as $\overline{\mathcal{F}} \in \wp(\ \wp(\ \mathcal{F}\ ) \setminus \{\emptyset\})$. Similarly, a *Product Line (PL) implementation* is denoted as $\overline{\mathcal{C}} \in \wp(\wp(\mathcal{C}) \setminus \{\emptyset\})$. In VMPL, the *scope, core assets, specifications and implementations* are as follows:

- *Scope* $\mathcal{F} = \{f_1 : VirtualMachine,\ f_2 : UserInterface,\ f_3 : Console,\ f_4 : GUI,\ f_5 : KDE,\ f_6 : GNOME,\ f_7 : XFCE,\ f_8 : Language,\ f_9 : C,\ f_{10} : C{+}{+},\ f_{11} : Java,\ f_{12} : Server,\ f_{13} : Tomcat,\ f_{14} : Glassfish,\ f_{15} : Klone\ \}$

- *Core Assets* $\mathcal{C} = \{c_1 : LinuxCore,\ c_2 : IUser,\ c_3 : IConsole,\ c_4 : XTerminal,\ c_5 : Terminal,\ c_6 : IGUI,\ c_7 : KDEApp,\ c_8 : GNOMEApp,\ c_9 : XFCEApp,\ c_{10} : IServer,\ c_{11} : TomcatApp,\ c_{12} : GlassfishApp,\ c_{13} : KloneApp,\ c_{14} : ILanguage,\ c_{15} : C\text{-}lang,\ c_{16} : c\text{-}lib,\ c_{17} : gcc,\ c_{18} : OpenJDK,\ c_{19} : OracleJDK,\ c_{20} : c{+}{+}\ lib\ \}$

- *PL Specification* $\overline{\mathcal{F}} = \{$
  $F_1 : \{VirtualMachine, UserInterface, Console\ \}$ or $\{f_1, f_2, f_3\}$,
  $F_2 : \{VirtualMachine, UserInterface, Console, GUI, KDE\ \}$ or $\{f_1, f_2, f_3, f_4, f_5\ \}$,
  $F_3 : \{VirtualMachine, UserInterface, Console, Server, Tomcat, Glassfish\ \}$ or $\{f_1, f_2, f_3, f_{12}, f_{13}, f_{14}\ \}$,
  $F_4 : \{VirtualMachine, UserInterface, Console, GUI, KDE, Server, Tomcat\ \}$ or $\{f_1, f_2, f_3, f_4, f_5, f_{12}, f_{13}\ \}$,
  $F_5 : \{VirtualMachine, UserInterface, Console, GUI, KDE, Language, Java, Server, Tomcat\ \}$ or $\{f_1, f_2, f_3, f_4, f_5, f_8, f_{11}, f_{12}, f_{13}\ \}$
  $\}$, where F_1, F_2, F_3, F_4 and F_5 are some specifications.

– *PL Implementation* $\overline{C}$={

$C_1 : \{LinuxCore, IUser, IConsole, XTerminal\}$ or $\{c_1, c_2, c_3, c_4\}$,

$C_2 : \{LinuxCore, IUser, IConsole, Terminal\}$ or $\{c_1, c_2, c_3, c_5\}$,

$C_3 : \{LinuxCore, IUser, IConsole, Terminal, IGUI, KDEApp\}$ or $\{c_1, c_2, c_3, c_5, c_6, c_7\}$,

$C_4 : \{LinuxCore, IUser, IConsole, Terminal, IServer, TomcatApp, GlassfishApp\}$ or $\{c_1, c_2, c_3, c_5, c_{10}, c_{11}, c_{12}\}$,

$C_5 : \{LinuxCore, IUser, IConsole, Terminal, IGUI, KDEApp, IServer, KloneApp, Glassfish\}$ or $\{c_1, c_2, c_3, c_5, c_6, c_7, c_{10}, c_{12}, c_{13}\}$,

$C_6 : \{LinuxCore, IUser, IConsole, Terminal, IGUI, KDEApp, IServer, TomcatApp, Glassfish\}$ or $\{c_1, c_2, c_3, c_5, c_6, c_7, c_{10}, c_{11}, c_{12}\}$,

$C_7 : \{LinuxCore, IUser, IConsole, Terminal, IGUI, KDEApp, ILang, OpenJDK, IServer, TomcatApp\}$ or $\{c_1, c_2, c_3, c_5, c_6, c_7, c_{14}, c_{18}, c_{10}, c_{11}\}$

}, where C_1 to C_7 are some implementations.

3.1.2 Traceability

We present a formalism for two variation of traceability relation: (i) $1 : M$ mapping and (ii) $N : M$ mapping. In *traceability* relation, $1 : M$ mapping is between a feature and a set of component sets, where as $N : M$ is a mapping between feature set and a set of component sets.

Traceability with $1 : M$ *mapping:* A feature is implemented using a set of non-empty subset of components in the core asset $\mathcal{C}$. This relationship is modeled by the partial function $\mathcal{T} : \mathcal{F} \rightarrow \wp(\wp(\mathcal{C}) \setminus \{\emptyset\})$. When $\mathcal{T}(f) = \{C_1, C_2, C_3\}$, we interpret it as the fact that the set of components C_1 (also, C_2 and C_3) can implement

the feature f. When $\mathcal{T}(f)$ is not defined, it denotes that the feature f does not have any components to implement it.

Traceability with N : M mapping: A set of features can be implemented using a set of non-empty subset of components in the core asset $\mathcal{C}$. This relationship is modeled by the partial function $\mathcal{T} : (\wp(\mathcal{F}) \setminus \{\emptyset\}) \rightarrow \wp(\wp(\mathcal{C}) \setminus \{\emptyset\})$. It may happen that, two features f_1 and f_2 can be implemented by a single component c_1. In this case, $\mathcal{T}(\{f_1, f_2\}) = \{C_1\}$, where $C_1 = \{c_1\}$.

Definition 1 (SPL)**.** *An SPL Ψ is defined as a triple $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, where $\overline{\mathcal{F}} \in \wp(\wp(\mathcal{F}) \setminus \{\emptyset\})$ is the PL specification, $\overline{\mathcal{C}} \in \wp(\wp(\mathcal{C}) \setminus \{\emptyset\})$ is the PL implementation and $\mathcal{T}$ is the traceability relation.*

3.1.3 The Implements Relation

A feature is *implemented* by a set of components C, denoted $implements(C, f)$, if C includes a non-empty subset of components C' such that $C' \in \mathcal{T}(f)$. It is obvious from the definition that if $\mathcal{T}(f) = \emptyset$, then f is not implemented by any set of components. In VMPL, f_5 is implemented by implementations C_3, C_5, C_6 and C_7, but not by implementations C_1, C_2 and C_4.

We define the formula to check the feature is traceabile as, $formula\_\mathcal{T}(f) = \bigvee_{j=1..k} \bigwedge_{c_i \in C_j} p_{c_i} = \bigvee_{j=1..k} \bigwedge_{c_i \in C_j} w_i$, and one of the disjuncts C_j is TRUE, the entire expression for $formula\_\mathcal{T}(f)$ evaluates to TRUE.

The formula $form\_implements_f(x_1, \ldots, x_n)$ which takes n Boolean values (0 or 1) as arguments, corresponding to the bit vector $\bar{C}$ of an implementation C and evaluates to either TRUE or FALSE.

$$form\_implements_f(x_1,\ldots,x_n) = \forall p_{c_1}\ldots p_{c_n}\,\{[\bigwedge_{i=1}^{n}(x_i \Rightarrow p_{c_i})] \Rightarrow formula\_\mathcal{T}(f)\}$$

In order to extend the definition to specifications and implementations, we define a function $Provided\_by(C)$ which computes all the features that are implemented by C: $Provided\_by(C) = \{f \in \mathcal{F} | implements(C,f)\}$.
In VMPL, $Provided\_by(C_1) = \{f_1, f_2, f_3\}$ and $Provided\_by(C_3) = \{f_1, f_2, f_3, f_4, f_5\}$. With the basic definitions above, we can now define when an implementation exactly implements a specification.

Definition 2 (Realizes)**.** *Given $C \in \overline{\mathcal{C}}$ and $F \in \overline{\mathcal{F}}$, $Realizes(C,F)$ if $F = Provided\_by(C)$.*

The *realizes* definition given above is rather strict. Thus, in the above example, the implementation C_3 realizes the specification F_2, but it does not realize F_1 even though it provides an implementation of all the features in F_1. In many real-life use-cases, due to the constraints on packaging of components, the exactness may be restrictive. We relax the definition of *Realizes* in the following.

Definition 3 (Covers)**.** *Given $C \in \overline{\mathcal{C}}$ and $F \in \overline{\mathcal{F}}$, $Covers(C,F)$ if $F \subseteq Provided\_by(C)$ and $Provided\_by(C) \in \overline{\mathcal{F}}$.*

The additional condition ($Provided\_by(C) \in \mathcal{F}$) is added to address a tricky issue introduced by the *Covers* definition. Suppose the scope $\mathcal{F}$ consisted of only two specifications $\{f_1\}$ and $\{f_2\}$. Let's say that the two variants (f_1 and f_2) are mutually exclusive features. The implementation $C = \{c_1, c_2\}$ implements the feature f_1, assuming $\mathcal{T}(f_1) = \{\{c_1\}\}$ and $\mathcal{T}(f_2) = \{\{c_2\}\}$. Without the provision, we would have $Covers(C, \{f_1\})$. However, since $Provided\_by(C) = \{f_1, f_2\}$, it actually implements both the features together, thus violating the requirement of mutual exclusion. In the VMPL, the implementation C_6 covers the specifications

F_1, F_2, F_3 and F_4. The set of products of the SPL is now defined as the specifications and the implementations covering them through the traceability relation.

Definition 4 (SPL Products)**.** *Given an SPL $\Psi = \langle \overline{\mathcal{F}}, \overline{C}, \mathcal{T} \rangle$, the products of the SPL, denoted by a function $Prod(\Psi)$ which generate a set of all specification-implementation pairs $\langle F, C \rangle$ where $Covers(C, F)$.*

Thus, in the VMPL example, we see that there are many potential products. Valid products are $\langle C_1, F_1 \rangle, \langle C_2, F_1 \rangle$, $\langle C_3, F_1 \rangle$, $\langle C_3, F_2 \rangle$, $\langle C_4, F_1 \rangle$, $\langle C_4, F_3 \rangle$ ....

Lemma 1. *(Implements) Given an SPL, a set of components C, and a feature f, $implements(C, f)$ iff $form\_implements_f(v_1, \ldots, v_n)$, where $\bar{C} = \langle v_1, \ldots, v_n \rangle$, evaluates to TRUE.*

Proof. ($\Rightarrow$) : Let $\mathcal{T}(f) = \{C_1, \ldots, C_k\}$ and assume $implements(C, f)$. By definition, there is a $C_j \in \mathcal{T}(f)$ such that $C_j \subseteq C$. Let $C_j = \{p_{c_{\ell_1}}, \ldots, p_{c_{\ell_m}}\}$. Then, $v_{\ell_i} = 1$ for all $i = 1, \ldots, m$. We have to show that

$$\forall p_{c_1} \ldots p_{c_n} \{ [\bigwedge_{i=1}^{n} (v_i \Rightarrow p_{c_i})] \Rightarrow formula\_\mathcal{T}(f) \}.$$

Let $\langle w_1, \ldots, w_n \rangle$ be an assignment of boolean values (TRUE or FALSE) to the propositional variables $p_{c_1} \ldots p_{c_n}$ such that $\bigwedge_{i=1}^{n} (v_i \Rightarrow p_{c_i}) = \bigwedge_{i=1}^{n} (v_i \Rightarrow w_i)$ evaluates to TRUE. Since $v_{\ell_i} = 1$ for all $i = 1, \ldots, m$, this implies $w_{\ell_j} = 1$ for all $j = 1, \ldots, m$ as well. Therefore, for $C_j \in \mathcal{T}(f)$, $(\bigwedge_{c_i \in C_j} w_i) = w_{\ell_1} \wedge \cdots \wedge w_{\ell_m} =$ TRUE.

Assume that $implements(C, f)$ does not hold. Then, for every $C_j \in \mathcal{T}(f)$, $C_j \not\subseteq C$. This implies that for all $j \in \{1 \ldots k\}$, there is a $c_j \in C_j \setminus C$. Define an assignment to the propositional variables as follows: $v_i = 1$ for p_{c_i} such that $c_i \in C$ and 0 for the rest. Hence, $v_j = 0$ for the proposition p_{c_j} corresponding to component $c_j \notin C$.

This assignment evaluates the antecedent $\bigwedge_{i=1}^{n}(v_i \Rightarrow p_{c_i})$ to TRUE. But the consequent $formula\_\mathcal{T}(f) = \bigvee_{j=1..k} \bigwedge_{c_i \in C_j} p_{c_i}$ = FALSE for the above assignment because each disjunct is falsified by the presence of an assignment $v_j = 0$ for the proposition $p_{c_j} \notin C$. Therefore, $form\_implements_f(v_1, \ldots, v_n)$ evaluates to FALSE. □

Lemma 2. *(Realizes, Covers) Given a set of components C and a set of features F, let* $\bar{C} = (c'_1, \ldots, c'_n)$ *and* $\bar{F} = (f'_1, \ldots, f'_m)$. *Then the following statements hold:*

1. *C covers F iff* $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$
2. *C realizes F iff* $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$

Proof. Suppose $\bar{C} = (c'_1, \ldots, c'_n)$ and $\bar{F} = (f'_1, \ldots, f'_m)$. If C covers F, then $Provided\_by(C) = \{f \mid Implements(C, f)\}$ will contain F. Pick $f \in F$. Since $F \subseteq Provided\_by(C)$, $\exists\, C_1 \in \mathcal{T}(f), C_1 \subseteq C$. We then have $\widehat{C} \Rightarrow \widehat{C}_1$ and $\widehat{C}_1 \Rightarrow formula\_\mathcal{T}(f)$. Therefore, $form\_implements_f(c'_1, \ldots, c'_n)$ holds. Hence, it will be the case that $p_{f'_i} \Rightarrow form\_implements_{f_i}(c'_1, \ldots, c_n)$ for every $f'_i \in \bar{F}$. Hence, $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$.

Conversely, suppose $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$. Then $\bigwedge_{i=1}^{m}(p_{f'_i} \Rightarrow form\_implements_{f_i}(c'_1, \ldots, c'_n))$ holds. If $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ holds, then (1) either $p_{f'_j} = 0$, or (2) $p_{f'_j} = 1$ and $form\_implements_{f_j}(c'_1, \ldots, c'_n)$ holds. As seen in Lemma 1, $form\_implements_{f_i}(c'_1, \ldots, c'_n)$ holds good when there exists a set $C_{j_i} \in \mathcal{T}(f_i)$ such that $C_{j_i} \subseteq C$. Since this is true for all f_i, we have $\bigcup_{j_i} C_{j_i} \subseteq C$. Therefore, $Provided\_by(C) \supseteq Provided\_by(\bigcup_{j_i} C_{j_i})$. By the formula $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$, whenever $p_{f'_i} = 1$, there exists $C_{j_i} \subseteq C$ which implements f_i. Therefore, C implements possibly a superset of F, hence covers F.

The proof for realizes is similar. The only difference is that the implication is both ways, which ensures that C implements F exactly. □

Lemma 3. *(Completeness, Soundness) Let* $\Psi = (\overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T})$ *be an SPL, with* $\mathcal{F} = \{f_1, \ldots, f_m\}$ *and* $\mathcal{C} = \{c_1, \ldots, c_n\}$.

1. Ψ *is complete iff*

   $\forall f'_1 \ldots f'_m[CON_F(f'_1, \ldots, f'_m) \Rightarrow \exists\, c'_1 \ldots c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]]$

2. Ψ *is sound iff*

   $\forall\, c_1 \ldots c_n[CON_I(c_1, \ldots, c_k) \Rightarrow \exists\, f_1 \ldots f_j[CON_F(f_1, \ldots, f_j) \wedge f\_covers(c_1, \ldots, c_k, f_1, \ldots, f_j)]]$

Proof. Let $\Psi = (\overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T})$ be an SPL with n components and m features. Let $\overline{\mathcal{C}} = \{S_1, \ldots, S_k\}$. Given a tuple of component parameters $c'_1, \ldots, c'_n$ where each c'_i is 0 or 1, the predicate $CON_I(c'_1, \ldots, c'_n)$ is defined as

$$\bigvee_j \bigwedge_{c_i \in S_j} c'_i$$

Then $CON_I(c'_1, \ldots, c'_n)$ is satisfied iff $\{c'_k \mid c'_k = 1\} = S_l$ for some $S_l \in \overline{\mathcal{C}}$. $CON_F(f'_1, \ldots, f'_m)$ is defined similarly.

1. Assume that the SPL is complete. Then for every $F \in \overline{\mathcal{F}}$, there exists some $C \in \overline{\mathcal{C}}$ such that $Covers(C, F)$. Pick any $F \in \overline{\mathcal{F}}$ and its corresponding $C \in \overline{\mathcal{C}}$. Let $\bar{F} = (f'_1, \ldots, f'_m)$ and $\bar{C} = (c'_1, \ldots, c'_n)$. Then $CON_F(f'_1, \ldots, f'_m)$ will be 1 iff there exists a set $F_j \in \overline{\mathcal{F}}$ such that $\bar{F}_j(i) = 1$ iff $f'_i = 1$. Since $Covers(C, F)$ holds for all $F \in \overline{\mathcal{F}}$, we have by Lemma 2, $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ is true for every tuple $(f'_1, \ldots, f'_m)$ such that $f'_i = 1$ iff $\exists F_j \in \overline{\mathcal{F}}$ such that $\bar{F}_j(i) =$

1. That is, for every tuple $(f'_1, \ldots, f'_m)$ that satisfies the feature constraints, there exists some tuple $(c'_1, \ldots, c'_n)$ such that $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ holds. Thus,
$\forall f'_1 \ldots f'_m.\{[CON_F(f'_1, \ldots, f'_m) \Rightarrow \exists\, c'_1 \ldots c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1,$ $\ldots, c'_n, f'_1, \ldots, f'_m)]\}$ holds.

   Conversely, assume $\forall f'_1 \ldots f'_m.\{[CON_F(f'_1, \ldots, f'_m) \Rightarrow \exists\, c'_1 \ldots c'_n[CON_I(c'_1,$ $\ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]\}$ holds. Then for all possible ways of satisfying $CON_F(f'_1, \ldots, f'_n)$ (i.e, over all $F \in \overline{\mathcal{F}}$), there exists some tuple satisfying $CON_I(c'_1, \ldots, c'_n)$ such that $f\_covers(c'_1, \ldots, c'_n, f'_1,$ $\ldots, f'_m)$. Each tuple satisfying $CON_F(f'_1, \ldots, f'_n)$ corresponds to a set in $\overline{\mathcal{F}}$. Corresponding to each such set, there is a tuple $(c'_1, \ldots, c'_n)$ satisfying $[CON_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$: that is, there is some $C \in \overline{C}$ with $\bar{C} = (c'_1, \ldots, c'_n)$ such that $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$. This says that for every $F \in \overline{\mathcal{F}}$, there exists some $C \in \overline{C}$ such that C covers F. Hence, Ψ is complete.

2. The proof of soundness is similar.

□

Lemma 4. *(Existentially Explicit Features) Given a set of features F, let $\bar{F} = (f'_1, \ldots, f'_m)$. Then F is existentially explicit iff*
$\exists c'_1 \ldots c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$.

Proof. Suppose F is existentially explicit. Then there exists a $C \in \overline{C}$ such that C realizes F. By Lemma 2, C realizes F iff $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$. $CON_I(c'_1, \ldots, c'_n)$ is true iff there exists a $C \in \overline{C}$ such that $\bar{C} = (c'_1, \ldots, c'_n)$.

Hence, if F is existentially explicit, $\exists c'_1, \ldots, c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$ holds.

Conversely, assume $\exists c'_1, \ldots, c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$ holds. Then, there exists a $C \in \overline{C}$ with $\bar{C} = (c'_1, \ldots, c'_n)$ and $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$. Again, by Lemma 2, $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ iff C realizes F, with $\bar{C} = (c'_1, \ldots, c'_n)$, $\bar{F} = (f'_1, \ldots, f'_m)$. That is, for $F \in \overline{\mathcal{F}}$, there exists a $C \in \overline{C}$ such that C covers F. Hence, F is existentially explicit. □

Lemma 5. *(Universally Explicit Features) Given a set of features F, let $\bar{F} = (f'_1, \ldots, f'_m)$. Then F is universally explicit iff φ_F holds, where φ_F is given by*

$\exists c'_1 \ldots c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \wedge$

$\forall c'_1 \ldots c'_n\{[(CON_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)\}$.

Proof. Assume F is universally explicit. Then by definition, (i) there exists a $C \in \overline{C}$ such that C realizes F and (ii) for all $C \in \overline{C}$, C covers $F \Rightarrow C$ realizes F.

The first point (i) can be expressed as $\exists c'_1 \ldots c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$ (recall that $CON_I(c'_1, \ldots, c'_n)$ holds iff there exists a $C \in \overline{C}$ with $\bar{C} = (c'_1, \ldots, c'_n)$, and $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ holds iff C realizes F by Lemma 2).

To formalize the second point (ii), we have to consider all possible component tuples $(c'_1, \ldots, c'_n)$ satisfying $CON_I(c'_1, \ldots, c'_n)$, such that $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ holds. For each such tuple, we have to ensure that $f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ holds. This is true iff $\forall c'_1 \ldots c'_n\{[(CON_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)\}$ holds. Clearly, if F is universally explicit, then φ_F holds.

Conversely, assume φ_F holds. Now, $\exists\, c'_1 \ldots c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_realizes($

$c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$] holds whenever there is a tuple $(c'_1, \ldots, c'_n)$ satisfying $CON_I(c'_1, \ldots, c'_n)$ for which $f\_realizes(\ c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ is true. This corresponds to a set in $C \in \overline{C}$ with $\bar{C} = (c'_1, \ldots, c'_n)$ which realizes F. $\forall c'_1 \ldots c'_n\{[(CON_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow f\_realizes(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)\}$ considers all possible tuples $(c'_1, \ldots, c'_n)$ satisfying $CON_I(c'_1, \ldots, c'_n)$ for which, whenever $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ is true, so is $f\_realizes(\ c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$. By definition, each tuple satisfying $CON_I(c'_1, \ldots, c'_n)$ corresponds to a set $C \in \overline{C}$. The formulae holds iff for each such $C \in \overline{C}$, whenever C covers F, C realizes F. Therefore, F is universally explicit whenever φ_F holds. □

Lemma 6. *(Unique Implementation) Given a set of features F, let $\bar{F} = (f'_1, \ldots, f'_m)$. Then F has a unique implementation iff φ_U holds. φ_U is given by*

$$\exists c'_1 \ldots c'_n[CON_I(c'_1, \ldots, c'_n) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \wedge$$
$$\forall d'_1 \ldots d'_n\{[CON_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)] \Rightarrow (\wedge_{i=1}^{n}(d'_i \Leftrightarrow c'_i)\}$$

Proof. Let F have a unique implementation. Then there exists a $C \in \overline{C}$ which covers F and for all $C' \in \overline{C}$ which covers F, $C = C'$. Two implementations C, C' are same when $\bar{C} = \bar{C}'$. That is, $\bar{C}(i) = \bar{C}'(i)$ for all $1 \leq i \leq n$. As given by the definition of $CON_I(c'_1, \ldots, c'_n)$, $CON_I(c'_1, \ldots, c'_n)$ is satisfiable iff there exists some $C \in \overline{C}$ with $\bar{C} = (c'_1, \ldots, c'_n)$. Such a C covers F iff $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ as given by Lemma 2. We have to check that there is a unique $C \in \overline{C}$ that can cover F - for this, we enumerate over all possible tuples $(d'_1, \ldots, d'_n)$ that satisfy $CON_I(d'_1, \ldots, d'_n)$, and then ensure that $(d'_1, \ldots, d'_n) = (c'_1, \ldots, c'_n)$. This check is given by $\forall d'_1 \ \ldots \ d'_n\{[CON_I(d'_1, \ \ldots, \ d'_n) \wedge f\_covers(d'_1, \ \ldots, \ d'_n, \ f'_1, \ \ldots, \ f'_m)] \Rightarrow (\wedge_{i=1}^{n} \ (d'_i \Leftrightarrow c'_i)\}$ Thus, if F has a unique implementation, we have φ_U holds.

The converse is similar. □

Lemma 7. *(Common, live and dead elements)*

1. *A component c is common iff*
   $\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow p_c\}$ *holds.*

2. *A component c is live iff*
   $\exists\, c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m) \wedge p_c\}$

3. *A component c is dead iff*
   $\forall\, c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow \neg p_c\}$ *holds.*

Proof. 1. Assume that c is a common component. Then by definition, for all $\langle F,C\rangle \in Prod(\Psi)$, $c \in C$. To enumerate all possible $\langle F,C\rangle$ for $F \in \overline{\mathcal{F}}$ and $C \in \overline{C}$, we consider all possible tuples $(c'_1,\ldots,c'_n)$ as well as $(f'_1,\ldots,f'_m)$ for which $CON_I(c'_1,\ldots,c'_n) \wedge CON_F(f'_1,\ldots,f'_m)$ holds. Clearly, every pair of tuples satisfying $CON_I(c'_1,\ldots,c'_n) \wedge CON_F(f'_1,\ldots,f'_m)$ corresponds to a pair $\langle F,C\rangle$. For each such pair $\langle F,C\rangle$ of tuples to be in $Prod(\Psi)$, we check if C covers F. By Lemma 2, this holds iff $f\_covers(c'_1,\ldots,c'_n,f'_1,\ldots,f'_m)$. Clearly, for all pairs of tuples for which this is true, if an element is common, then it will evaluate to 1. This is given by saying $\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow p_c\}$. Note that in $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$, we are evaluating over all possible values of $p_{c_1}, \ldots, p_{c_n}$. In particular, for $c = c_i$, we are checking that whenever $c'_i = 1$ in a product, then $p_{c_i} = 1$.

The converse is similar.

2. Assume c is live. Then there is a pair $\langle F,C\rangle \in Prod(\Psi)$ such that $c \in C$. The existence of a pair $\langle F, C\rangle \in Prod(\Psi)$ is expressed by saying $\exists\, c'_1, \ldots, c'_n, f'_1, \ldots, f'_m[CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]$. Clearly, if c is in one such tuple, $p_c = 1$. This is written by conjuncting p_c and obtaining $\exists c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m) \wedge p_c\}$. The converse is similar.

3. This is similar to 1.

□

Lemma 8. *(Superflous) A component c_i is superflous iff* $\forall\, c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[c'_i \wedge CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_i, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow \exists\, d'_1, \ldots, d'_n[\neg d'_i \wedge CON_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)]\}$.

Proof. Assume c_i is superflous. Then by definition, for all $C \in \overline{C}$ containing c_i and which covers F, there exists $C' \in \overline{C}$ which does not contain c_i and which covers F. First consider all $C \in \overline{C}$ containing c_i which covers F. This is given by considering all tuples $(c'_1, \ldots, c'_n)$ and $(f'_1, \ldots, f'_m)$ which satisfy $CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m)$, $c'_i = 1$, for which $f\_covers(c'_1, \ldots, c'_i, \ldots, c'_n, f'_1, \ldots, f'_m)$ holds. The pairs $\langle C, F\rangle$ are enumerated by considering all tuples satisfying $CON_I(c'_1, \ldots, c'_i, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m)$, and for those in $Prod(\Psi)$, we need to check that C covers F. Now, if such a C contains c_i, then $c'_i = 1$ in the tuple $(c'_1, \ldots, c'_n)$. To check if there exists a pair in $Prod(\Psi)$ which does not contain the ith component c_i, among all tuples $(c'_1, \ldots, c'_n)$, we check if there exists a tuple $(d'_1, \ldots, d'_n)$ such that $CON_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)$ holds and where $d'_i = 0$. This is expressed by $\neg d'_i$. Thus, if c_i is superflous, we have $\forall\, c'_1, \ldots,$

$c'_n, f'_1, \ldots, f'_m\{[c'_i \wedge CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow \exists\, d'_1, \ldots, d'_n[\neg d'_i \wedge CON_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)]\}$ holds.

The converse is similar. □

Lemma 9. *(Redundant) A component c_i is redundant iff*
$\forall\, c'_1, \ldots, c'_n\, f'_1 \ldots, f'_m\{[c'_i \wedge CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow f\_covers(c'_1, \ldots, \neg c'_i, \ldots, c'_n, f'_1, \ldots, f'_m)\}$

Proof. Suppose c_i is redundant. Then for every $C \in \overline{C}$ containing c_i, there exists a $C' \in \overline{C}$, $C' \subseteq C$, $c_i \notin C'$, and $Provided\_by(C) = Provided\_by(C')$. First, we have to enumerate all tuples $(c'_1, \ldots, c'_n)$ for which we have $CON_I(c'_1, \ldots, c'_n)$ and c'_i (i.e, enumerate all members of $\overline{C}$ containing c_i). Now, we have to look at the sets of features F that these implementations cover - by definition of covers, this means that the set $Provided\_by(C)$ is in $\overline{\mathcal{F}}$, and $F \subseteq Provided\_by(C)$. This basically means to look at all tuples $(f'_1, \ldots, f'_m)$ such that $CON_F(f'_1, \ldots, f'_m)$ (which correspond to some element of $\overline{\mathcal{F}}$) and $f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)$ (which are covered by C). This is expressed by specifying $\forall c'_1, \ldots, c'_n, f'_1, \ldots, f'_m\{[c'_i \wedge CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)]\}$. For each such C covering F, we want to say that there exists a $C' \subseteq C$ which covers F and which does not contain c'_i. This is expressed by saying that there exists a tuple $(d'_1, \ldots, d'_n)$ for which (i) $CON_I(d'_1, \ldots, d'_n)$ holds, (ii) $f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)$ holds, (iii) $\neg c'_i$ holds, and $(iv)(\bigwedge_{i=1}^{n} c'_i \Rightarrow \bigwedge_{i=1}^{n} d'_i)$. This last condition checks that $C' \subseteq C$. Thus, the required formula to hold is $\forall\, c'_1, \ldots, c'_n\, f'_1 \ldots, f'_m\{[c'_i \wedge CON_I(c'_1, \ldots, c'_n) \wedge CON_F(f'_1, \ldots, f'_m) \wedge f\_covers(c'_1, \ldots, c'_n, f'_1, \ldots, f'_m)] \Rightarrow \exists\, d'_1 \ldots d'_n[\neg d'_i \wedge (\bigwedge_{i=1}^{n} c'_i \Rightarrow \bigwedge d'_i) \wedge CON_I(d'_1, \ldots, d'_n) \wedge f\_covers(d'_1, \ldots, d'_n, f'_1, \ldots, f'_m)]\}$.

The converse is similar.

□

Lemma 10. *(Critical) A component c is critical for f_j iff* $\forall\, p_{c_1}, \ldots, p_{c_n}\ \{formula\_\mathcal{T}(f_j) \Rightarrow p_c\}$.

Proof. Assume c is critical for f_j. Then, every implementation which does not contain c cannot implement f_j. In other words, every implementation in $\overline{\mathcal{C}}$ which implements f_j must contain c. Lets look at $\mathcal{T}(f_j) = \{C_1, \ldots, C_k\}$. Then, c must belong to all the C_i's. Clearly, if this is the case, then whenever $\bigvee_{i=1}^{k} \bigwedge_{d \in C_i} p_d$ is true, so must be p_c : Assume there exists $C_l \in \mathcal{T}(f_j)$ such that $c \notin C_l$. Then clearly, we have an assignment of $p_{c_1}, \ldots, p_{c_n}$ where $\bigwedge_{d \in C_l} p_d$ is true, but $p_c = 0$ (as $c \notin C_l$). Thus, c is critical for f_j iff $\forall p_{c_1}, \ldots, p_{c_n}\{formula\_\mathcal{T}(f_j) \Rightarrow p_c\}$. □

Lemma 11. *(Extends) Let F and F' be subsets of features. Let $\bar{F} = (f_1, \ldots, f_m)$ and $\bar{F}' = (f'_1, \ldots, f'_m)$. Then F' extends F iff $\bigwedge_{i=1}^{m}(f_i \Rightarrow f'_i)$ is true. F' is extendable iff $\exists f'_1, \ldots, f'_m[\bigwedge_{i=1}^{m} f_i \Rightarrow f'_i)]$.*

Proof. If F' extends F, then $\bar{F}(i) = 1 \Rightarrow \bar{F}'(i) = 1$. Then clearly, $\bigwedge_{i=1}^{m}(f_i \Rightarrow f'_i)$ is true. Conversely, if $\bigwedge_{i=1}^{m}(f_i \Rightarrow f'_i)$, then whenever $f_i = 1$, $f'_i = 1$. That is, $\bar{F}(i) = 1 \Rightarrow \bar{F}'(i) = 1$. Clearly, then F' extends F. If F is extendable, then there exists some F' such that F' extends F. This is same as existentially quantifying the variables of F' such that the implication holds. □

3.2 Analysis Operations

Given an SPL $\Psi = \langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, we define the following analysis problems. The problems center around the new definition of an SPL product.

3.2.1 SPL Model Verification

Questions: Is it a valid SPL model? Is it a void SPL model? Is the SPL model complete?

A given SPL model $\Psi = \langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$ is *valid*, if there exists a specification and implementation. Let's assume a feature model with three features f_1, f_2 and f_3. The feature f_1 is the root and the features f_2 and f_3 are the mandatory children of f_1. An *excludes* relation exists between f_2 and f_3. The feature model cannot have any specification because of *excludes* relation and such SPL model is not a *valid model*. An SPL models should be validated before analyzing any operations over it. Large and complex SPLs undergoes continuous modification, such SPLs has to be verified for its validity after every modification. In case of VMPLs, after adding new features, components and cross-tree constraints, a validity of model should be tested. "*Is the virtual machine feature model and component model valid?*", such questions must be verified before further analysis of VMPL.

If all the features have a traceability relation with the components which implement them, such a traceability relation is called as *complete traceability relation*. If there exists a feature which does not have a traceability relation with any components, then such a traceability relation is called an *incomplete traceability relation*. When a SPLs are under development, all the features many not have its corresponding components developed. The operation *complete traceability relation* help us to identify such features and proceed for its components development. The preliminary properties *valid model* and *complete traceability relation* should hold before analyzing any other properties. Let us assume an SPL model $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$ which is a *valid model* but none of the implementations C_i cover any of the specification F_j. Such a model is called *void product model*, i.e., the model is not able

to return a single product. In SPL model, it may happen that a feature model is valid, a component model is valid and a traceability is also complete, but the SPL model is not able to generate a single product. This is possible if no specification *covers* any of the implementation. A question like, "*Do a Virtual Machine Product Line can generate at least one virtual machine ?*" is very important to conduct further analysis of a product line.

3.2.2 Complete and Sound SPL

Question: Is the SPL model adequate for all the user specifications? Do all implementation has it corresponding specification? Which are the useful implementations? Is there at least one implementation which realizes a given user specification?

The *completeness* property of the SPL relates to the implementability of a specification. A specification F is *implementable* if there is an implementation C such that $Covers(C,F)$. Completeness determines if the PL implementation (set of implementation variants) is adequate to provide implementations for all the variant specifications in the PL specification. An SPL $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$ is *complete* if for every $F \in \overline{\mathcal{F}}$, there is an implementation $C \in \overline{\mathcal{C}}$ such that $Covers(C,F)$. The *soundness* property relates to the usefulness of an implementation in an SPL. An implementation is said to be *useful* if it implements some specification in the scope. An SPL $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$ is *sound* if for every $C \in \overline{\mathcal{C}}$, there is a specification $F \in \overline{\mathcal{F}}$ such that $Covers(C,F)$. The completeness and soundness are very crucial properties of any SPLs. *Is a VMPL is able to provide a virtual machine for every valid requirements (specifications) from users?, if YES then the VMPL is complete*. If there is some specification which cannot be implemented by any of

the implementation in the PL implementation, then such PL implementation is not adequate to fill the wish of all the user specifications. In VMPL, there may be such requirements for which no virtual machine can be generated. In such case, either feature model, component model or traceability relation should be analyzed to figure out the actual problem. On the other hand, PL implementation may provide huge set of implementation where as PL specification may be answered by a subset of PL implementation. In case of VMPL, we may end up with such virtual machine which may not get covered by any of the user specifications. Such machine should be removed from the pre-configured machine list.

3.2.3 Product Optimization

Questions: Do the given specification and implementation forms a product? Is there an implementation which provides all the features in a given user specification? Is there an implementation exactly meeting a given user specification? Is there only one implementation for a given specification?

Given a specification, we want to find out all the variant implementations that cover the specification. This is given by a function $FindCovers(F) = \{C | Covers(C,F)\ \}$. At times, it is necessary for a premier set of features to be provided exactly for some product variants. For example, a client company with a critical usage of the product would limit the risk of feature interaction. In this case, we want to find out if there is an implementation that *realizes* the specification. A specification is *existentially explicit* if there exists an implementation C such that $Realizes(C,F)$. Dually, it is *universally explicit* if for all implementations $C \in \overline{C}$, $Covers(C,F)$ implies $Realizes(C,F)$. Multiple implementations may implement a given specification. This may be a desirable criterion of the PL im-

plementation from the perspective of optimization among various choices. Thus, the specifications which are implemented by only a single implementation are to be identified. $F \in \overline{\mathcal{F}}$ has a *unique implementation* if $|FindCovers(F)| = 1$.

Is there a virtual machine which provide all the features as per the client specification? A *Covers* is more relaxed version where a specification is implemented by an *implementation*, but the *implementation* may contain extra components which may not require to implement any of the features in a specification. It may happen that, the cloud may have such pre-configured virtual machines which provides all the features as per user specifications. Furthermore, this pre-configured machines has extra components which are not required to support any of the features in user specifications. This may result in redundancy of components in a virtual machine. *Is there a virtual machine which provides exactly all features as per the client specification?* The tighter version of cover is realize, which strictly does not allow any extra components which are not required for features implementation present in a specification. A realize is the optimized version of cover operation. Finding the optimized virtual machine on cloud which match the exact user specifications is achieved by *realize*. *Is there at least one virtual machine which provides exactly all the features as per the client specification?* The *existentially explicit* operations guarantee the presence of at least one implementation which is realized by a given specifications. It means, in VMPL for a user specification there exists at least one virtual machine which realizes it and this guarantees the presence of at least one optimized configuration. The *universally explicit* is the tighter version of *existentially explicit*, which means all the implementation covers by a given specification implies that it is realization. For *universally explicit* specifications, cloud always 'produces' the optimized virtual

machine. *Is a given user specification has only one virtual machine provided by cloud?* In VMPL, there may be some specification which is covered by only one virtual machine, such implementations are unique.

3.2.4 SPL Optimization

Questions: Is an element is present across all the products? Is an element is used in at least single product? Is an element not in use? Which all elements are redundant in a given product? Which are the extra features provided by a product apart from the given user specification?

Identification of common, live and dead elements in an SPL are some of the basic analyses operations in the SPL community. We redefine these concepts in terms of our notion of products: An element e is *common* if for all $\langle F,C\rangle \in Prod(\Psi)$, $e \in F \cup C$. An element e is *live* if there exists $\langle F,C\rangle \in Prod(\Psi)$ such that $e \in F \cup C$. An element e is *dead* if for all $\langle F,C\rangle \in Prod(\Psi)$, $e \notin F \cup C$. Now a days with the advance in technology, business changes it requirements so quickly that, existing products in market get replaced by another advance product in a very short time span. As the SPLs evolves, new cross-tree constraints get added or removed, this results in change of products. Due to such modification, few features or components in SPL may become *live* or *dead*. *Is the component c is present in at least one virtual machine provided by VMPL? Is the component c is not present in any of the virtual machines provided by VMPL?* The *common* property find all the common elements (features or components) across all the products. This operation is required to create a common platform for a SPL. *Is the component c is present in all the virtual machines provided by VMPL?*

There may be certain implementations that are useful but the implementable

specifications are not affected if these implementations are dropped from the PL implementation. These implementations are called *superfluous*. Formally, an implementation $C \in \overline{\mathcal{C}}$ is *superfluous* if for all $F \in \overline{\mathcal{F}}$ such that $Covers(C,F)$, there is a different implementation $D \in \overline{\mathcal{C}}$ such that $Covers(D,F)$. Superfluousness is relative to a given PL implementation. If in an SPL Ψ, $\overline{\mathcal{F}} = \{\{f\}\}$, $\overline{\mathcal{C}} = \{\{a\},\{b\}\}$ and $\mathcal{T}(f) = \{\{a\},\{b\}\}$, then both the implementations $\{a\}$ and $\{b\}$ are superfluous with respect to Ψ, whereas if either $\{a\}$ or $\{b\}$ is removed from the PL implementation, the remaining implementation ($\{b\}$ or $\{a\}$) is not superfluous anymore (with respect to the reduced SPL). The feature *Java* in VMPL, can be implemented by component *OpenJDK* or *OracleJDK*. Such traceability results in many superfluous implementations . Superfluousness for a specification guarantees the presence of alternate implementations.

Which are the components in the virtual machines that can be removed without impacting the user specification? A component is *redundant* if it does not contribute to any feature in any implementation in the SPL. A component $c \in \mathcal{C}$ is redundant if for every $C \in \overline{\mathcal{C}}$, we have $Provided\_by(C) = Provided\_by(C \setminus \{c\})$. An SPL can be optimized by removing the redundant components without affecting the set of products. Redundant elements may not be dead. Due to the packaging, redundant elements can be part of useful implementations of the SPL and hence be live. *Is the component c is required for any of the features in a user specification?* A component c is *critical* for a feature f in the SPL scope $\mathcal{F}$, when the component *must* be present in an implementation that implements the feature f: for all implementations $C \in \overline{\mathcal{C}}$, $(c \notin C \implies \neg implements(C,f))$. This definition can be extended to specifications as well: a component c is *critical* for a specification F, if for all implementations $C \in \overline{\mathcal{C}}$, $(c \notin C \implies \neg Covers(C,\ F))$.

A virtual machine may contain components which may not be required for any of the features in a user specifications, but it may remain due to packaging. Such components are redundant but not critical.

Can virtual machine provide more features with the same set of components? When a specification is covered (but not realized) by an implementation, there may be extra features (other than those in the specification) provided by the implementation. These extra features are called *extraneous* features of the implementation. Since there can be multiple covering implementations for the same specification, we get different choices of implementation and extraneous features pairs: $Extra(F) \equiv \{\langle C, Provided\_by(C) \setminus F\rangle | Covers(C,\ F)\}$. User may demand for virtual machines with some specification. The available pre-configured machine provide all the features in user specification, and also provide few more features which are extraneous.

3.2.5 Generalization and Specialization in SPL

Questions: Is the union of two or more products result in a new product? What is the difference between two products?

In an SPL, sometimes there is a need to check the *aggregation* relationship between the specifications, implementations or products. *Is there a virtual machine which has features provided by a given set of virtual machines?* The *union* property on two specifications will result in a new specification which has features of both the specifications. Let's say specification F_1 has features $\{f_1, f_2, f_3\}$ and specification F_2 has features $\{f_2, f_5, f_7\}$. The *union* property will check for some specification F which has features of specifications F_1 and F_2, so F should have features $\{f_1, f_2, f_3, f_5, f_7\}$. Assume an *excludes* relation between features

f_3 and f_5, then the union property will return FALSE. In VMPL, the user always demand a virtual machines which has equivalent features of two or more machines. The union property is used to verify the combination of two or more virtual machines is valid. Similar to specifications, this property can be applied on implementations or products.

In an SPL, most of the time there is a need to distinguish between the multiple specifications or implementations or products. *Is there a virtual machine whose features are present in all virtual machines in a given set?* The *intersection* property on multiple specifications will check the existence of any specification which is common to those specifications. Let's say specification $F_1 = \{f_1, f_2, f_3\}$ and $F_2 = \{f_1, f_2, f_7\}$, then the *intersection* property applied on specification F_1 and F_2 will result in specification $F = \{f_1, f_2\}$. The distinguishable features or variants between F_1 and F_2 are obtained as $F_1 \setminus F = \{f_3\}$ and $F_2 \setminus F = \{f_7\}$. A specification which is contained in all the specifications of an SPL is called *core specification*. The *intersection* property applied on a given SPL model will result in a *core specification*. Similar to specifications, this property can be applied to implementations or products.

In the literature, different analysis problems in SPLs are usually encoded as satisfiability problems for propositional constraints [59] and SAT solvers such as Yices [74] or Bddsolve [75] are used to solve them. As it has been noted in [6], it is not possible to cast certain problems such as completeness and soundness as a single propositional constraint. However, we observe that these problems need quantification over propositional variables encoding features and components. The expressive logic formalism, Quantified Boolean Formula (QBF) as compared to propositional logic, is necessary to encode such analysis problems.

The Boolean satisfiability problem for a propositional formula is then naturally extended to a QBF satisfiability problem (QSAT).

Given an SPL Ψ, each of the properties listed in Table 3.1 holds if and only if the corresponding formula evaluates to true.

3.3 Validation

In order to validate the approach presented in this report, a tool SPLAnE [38] for the automated analysis of SPL models has been developed. SPL models consists of feature models with traceability relationships to the component models (Core assets). The Virtual Machine Product Line (VMPL) case study based on cloud computing concepts is presented and analyzed.

The tool SPLAnE provide analysis operations: valid model, complete traceability, void product model, implements, covers, realizes, soundness, completeness, existentially explicit, universally explicit, unique implementation, common, live, dead superfluous, redundant, critical, union and intersection. SPLAnE encode each analysis operation in single QBF. The tool FaMa [37] provide analysis operations—commonality, core features, dead feature, detect error, explain error, filter question, unique feature, variability question, valid configuration, variant feature and valid product [5]. FaMa encode each analysis operation in single propositional formula. Every analysis operation of FaMa can be encoded in QBF and can be solved by SPLAnE , where as the formula like soundness cannot be encoded in a single SAT formula. To compare our QSAT approach with SAT approach we implemented analysis operation provided by SPLAnE on FaMa. There are few analysis operations provided by SPLAnE like valid model, complete trace-

Table 3.1: Properties and Formulae

Properties	Formula
Valid Model	$\exists p_{f_1} \ldots p_{f_m} \exists p_{c_1} \ldots p_{c_n} [\mathrm{CON}_I] \wedge [\mathrm{CON}_F]$
Complete Traceability	$\exists p_{c_1} \ldots p_{c_n} \{(\mathcal{T}(p_{f_1}) \wedge \cdots \wedge \mathcal{T}(p_{f_m})) \implies (p_{c_1} \vee p_{c_2} \vee \cdots \vee p_{c_n})\}$
Void Product Model	$\exists p_{f_1} \ldots p_{f_m} \exists p_{c_1} \ldots p_{c_n} [\mathrm{CON}_I] \wedge [\mathrm{CON}_F] \wedge \neg f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})$
$Implements(C,f)\ \bar{C} = (v_1, \ldots, v_n)$	$form\_implements_f(v_1, \ldots, v_n)$
$Covers(C,F)$ $Realizes(C,F)$ $\bar{C} = (v_1, \ldots, v_n), \bar{F} = (u_1, \ldots, u_m)$	$f\_covers(v_1, \ldots, v_n, u_1, \ldots, u_m)$ $f\_realizes(v_1, \ldots, v_n, u_1, \ldots, u_m)$
Ψ complete	$\forall p_{f_1} \ldots p_{f_m} \{\mathrm{CON}_F \Rightarrow$ $\exists p_{c_1} \ldots p_{c_n} [\mathrm{CON}_I \wedge f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})]\}$
Ψ sound	$\forall p_{c_1} \ldots p_{c_n} \{\mathrm{CON}_I \Rightarrow \exists p_{f_1} \ldots p_{f_m} [\mathrm{CON}_F \wedge f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})]\}$
F existentially explicit $\bar{F} = (u_1, \ldots, u_m)$	$\exists p_{c_1} \ldots p_{c_n} \{\mathrm{CON}_I \wedge f\_realizes(p_{c_1}, \ldots, p_{c_n}, u_1, \ldots, u_m)\}$
F universally explicit $\bar{F} = (u_1, \ldots, u_m)$	$\exists p_{c_1} \ldots p_{c_n} \{\mathrm{CON}_I \wedge f\_realizes(p_{c_1}, \ldots, p_{c_n}, u_1, \ldots, u_m)\} \wedge$ $\forall p_{c_1} \ldots p_{c_n} \{[(\mathrm{CON}_I \wedge f\_covers(p_{c_1}, \ldots, p_{c_n}, u_1, \ldots, u_m)] \Rightarrow$ $f\_realizes(p_{c_1}, \ldots, p_{c_n}, u_1, \ldots, u_m)\}.$
F has unique implementation $\bar{F} = (u_1, \ldots, u_m)$	$\exists p_{c_1} \ldots p_{c_n} [\mathrm{CON}_I \wedge f\_covers(p_{c_1}, \ldots, p_{c_n}, u_1, \ldots, u_m)] \wedge$ $\forall q_{c_1} \ldots q_{c_n} ((\mathrm{CON}_I[q_{c_1} \ldots q_{c_n}] \wedge f\_covers(q_{c_1}, \ldots, q_{c_n}, u_1, \ldots, u_m)) \Rightarrow (\wedge_{l=1}^{n} (p_{c_l} \Leftrightarrow q_{c_l})))]$
c_i common	$\forall p_{c_1} \ldots p_{c_n} p_{f_1} \ldots p_{f_m} \{(\mathrm{CON}_I \wedge \mathrm{CON}_F \wedge$ $f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})) \Rightarrow p_{c_i}\}$
c_i live	$\exists p_{c_1} \ldots p_{c_n}, p_{f_1} \ldots p_{f_m} \{(\mathrm{CON}_I \wedge \mathrm{CON}_F \wedge$ $f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})) \wedge p_{c_i}\}$
c dead	$\forall p_{c_1} \ldots p_{c_n} p_{f_1} \ldots p_{f_m} \{(\mathrm{CON}_I \wedge \mathrm{CON}_F \wedge$ $f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})) \Rightarrow \neg p_{c_i}\}$
C superfluous $\bar{C} = (v_1, \ldots, v_n)$	$\forall p_{f_1} \ldots p_{f_m} [(\mathrm{CON}_F \wedge f\_covers(v_1, \ldots, v_n, p_{f_1}, \ldots, p_{f_m})) \Rightarrow$ $\exists p_{c_1} \ldots p_{c_n} (\mathrm{CON}_I \wedge \vee_{i=1..n} (p_{c_i} \neq v_i) \wedge$ $f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m}))]$
c_i redundant	$\forall p_{c_1} \ldots p_{c_n} p_{f_1} \ldots p_{f_m} \{(p_{c_i} \wedge \mathrm{CON}_I \wedge \mathrm{CON}_F \wedge$ $f\_covers(p_{c_1}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})) \Rightarrow$ $f\_covers(p_{c_1}, \ldots, \neg p_{c_i}, \ldots, p_{c_n}, p_{f_1}, \ldots, p_{f_m})\}$
c_i critical for f_j	$\forall p_{c_1} \ldots p_{c_n} [form\_implements_{f_j}(p_{c_1} \ldots p_{c_n}) \Rightarrow p_{c_j}]$
Union	$\exists p_{f_{11}}, \ldots, p_{f_{1n}} \exists p_{f_{21}}, \ldots, p_{f_{2n}} \exists f_1, \ldots f_n \{f_1 \Leftrightarrow (p_{f_{11}} \vee p_{f_{21}})$ $\ldots f_n \Leftrightarrow (p_{f_{1n}} \vee p_{f_{2n}}) \wedge [\mathrm{CON}_F]\}$
Intersection	$\exists p_{f_{11}}, \ldots, p_{f_{1n}} \exists p_{f_{21}}, \ldots, p_{f_{2n}} \exists f_1, \ldots f_n \{f_1 \Leftrightarrow (p_{f_{11}} \wedge p_{f_{21}})$ $\ldots f_n \Leftrightarrow (p_{f_{1n}} \wedge p_{f_{2n}}) \wedge [\mathrm{CON}_F]\}$

ability, void product model, implements, covers, realizes and live which uses only one existential quantifiers or universal quantifier, and can be encoded in SAT and executed with FaMa. The operations like soundness, completeness, existentially explicit and universally explicit cannot be encoded in single SAT, so for experimental comparison such formula is executed with FaMa in iteration.

3.3.1 SPLAnE Architecture

SPLAnE (Software Product Line Analysis Engine) is designed and developed to analyze the traceability between the features and implementation assets. Nowadays, there is the large set of tools that enable the reasoning over feature models. However, none of them is capable of reasoning over the feature model and a set of implementations as described throughout this report. For the sake of reusability and because it has been proven to be easily extensible [76, 77], we chose to use the FaMa framework [37] as the base for SPLAnE . The FaMa framework provides a basic architecture for building FM analysis tools while defining interfaces and standard implementation for existing FM operations in the literature such as *Valid Model* or *Void Product Model*.

On the one hand, SPLAnE benefits from being a FaMa extension in different ways. Fama is capable to analyze SAT based problems only across feature model. SPLAnE can analyze more complex problems encoded in QBFs across feature model, component model and traceability relation. For example, SPLAnE can read a large set of different file formats used to describe feature models. It is also possible to perform some of the existing operations in the literature to the feature model prior to executing the reasoning over the component layer. On the other hand, FaMa was not designed for reasoning over more than one model. There-

fore, different modifications have been addressed to fill this gap. Namely, (i) we modified the architecture to enable this new extension point into the FaMa architecture; (ii) created a new reasoner for a new set of operations; (iii) implemented the operations, and (iv) defined two new file formats to store and input traceability relationships and component models in SPLAnE .

The reasoning process performed by SPLAnE is shown in the Figure 5.1. First, SPLAnE takes as input a feature model, a traceability relationship and a component model. The SPLAnE parser creates the SPL model from this input files. Second, SPLAnE constructs the QBF/QCIR formula based on the selected analysis operations. QBF is further encoded in qpro format, this is done by SPLAnE translator. qpro [78] and QCIR [79] format is a standard input file format in non-prenex, non-CNF form. Later, SPLAnE invokes the QSAT solver CirQit [35] or RaReQS [36] in the back-end to check the satisfiability of the generated QBFs in qpro/QCIR format. The choice of the tool is based upon its performance: CirQit has solved the most number of problems in the non-prenex, non-CNF track of QBFEval'10 [80]. RaReQS [36] is a Recursive Abstraction Refinement QBF Solver. Table 3.1 shows the analysis operations provided by SPLAnE .

The design of SPLAnE makes it possible to use different QSAT solvers. Furthermore, SPLAnE can now work hand in hand with other products based on FaMa such as Betty [37], which enables the testing of feature models. SPLAnE is now available for download with its detailed documentation from the website [38].

3.3.2 Experimentation

In this section, we go through the different experiments executed to validate our approach. The experiments was conducted with (i) Real Debian models, (ii) Ran-

domly generated models and (iii) SPLOT (Software Product Line Online Tools) Repository models. Each analysis operation was executed with two QSAT solver (CirQit and RaReQS) and three SAT solver (Sat4j [32], PicoSAT [34] and MiniSAT [33]). All experiments was run on a 3.2 GHz i7 processor (Intel Corporation, Santa Clara, CA, USA) machine with 16 GB RAM.

Table 3.2: Hypotheses and design of experiments.

Hypotheses of Experiment 1	
Null Hypothesis (H_0)	SPLAnE does not scale when coping with SPLOT model repository.
Alt. Hypothesis (H_1)	SPLAnE does scale when coping with SPLOT model repository.
Models used as input	*Feature Model for TPL, MPPL and ESPL were taken from [81]. ECPL is taken from [82]. VMPL is presented in current report. SPLOT repository. The 69,800 SPL models were generated from 698 SPLOT Models.*
Blocking variables	For each SPLOT model, we used 10 different topology and 10 level of cross-tree constraints to get 100 SPL models. Percentages of cross-tree constraints were 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45% and 50%.
Hypotheses of Experiment 2	
Null Hypothesis (H_0)	SPLAnE does not scale when coping with randomly generated SPL models.
Alt. Hypothesis (H_1)	SPLAnE does scale when coping with randomly generated SPL models.
Model used as input	*1000 Randomly generated SPL Models.*
Blocking variables	We generated 10 random feature models with the number of features as 10, 50, 100, 500, 1000, 3000, 5000, 10,000, 15,000 and 20,000. For each feature model, 100 SPL models were generated by changing it to 10 different topology across 10 different cross tree constraints. Number of components in each model were three-times the number of features. Percentages of cross-tree constraints: 5%, 10%, 15%, 20%, 25%, 30%, 35%, 40%, 45% and 50%.

Table 3.4: *Hypotheses and design of experiments. Cont... 1*

Hypotheses of Experiment 3	
Null Hypothesis (H_0)	The use of SPLAnE will not result in a faster executions of operations than SAT-based techniques in front of a real very-large SPL models.
Alt. Hypothesis (H_1)	The use of SPLAnE will result in a faster executions of operations than SAT-based techniques in front of a real very-large SPL models.
Model used as input	*We used as input the Debian variability model extracted from [77] that you can find at [38]*
Hypotheses of Experiment 4	
Null Hypothesis (H_0)	The use of SPLAnE will not result in a faster executions of operations than SAT-based techniques in front of randomly generated SPL models.
Alt. Hypothesis (H_1)	The use of SPLAnE will result in a faster executions of operations than SAT-based techniques in front of randomly generated SPL models.
Model used as input	*We used as input random models varying from ten features to twenty thousand features.*

Table 3.6: *Hypotheses and design of experiments. Cont... 2*

Hypotheses of Experiment 5	
Null Hypothesis (H_0)	The QSAT based reasoning technique is not faster as compare to SAT based technique for operations like *completeness* and *soundness*.
Alt. Hypothesis (H_1)	The QSAT based reasoning technique is faster as compare to SAT based technique for operations like *completeness* and *soundness*.
Model used as input	*We used as input random models varying from ten features to twenty thousand features and SPLOT repository models.*
Constants	
QSAT and SAT solvers	CirQit solver [35], RaReQS solver [36], Sat4j [32], PicoSAT [34] and MiniSAT [33]
Heuristic for variable selection in the QSAT and SAT solver	*Default*

Alt.: Alternative; *TPL*: Tablet Product Line; *MPPL*: Mobile Phone Product Line; *ESPL*: Electronic Shopping Product Line; *ECPL*: Entry Control Product Line.

Experiment 1: Validating SPLAnE with Feature Models from the SPLOT Repository

To illustrate the SPL analysis method described in the report, we considered case studies of various sizes. Concretely, the following SPLs were used: Entry Con-

trol Product Line (ECPL), Virtual Machine Product Line (VMPL), Mobile Phone Product Line (MPPL), Tablet Product Line (TPL) and Electronic Shopping Product Line (ESPL). The TPL, MPPL, and ESPL models were taken from the SPLOT repository [81]. More details of the ECPL models can be found at [82]. Table 3.7 gives the number of features, components in each SPL model, and the execution time taken by various analysis operations on SPLAnE reasoner: CirQit.

The SPLOT repository is a common place where a practitioners store feature models for the sake of reuse and communication. The SPLOT repository contains small and medium size feature models, most of it are conceptual and few are realistic. We extracted 698 feature models from the SPLOT repository. These feature models were given as an input to extended Betty tool to generate corresponding SPL models. Usually, components models which represent the solution space of SPLs are larger in size. So, the generated component models contain three-times more components then the number of features present in the corresponding feature model. SPLAnE generated the random traceability relation between feature model and component model to generate a complete SPL model. Further, to increase the complexity of experiments, each SPL model is generated using 10 different topologies and 10 different level of cross-tree constraints with percentage as $\{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$, resulting in a total of 100 SPL models per SPLOT model. So from 698 SPLOT models, we got 69800 SPL models. The percentage of cross-tree constraint is defined by the percentage of constraints over the number of features. Basically it is the number of constraints depending on the number of features. For example, if we specify a 50% percentage over a model with 10 features, then we have five cross-tree constraints.

Table 3.7: Time complexity for properties and formulae with SPLAnE reasoner: CirQit. (# means the "Number" of features and components.)

SPL name	**ECPL**	**VMPL**	**MPPL**	**TPL**	**ESPL**
#Features	8	15	25	34	290
#Components	12	20	41	40	290
Analysis Operations	Time (ms)	Time (ms)	Time (ms)	Time (ms)	Time (ms)
Valid Feature Model	10	14	14	20	35
Valid Component Model	12	15	13	12	37
Valid SPL Model	14	16	15	28	40
Void Product Model	18	25	23	29	55
Valid Specification	7	13	11	17	45
Valid Implementation	9	15	13	12	48
Complete Traceability	0	1	0	1	1
Implements	8	10	9	10	22
Realizes	10	26	24	14	78
Covers	12	13	10	14	74
Completeness	18	30	26	284	2135
Soundness	350	2120	1323	730	6550
Common	15	22	19	28	74
Live	18	45	61	25	82
Dead	11	20	18	27	65
Redundant	14	24	19	21	38
Critical	10	14	15	19	34
Union over Specifications	14	18	16	26	45
Union over Implementations	18	30	25	35	37
Union over Products	18	24	20	32	48
Intersection over Specifications	9	14	11	22	37
Intersection over Implementations	11	17	16	28	28
Intersection over Products	15	20	21	19	46

The tool SPLAnE was executed with 69800 SPL models to verify the QSAT scalability when applying it to feature modeling. SPLAnE provides an option to select any one of the two QSAT reasoners (CirQit and RaReQS). Figure 3.1 shows the box plot, representing the QSAT behavior with increase in cross-tree constraints for few analysis operations on real models taken from the SPLOT repository. This experiment was executed with the QSAT reasoner CirQit. The results for experiment 1 (Section 3.3.2) shows that for the small and medium size real models, all analysis operations does not take much execution time which motivated us to experiment with large size models. The overall results for experiment 1 (Section 3.3.2) point out that the null hypothesis H_0 was wrong, thus, resulting in the acceptance of the alternative hypothesis H_1.

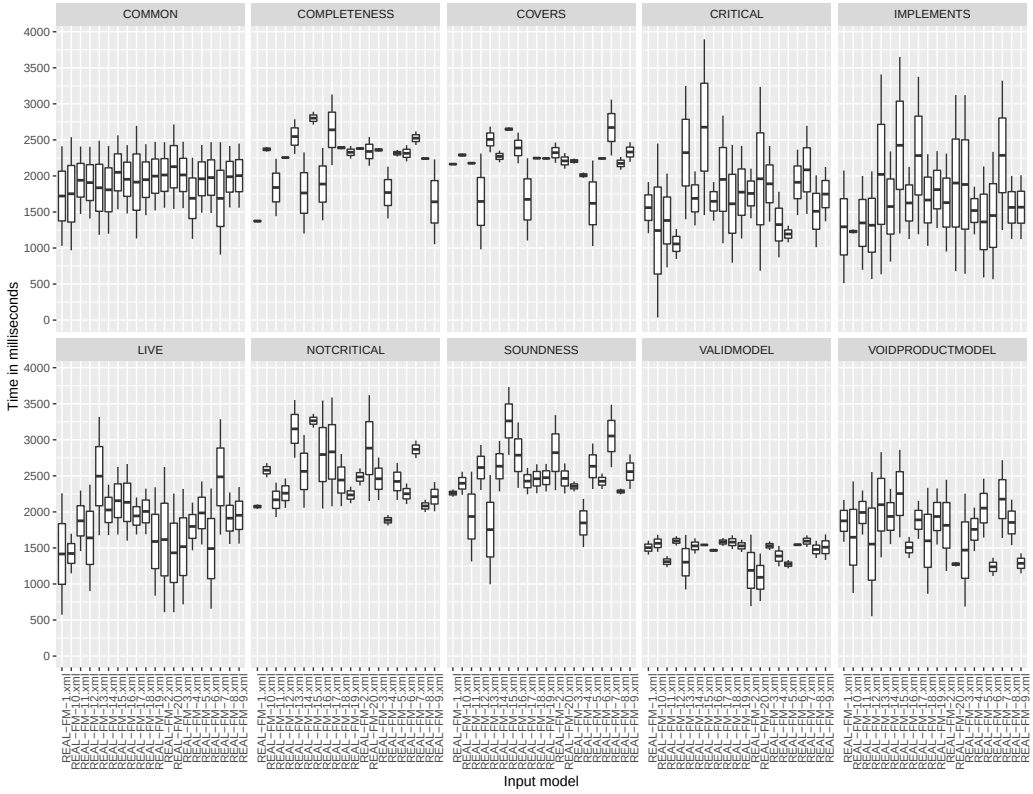

Figure 3.1: Impact on QSAT scalability on Real SPLOT models with the increment in Cross Tree Constraints (CTC) levels.

Experiment 2: Validating SPLAnE with Randomly Generated Large Size SPL Models

In this experiment we have compared scalability of SPLAnE and FaMa based analysis techniques over a large size randomly generated SPL models. The Betty tool suite [83] is used to generate random feature models relying on the approach of Thüm et al. [84]. SPLAnE extended the Betty tool suite to generate a set of random SPL models. Those models were generated for a number

of features ranging from ten features to twenty thousand features. Concretely, $\{10, 50, 100, 500, 1000, 3000, 5000, 10,000, 15,000, 20,000\}$ features with three-times larger size of each component models. For each SPL model, 10 different topologies were generated to avoid the threats to internal validity. Further, to increase the complexity of experiments, 10 different levels of cross-tree constraints $\{5, 10, 15, 20, 25, 30, 35, 40, 45, 50\}$ were added. Each randomly generated SPL models consists of a feature model, a component model and a traceability relation. Note that, for model with 20,000 features there are 60,000 components in component model, which result in 80,000 variables in a generated SPL model. For each model, 10 different topologies and 10 levels of cross-tree constraints will result in 100 random SPL models, so in total 1000 SPL model were generated. The tool SPLAnE was executed against randomly generated 1000 SPL models to check the scalability of our approach with all analysis operations. Figure 3.2 shows the box plot for randomly generated large size SPL models. For data clarity we plotted only eight analysis operations for all models (except models with 50 features) and $\{10, 20, 30, 40, 50\}$ cross-tree levels of constraints. The experiment 2 (Section 3.3.2 was executed with SPLAnE reasoner: RaReQS. The plot clearly shows the QSAT approach is more scalable even with 80,000 variables in a SPL model with maximum 50% constraints. The Figure 3.2 shows that the execution time for all analysis operation grows with the increase in number of features. Figure 3.3 shows the graph plot for the same results, which help to clearly distinguish the behavior of each analysis operations against the Cross Tree Constraints (CTC) levels. From the graphs we observed that, the number of features in a model has more impact on the execution time than the different levels of CTC. The levels of CTC has very less impact on the execution time. The operation *soundness* takes

more time as it checks *for all implementations there exist a specification* and the number of components are three times more than the number of features. The operation *completeness* takes less time as compared to *soundness*, but takes more time compared to all remaining operations. In completeness we checked for all specifications there exist an implementation, here the number of features are less compared to the number of components. So the completeness requires less execution time compared to soundness. The results for experiments 2 (Section 3.3.2) shows that, SPLAnE can scale up to 80,000 variables size models and this rule out the hypothesis H_0 with no option to accept the alternative hypothesis H_1.

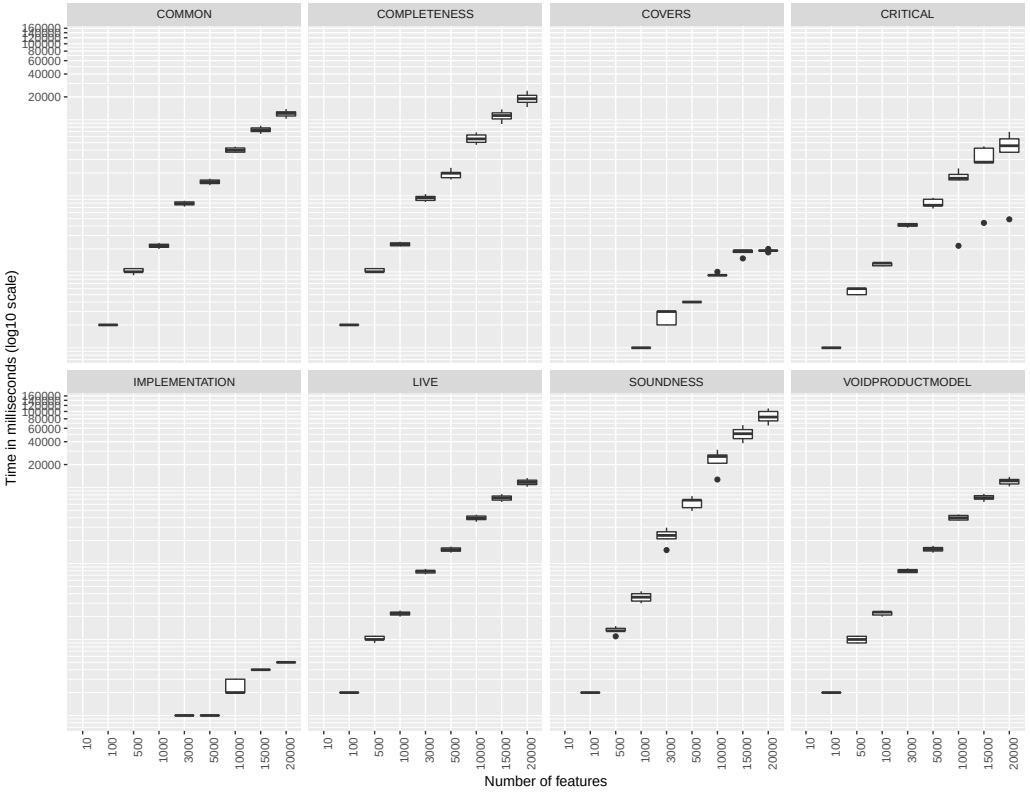

Figure 3.2: Boxplot for QSAT scalability on large random SPL models with the increment in CTC levels.

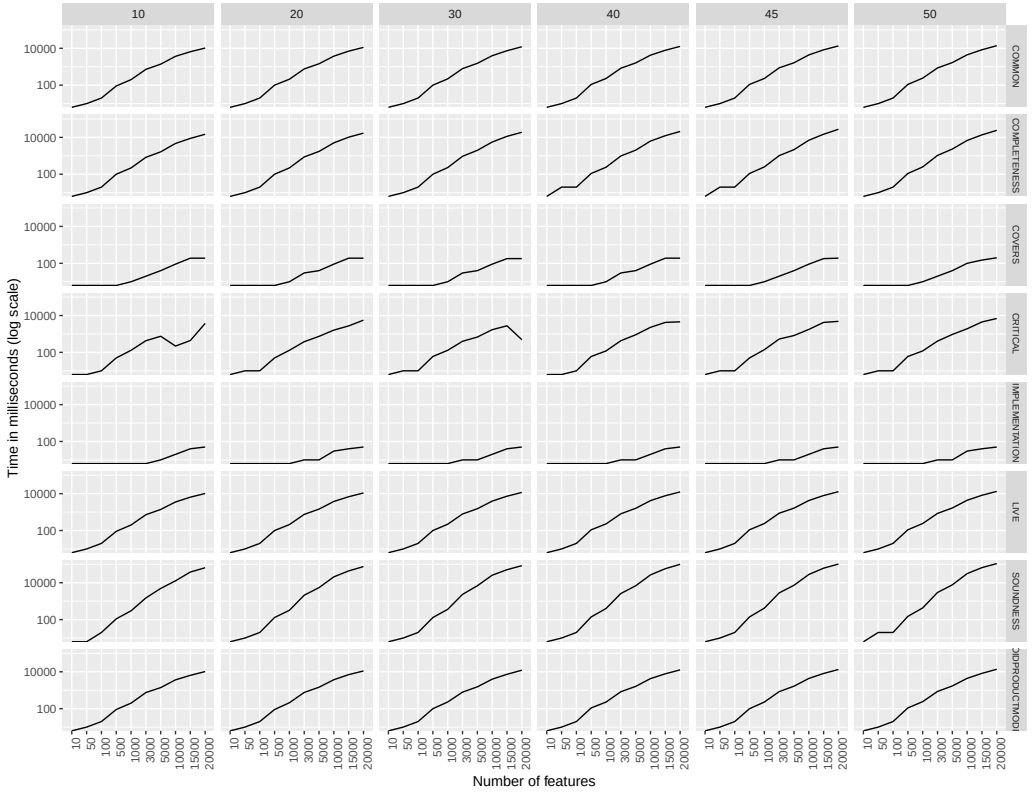

Figure 3.3: QSAT scalability on large random SPL models with the increment in CTC levels.

Experiment 3: Comparing SPLAnE and FaMa Approach in Front of Real and Large Debian Models

This experiment checks the behavior of SPLAnE reasoners (CirQit and RaReQS) and FaMa reasoners (Sat4j, PicoSAT and MiniSAT) on real and large Debian models with the analysis operation presented in the report. We used the feature model extracted from Debian distributions [77]. This model encodes the variability present in the Ubuntu 10.04 distribution packaging system. We used four

initial models containing the data from the repositories: main (7065 features), restricted (7098 features), multiverse (8122 features) and universe (26,338 features). To generate the SPL model from this real feature model, we used the same actual models as component models and added 1:1 traceability relation from each features to components. Consider, the universe Debian model with 26,338 features then its corresponding SPL model will contain 52,676 variables.

Figure 3.4 shows the performance of SPLAnE reasoners (CirQit and RaReQS) and FaMa reasoners (Sat4j, PicoSAT and MiniSAT) against the proposed analysis operations. We see that both approaches scale for all operations in first three Debian models except the completeness and soundness where QSAT is clearly more efficient. For completeness and soundness operations, FaMa reasoners was not able to solve even a single instance of the Debian models. For the fourth model, i.e., universe Debian model, FaMa reasoners was not able to solve any of the analysis operations. Whereas QSAT reasoners against completeness and soundness operations, was able to solve first three Debian models (main, restricted, multiverse), but was not able to solve the huge universe Debian model (26,338 features) in the given timeout (two hours). Overall, for the operations where both approaches scale up, QSAT is faster than SAT. This experiments clearly accepts the hypothesis H_1.

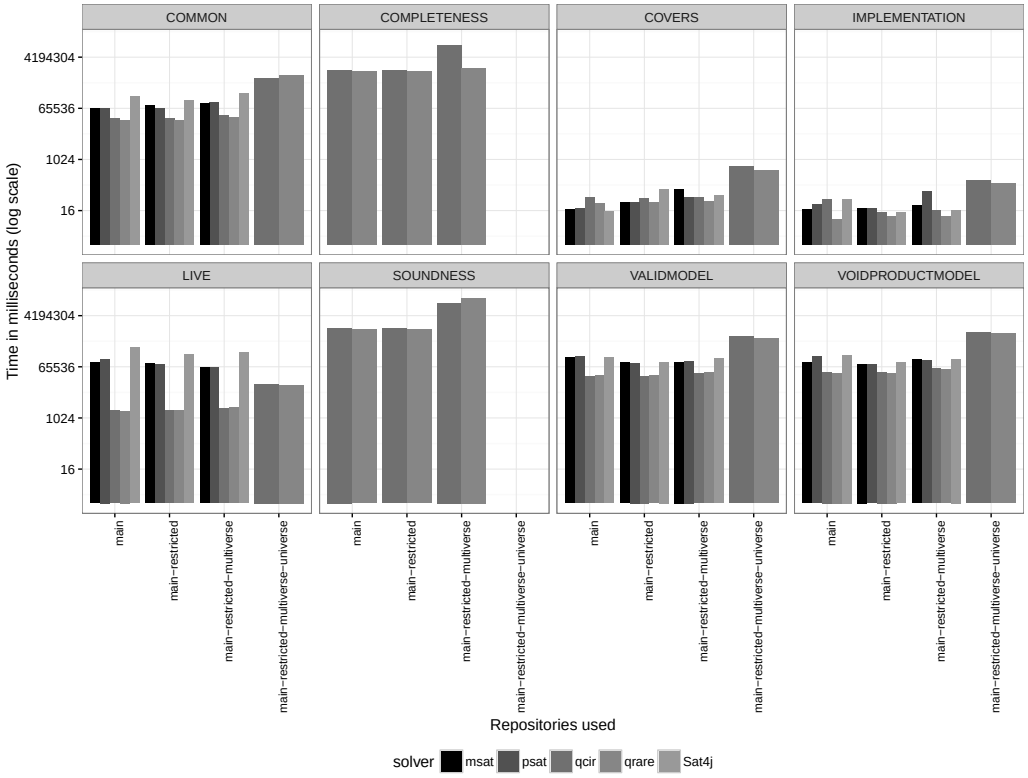

Figure 3.4: SPLAnE required time vs. FaMa required time in front of real and large Debian based feature models.

Experiment 4: Comparing SPLAnE and FaMa Scalability in Front of Randomly Generated Large Size Models

In this experiment, we checked the behavior of SPLAnE reasoners (CirQit and RaReQS) and FaMa reasoners (Sat4j, PicoSAT and MiniSAT) on randomly generated SPL models taken from experiments 2 (Section 3.3.2). Figure 3.5 shows the scalability of SPLAnE reasoners and FaMa reasoners against randomly generated models. The results are only shown for large models from 1000 features

to 20,000 features with 50% cross-tree constraints. Here, the feature model with 10,000 features means its corresponding SPL model contains 40,000 variables with 50% CTC. The results clearly shows that, the SAT reasoners are not able to solve any analysis operations after getting a models of size 10,000 features or more. For completeness and soundness operations, SAT reasoners was not able to solve any SPL models after 1000 features. The QSAT reasoners were able to solve all analysis operations on the random SPL models. The results deny the hypothesis H_0 with no option left to accept the hypothesis H_1.

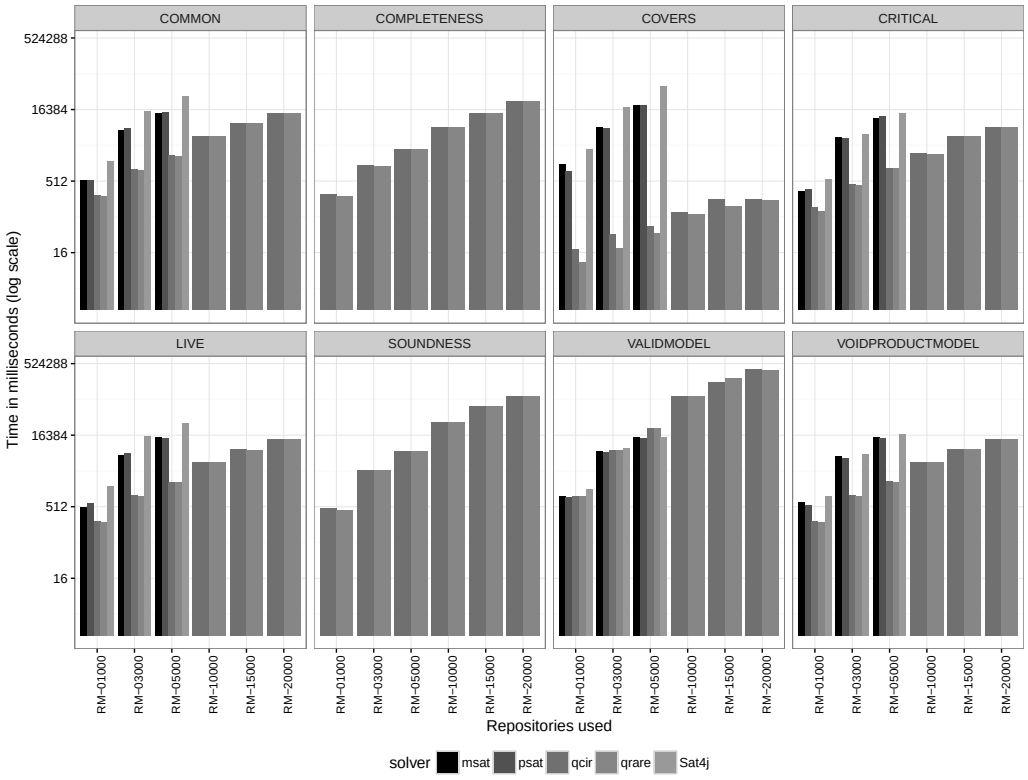

Figure 3.5: SPLAnE required time vs. FaMa required time in front of random and large SPL models.

Experiment 5: Comparing SPLAnE with FaMa based Reasoning Techniques

The tool SPLAnE improves the performance with the set of models obtained from SPLOT and random SPL models. In this experiment, we are comparing QSAT based technique with SAT based techniques over the analysis operations. From the SPLOT models used in the experiment 1 (Section 3.3.2) we took those marked as realistic. FaMa supports analysis operations expressed using propositional formulae. We acknowledge that there are analysis operations such as *completeness* and *soundness* that cannot be expressed using single propositional formulae like QBF. For example, the formula with 3 boolean variable can end up with 8 propositional formulae and for soundness or completeness analysis operation this all 8 formulae needs to be verified. So, for comparing QSAT vs. SAT reasoning, such operations where written in the FaMa tool suite with loop statements (*for* or *while*) for traversing the whole set of solutions. Here, the loop allows us to express such operations (*completeness*, *soundness*, etc.) to its equivalent QSAT formula but note that, the complete operations cannot be expressed using standalone propositional formula. Later, we executed the analysis operations with SPLAnE reasoner (CirQit and the FaMa reasoner) Sat4j.

Figure 3.6 shows the results for QSAT vs. SAT based reasoning for few analysis operations on real models taken from the SPLOT repository. QSAT performs better than SAT encoded formulae for every analysis operations. The execution of all models is available on website at [38] as well as the scripts used to generate this data. The first noticeable results are that SPLAnE overtakes all executions of all operations when comparing to the standard FaMa version (which is using Sat4j as a solver). Moreover, we see improvements of more than 70% (*note the log scale*) when talking about the soundness operation. Therefore, after trying to

refute the null hypothesis H_0 (for experiment 5 (Section 3.3.2)) with no luck, we have to accept the alternative hypothesis H_1 which states that SPLAnE is faster and scalable than previously standard SAT-based techniques.

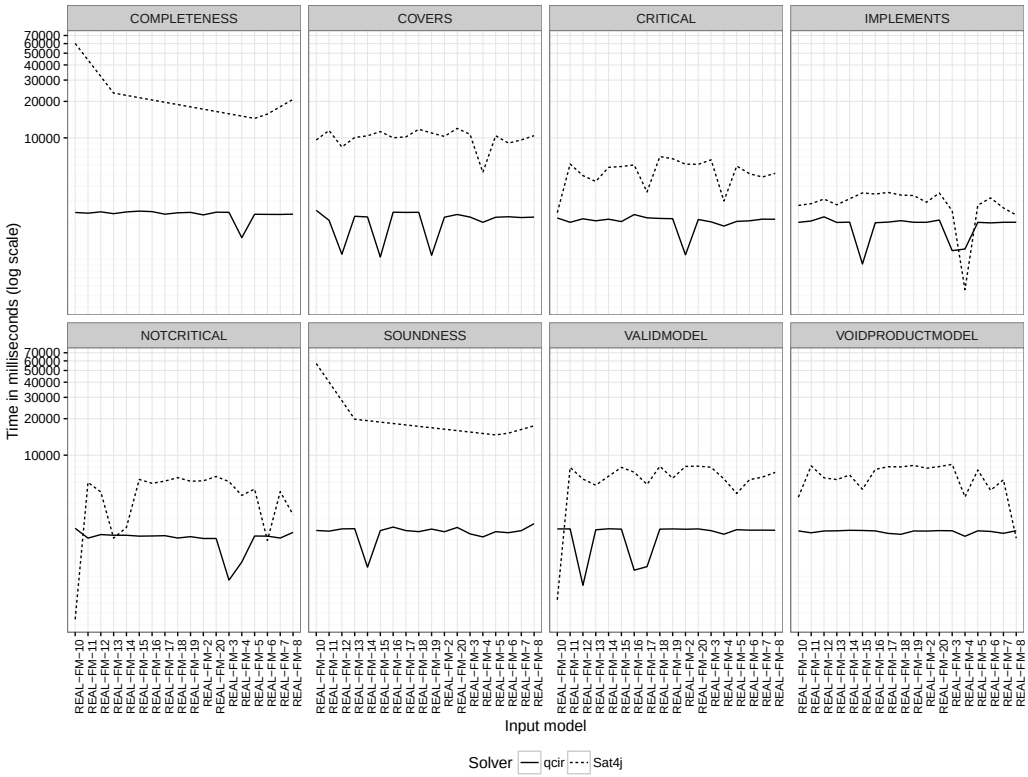

Figure 3.6: SPLAnE required time vs. FaMa required time.

3.3.3 Threats to Validity

Even though the experiments presented in this report provide evidence that the solution proposed is valid, there are some conditions that may affect their validity. In this section, we discuss the different threats to validity that affect the evaluation.

External Validity: The inputs used for the experiments presented in this report

try to mimic realistic feature models. However, SPLOT models are not necessarily realistic. To ease off this threat we decide to used feature models based on the Debian repository. Furthermore, the random feature models may not reflect the same structure as other realistic models. The major threats to the external validity are:

- *Population validity*, the models may not be realistic. To reduce these threats, we generated the models as in [84] and implemented in the Betty tool [83]. We also used models coming out the Debian repositories to provide more realistic topologies.

- *Ecological validity*: While external validity, in general comes with the generalization of the results to other contexts (e.g., using other models), the ecological validity faces the threats affecting the experiment materials and tools. To prevent the threats of third party threads running on the machines, SPLAnE analyses were executed 10 times and then averaged.

Internal validity: The CPU capabilities required when analyzing an SPL model depend on the number of features, components and percentage of cross-tree constraints. However, there might be some variables affecting the performance, such as the topology, so we generated 10 different topologies for each SPL model.

Construct validity: The results look promising in terms of time required to solve problems related to the feature model. However, we can not grant its validity with models more than 20,000 features.

Chapter 4

SPL Engine for Design verification

4.1 Modeling features and SPLs

We use a slightly different notion of feature from the one used in the SPL literature. In prior works such as in [41], a feature is an incremental function or computation and a system is composed of a core function with an optional list of features. The core function with one or more features satisfying the variability constraints define the set of all products in the SPL. An important point to be noted is that the system exhibits variability but not the features. In contrast, our approach considers a feature as a basic unit of functionality and a system is a composition of features. Here, each feature may exhibit variability and the system variability is inherited from that of its constituent features. We adopted this terminology as it is consistent with the automotive application domain, which triggered this work. Recent research activities start considering the case of feature with attributes ([10]).

In this section, we focus on individual features and their modeling with and

without time. As noted in the introduction, we restrict ourselves in this report to only two levels of abstraction in the development life-cycle: the requirement and design levels.

4.1.1 Modeling the behavior of a single feature

A feature, as mentioned above, exhibits variability. In order to describe a feature, we introduce, *Finite State Machines with Variability (FSMv)*, an extension of finite state machines, Let *Var* be a finite set of variables, each taking a value ranging over a finite set of values. Given $x \in Var$, and let $Dom(x)$ denote the finite set of values x can assume. We define, A_{Var}, to be the following set of atomic formulas over *Var* and the values in the domains of variables in *Var*: $x = a$, $x \neq a$, for $a \in Dom(x)$, and $x = y$, $x \neq y$ for $x, y \in Var$. Define $\Delta ::= A_{Var} \mid \neg\Delta \mid \Delta \wedge \Delta \mid \Delta \vee \Delta \mid \Delta \Rightarrow \Delta$ to be the set of all well formed predicates over *Var*.

Definition 1 (FSMv)**.** *An FSMv is a tuple* $\mathcal{A} = \langle Q, q_0, \Sigma, Var, E, \rho \rangle$ *where:*
(1) Q *is a finite set of states;* q_0 *is the initial state; (2)* Σ *is a finite set of events; (3) Var is a finite set of variables; (4)* $E \subseteq Q \times \Delta \times \Sigma \times Q$ *gives the set of transitions. A transition* $t = (s, g, a, s')$ *represents a transition from state* s *to state* s' *on event* a*; the predicate* g *is called a guard of the transition* t*;* g *defines the variability domain of the transition; (5)* $\rho \in \Delta$ *is a consistent predicate called the global predicate.*

The variables in *Var* determine the variability allowed in the feature with each possible valuation of the variables corresponding to a variant. The allowed values of the variables are constrained by the global predicate ρ. For example, if ρ is $((x = 1) \vee (x = 2)) \wedge (y \neq x)$, then the allowed variants are those for which the

values for the pairs (x,y) are $(1,2),(2,1)$ (with $y \in \{1,2\}$). The predicate in a transition determines the variants to which the transition is applicable. While drawing a transition $t = (s,g,a,s')$, the edge connecting s to s' is decorated with $g : a$. When g is the predicate "true", we simply write a on the edge.

Definition 2 (Configuration)**.** *A configuration, denoted by* π*, is an assignment of values to the variables in Var. The set of all configurations is denoted by* Π_{Var}*, or* Π*, when Var is clear from the context. Define* $\Pi(\rho) = \{\pi \mid \pi \models \rho\}$ *to be the set of all those configurations that satisfy* ρ*. The elements of* $\Pi(\rho)$ *are called valid configurations. Given a valid configuration* π *and a transition* $t = (s,g,a,s')$*, we say that* t *is enabled by* π *if* $\pi \models g$*.*

As a concrete example of an FSMv, consider the feature *Door lock* in an automotive SPL which controls the locking of the doors of a car. The expected behaviour of this feature is modeled using the FSMv Req_{dl} described pictorially in Figure 4.1. In the initial state, this feature becomes active when all the doors are closed. The doors are locked when either the speed of the vehicle exceeds a predefined value or the gear is shifted out of park. An unlock event reactivates the feature.

There are three configurations for this feature all of which are described using the two variables: $Transmission_{dl} = \{Auto, Manual\}$ and $DL\_User\_Pref = \{Speed, Park\}$. The global predicate ($\rho$) associated with the machine is $\rho = \{Manual \rightarrow Speed\}$. To avoid clutter in the diagram, we write any atomic formula, $x = i$, simply as i, in the global predicate as well as in guards. ρ ensures that in every valid configuration, the variable $Transmission_{dl}$ having the value $Manual$ implies that DL_User_Pref takes the value $Speed$. This captures the fact that in manual transmission, there is no park position on the gearbox. The set of events

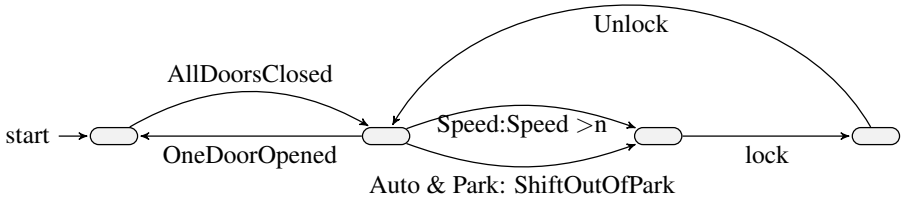

Figure 4.1: *Door lock* FSMv

for the state machine is $\{AllDoorsClosed$, $OneDoorClosed$, $ShiftOutofPark$, $Speed > n$, $Lock$, $Unlock\}$. In the state machine diagram, the transition labeled Auto&Park:ShiftOutofPark, denotes the fact that this transition is applicable only when DL_User_Pref is set to Park; and the transmission is automatic. the ShiftOutofPark denotes the event required for taking this transition.

Requirement against Design As we can see from the above example, in the requirement of a product line, the variability is usually discussed in terms of variation points, like Transmission Type, User Preference. These variation points are at a high level of abstraction and focused on clarity and expressibility. The restriction of the possible configurations is expressed as general constraints on these variation points, e.g., the global predicate $Manual \implies Speed$ in the *Door lock* example. In contrast, in a design, the variability description is constrained by efficiency, implementability, ease of reconfiguration and deployment considerations. For instance, in the automotive applications, one often finds a list of Boolean valued *calibration parameters* which are set appropriately at the time of deployment to obtain a specific variant. Further, the constraint on the calibration parameters (ρ) also takes the special form of the list of the possible configurations of the different calibration parameters in order to easily configure the design. The

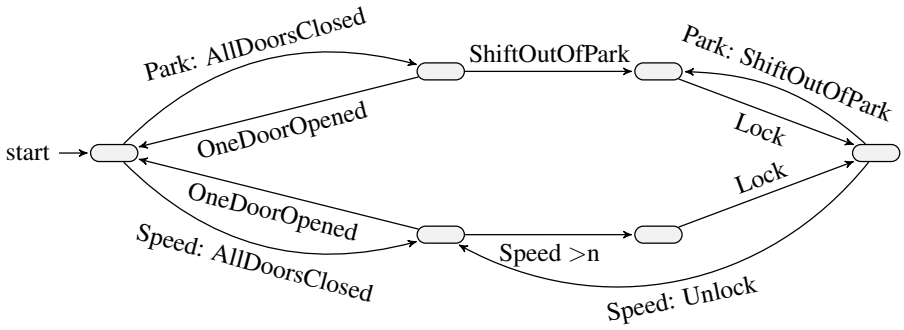

Figure 4.2: Des_{dl}: the FSMd abstracted from the design of the feature *Door lock*.

primary concern in the choice of calibration parameters and their constraints, is ease of deployability which constrains the number of variables used and the type of constraints.

FSMv can capture both the design as well as the requirements of a feature. We distinguish the requirement and design models by denoting them FSMr and FSMd respectively. Thus Figure 4.1 presents the FSMr, Req_{dl}, of the feature *Door lock*. Figure 4.2 describes FSMd, Des_{dl}, the design model of the *Door lock* feature.

The structure of Des_{dl} is similar to Req_{dl} except that the top elliptical shaped state in Figure 4.1 is split into two states (the top and the bottom elliptical shaped states) in Figure 4.2. The top state is for auto-transmission whereas the bottom one is for manual transmission as can be seen from the configuration label of the two transitions going from the initial state. One major difference to be noted in these two machines is the variability representation. One variable $Cp1 = \{Park, Speed\}$ encode the possible configurations in the FSMd. The box in Figure 4.2 depicts the set of possible values of these, *park* or *speed*, corresponding the the user preference but the transmission does not appear.

Variants of FSMv and Conformance Having described the design and requirement behaviour of a feature f, we now define the notions of variants and conformance. A variant of an FSMv corresponds to one of the several possible behaviors of the feature (at the design, requirement level respectively).

Definition 3 (Variant of an FSMv)**.** *Let $\mathcal{A} = \langle Q, q_0, \Sigma, Var, E, \rho \rangle$ be an FSMv and $\pi \in \Pi(\rho)$ be a valid configuration of $\mathcal{A}$. A variant of $\mathcal{A}$ is the standard finite state machine (FSM) obtained from the state machine of A, by retaining only the transitions $t = (s, g, a, s')$, and states s, s' such that $\pi \models g$. Once the relevant states and transitions are identified, we remove the guards g from all the transitions; The resultant FSM is denoted $\mathcal{A} \downarrow \pi$.*

In the example of FSMr for the feature *Door lock*, the variant $Req_{dl} \downarrow \langle Auto, Park \rangle$ does not contain the transition with the event $Speed > n$.

Given a feature f, and a (FSMd, FSMr) pair corresponding to f, we define a notion of *conformance*. We say that the design of f conforms to the requirements of f, iff every variant of the FSMd has a corresponding FSMr variant. Given an FSMv $\mathcal{A}$, we associate with each configuration π of $\mathcal{A}$ the language of the FSM $\mathcal{A} \downarrow \pi$, denoted by $L(\mathcal{A} \downarrow \pi)$. We say that an FSMd $\mathcal{A}_d$ conforms to an FSMr $\mathcal{A}_r$ if and only if the behaviour of every variant of $\mathcal{A}_d$ is contained in the behaviour of some variant of $\mathcal{A}_r$.

Definition 4 (The conformance mapping Φ)**.** *Let $\mathcal{A}_r$ and $\mathcal{A}_d$ be a pair of FSMr and FSMd respectively with global predicates ρ^r and ρ^d. Let Π_d, Π_r be the set of all design, requirement configurations. Then $\mathcal{A}_d$ conforms to $\mathcal{A}_r$ if there exists a mapping $\Phi : \Pi_d(\rho^d) \rightarrow 2^{\Pi_r(\rho^r)}$ as follows: For any $\pi_d \in \Pi_d(\rho^d)$, $\Phi(\pi_d) = \{\pi_r \in \Pi_r(\rho^r) \mid L(\mathcal{A}_d \downarrow \pi_d) \subseteq L(\mathcal{A}_r \downarrow \pi_r)\}$. Φ is called the conformance mapping, and the conformance via Φ is denoted $\mathcal{A}_d \leq_\Phi \mathcal{A}_r$.*

In the feature *Door lock*, $\Phi(\langle Speed \rangle) = \{\langle Manual, Speed \rangle,\ \langle Auto, Speed \rangle\}$ and $\Phi(\langle Park \rangle) = \{\langle Auto, Park \rangle\}$.

4.1.2 Modeling the behavior of a SPL

In general, an SPL contains multiple features. Each feature may bring its own variability and there could be constraints relating the variabilities among different features. The behavior of the whole system is a composition of the behavior of the features. There exists in the literature several kinds of compositions [85, 86]. Our focus in this report is variability and SPL modeling and hence we restrict ourselves to one of the well-known compositions, namely synchronous composition. By no means, our study is complete as far as the compositions are concerned. We believe that the issues related to the interaction of variability and composition will be similar to the one discussed here and the present work is a starting point for studying other compositions.

We define a parallel composition operator over FSMv to model an SPL. The features in an SPL can interact and we follow one of the standard methods of allowing the composed FSMv models to share some common events, which correspond to two-party handshake communication events. A distinguishing aspect of the proposed parallel operator is that it takes into account the constraints over variability information across the composed machines.

Definition 5 (Parallel composition of FSMv)**.**
Let $\mathcal{A}_i = \langle Q_i, q_0^i, \Sigma_i, Var_i, E_i, \rho_i \rangle$, $i \in \{1,2\}$ *be two FSMv's with* $Var_1 \cap Var_2 = \emptyset$. *Let* $H = \Sigma_1 \cap \Sigma_2$ *be the set of handshaking events. Let* ρ_{12} *be a predicate over* $Var_1 \cup Var_2$, *such that* $\rho_{12} \wedge \rho_1 \wedge \rho_2$ *is consistent.* ρ_{12} *is the composition predicate*

capturing the possible constraints between the variabilities of the two composed features. Let $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$.

The parallel composition of $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *denoted by* $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ *is a tuple* $\langle Q_1 \times Q_2, (q_0^1, q_0^2), \Sigma_1 \cup \Sigma_2, Var_1 \cup Var_2, E, \rho \rangle$ *with transitions defined as follows: Consider a state* $(s_1, s_2) \in Q_1 \times Q_2$, *and transitions* $(s_1, g_1, a_1, s_1') \in E_1$ *and* $(s_2, g_2, a_2, s_2') \in E_2$.
(1) If $a_1 = a_2 = a \in H$, *define* $((s_1, s_2), g_1 \wedge g_2, a, (s_1', s_2')) \in E$, *if* $g_1 \wedge g_2$ *is consistent. This transition is enabled under a valid configuration* $\pi \in \Pi(\rho)$, *such that* $\pi \models g_1 \wedge g_2$.
(2) If $a_1 \in \Sigma_1 \backslash H$, *define* $((s_1, s_2), g_1, a_1, (s_1', s_2)) \in E$. *This transition is enabled under valid configurations* π *such that* $\pi \models g_1$.
(3) If $a_2 \in \Sigma_2 \backslash H$, *define* $((s_1, s_2), g_2, a_2, (s_1, s_2')) \in E$. *This transition is enabled under valid configurations* π *such that* $\pi \models g_2$.

For illustration, consider the feature *Door unlock* which automates the unlocking of the doors in a vehicle. Figure 4.3-a gives the FSMr of the feature extracted from the requirements. From the initial state, the feature becomes active when the event *Lock* happens. As soon as either the key is removed from ignition or the gear is shifted to park position, the doors get unlocked and the feature *Door unlock* becomes inactive. Figure 4.3-b presents the FSMd of the feature *Door unlock*. It is quite similar to the requirement except that the active state is split in two: the feature reacts to the *ignition Off* event in one state, and to the *Shift Into Park* event in another state.

Let us consider the composition of the two FSMr's of the features *Door lock* and *Door unlock*. The handshake events between the two features are *Lock* and *Unlock*. In the composition, we introduce the following composition predicate:

$Transmission_{dl} = Transmission_{du}$, which brings out the natural constraints that the transmission status has to be the same. The valid configurations after composition are restricted by the composition predicate. We provide a few definitions to define composite valid configurations.

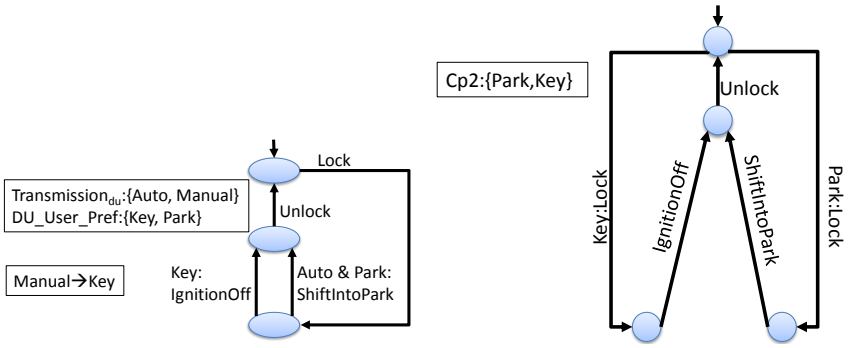

Figure 4.3: a) Req_{du}: *Door unlock* FSMr and b) Des_{du}: *Door unlock* FSMd.

Definition 6 (Composing Configurations)**.**

For $i = 1,2$, let $\mathcal{A}_i$=$(Q_i, q_0^i, \Sigma_i, Var_i, E_i, \rho_i)$ be two FSMv's, and let $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ be as given by Definition 5. Let $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$ be the global predicate of $\mathcal{A}$. Consider two valid configurations $\pi_1 \in \Pi(\rho_1)$ and $\pi_2 \in \Pi(\rho_2)$ of $\mathcal{A}_1$ and $\mathcal{A}_2$. The composition of π_1, π_2 denoted $\pi_1 + \pi_2$ is a configuration over $Var_1 \cup Var_2$ such that (i) $\pi_1 + \pi_2$ agrees with π_1 over Var_1, agrees with π_2 over Var_2, and (ii) $\pi_1 + \pi_2 \models \rho$.

Lemma 5. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two FSMv's. For each valid configuration π of $\mathcal{A}_1 \parallel \mathcal{A}_2$, there are valid configurations π_1 of $\mathcal{A}_1$ and π_2 of $\mathcal{A}_2$ such that $\pi = \pi_1 + \pi_2$.*

Proof. Let $\pi \in \Pi(\rho)$ with $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$ be a valid configuration of $\mathcal{A}_1 \parallel \mathcal{A}_2$. ρ_1 and ρ_2 are the global predicates of $\mathcal{A}_1$, $\mathcal{A}_2$ respectively, and ρ_{12} is the composition

predicate of $\mathcal{A}_1$, $\mathcal{A}_2$. By definition of valid configuration, $\pi \models \rho$; hence $\pi \models \rho_1$ and $\pi \models \rho_2$. Since π is a configuration over $Var_1 \cup Var_2$, let us consider the restriction of π on Var_1, call the resulting configuration π_1. Then $\pi_1 \models \rho_1$. Similarly, call the restriction of π on Var_2 as π_2. Then $\pi_2 \models \rho_2$. Then, π_1, π_2 are respectively valid configurations of $\mathcal{A}_1$ and $\mathcal{A}_2$. Hence, by definition 6, we obtain $\pi = \pi_1 + \pi_2$. □

In the example of feature *Door Lock*, the configuration $\langle Auto,\ Speed \rangle$ from Req_{dl} can be composed with $\langle Auto,\ Key \rangle$ from Req_{du} because the transmission is $Auto$ in both (which is specified in the composition predicate $Transmission_{dl} = Transmission_{du}$). $\langle Auto, Speed, Auto, Key \rangle$ is a configuration of the parallel composition of Req_{dl} with Req_{du}. The parallel composition of FSMv's is such that each variant of the composition of two FSMv's is equal to the composition of variants of the individual FSMv's.

Lemma 6 (Variants of a composed FSMv)**.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two FSMv's. Let π be a valid configuration of $\mathcal{A}_1 \parallel \mathcal{A}_2$. Then $L([\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi) = L(\mathcal{A}_1 \downarrow \pi) \parallel L(\mathcal{A}_2 \downarrow \pi)$.* [1]

We review some preliminary definitions before the proof. In the following, we denote by the same $\parallel$, the following operations: (i) shuffle of words, (ii) shuffle of languages, (iii)parallel composition of FSMs, and (iv) parallel composition of FSMv. The context would be clear in each case avoiding any confusion.

Definition 7. *Let $\Sigma_1, \ldots, \Sigma_n$ be n finite sets of symbols. Let Σ be a finite set. Given a word $w \in \Sigma^*$, we denote by $w \downarrow \Sigma_i$, the unique subword of w over Σ_i^*. For example, if $\Sigma_1 = \{a, b, e\}, \Sigma_2 = \{a, e, f\}$, and if we consider $w = aefedefr \in \{a, d, e, f, r\}^*$, then $w \downarrow \Sigma_1 = aeee$ and $w \downarrow \Sigma_2 = aefeef$.*

[1] The right hand side $\parallel$ refers to the standard communicating finite state machine composition.

Definition 8 (Asynchronous Shuffle)**.**

Let $\Sigma_1, \ldots, \Sigma_n$ be n finite sets. Let $\Sigma = \cup_{i=1}^{n} \Sigma_i$. Consider n words $u_1, u_2, \ldots, u_n$, $u_i \in \Sigma_i^*$. The asynchronous shuffle of $u_1, \ldots, u_n$ denoted $u_1 \parallel \cdots \parallel u_n$ is defined as $\{w \mid w \downarrow \Sigma_i = u_i\}$.

As an example, consider $\Sigma_1 = \{a,b,c,f\}, \Sigma_2 = \{a,d,e,f\}, \Sigma_3 = \{c,d,f\}$, and the words $u_1 = abcf, u_2 = adfe, u_3 = dcf$. Then the word $w = abdcfe$ is in $u_1 \parallel u_2 \parallel u_3$ since, $w \downarrow \Sigma_i = u_i$ for $i = 1,2,3$. Similarly, the word $w' = adbcfe$ is also in $u_1 \parallel u_2 \parallel u_3$. However, the word $w'' = aebcfd$ is not in $u_1 \parallel u_2 \parallel u_3$, since $w'' \downarrow \Sigma_2 = aefd$, not u_2.

The definition of shuffle can be extended from words to languages. We use the same notation $\parallel$ for the shuffle of sets, as well as for the shuffle of words.

The asynchronous shuffle of two languages L_1, L_2 is defined as $L_1 \parallel L_2 = \{w_1 \parallel w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. For example, if $L_1 = \{abcf, abbf\}$ is a language over $\Sigma_1 = \{a,b,c,f\}$ and $L_2 = \{adfe\}$ is a language over $\{a,d,e,f\}$, then $L_1 \parallel L_2 = \{abcf \parallel adfe, abbf \parallel adfe\}$ =$\{abcdfe, adbcfe, abdcfe, abbdfe, abdbfe, adbbfe\}$.

Definition 9 (Asynchronous product)**.**

Let $M_i = (Q_i, q_i, \Sigma_i, \delta_i)$ and $M_j = (Q_j, q_j, \Sigma_j, \delta_j)$ be complete FSMs. The asynchronous product of M_i, M_j is defined as the FSM $M_i \parallel M_j = (Q_i \times Q_j, (q_i, q_j), \Sigma_i \cup \Sigma_j, \delta)$ where

1. $\delta((q,q'),a) = (\delta_i(q,a), \delta_j(q',a)), a \in \Sigma_i \cap \Sigma_j$,
2. $\delta((q,q'),a) = (\delta_i(q,a), q'), a \in \Sigma_i, a \notin \Sigma_j$,
3. $\delta((q,q'),a) = (q, \delta_j(q',a)), a \in \Sigma_j, a \notin \Sigma_i$.

On the common events, both FSMs move in parallel; otherwise, they move independent of each other.

Proof. of Lemma 6:

It is known that $L(M_i \parallel M_j) = L(M_i) \parallel L(M_j)$. Consider a valid configuration π of $\mathcal{A}_1 \parallel \mathcal{A}_2$. As seen in Lemma 5, we can find valid configurations π_1 of $\mathcal{A}_1$ and π_2 of $\mathcal{A}_2$ such that $\pi = \pi_1 + \pi_2$. The initial state of $\mathcal{A}_1 \parallel \mathcal{A}_2$ is (q_0^1, q_0^2), where q_0^1 is the initial state of $\mathcal{A}_1$ and q_0^2 is the initial state of $\mathcal{A}_2$. By definitions 5 and 9, if we consider a string $w = a_1 a_2 \ldots a_n \in L[\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi$, then we can find strings $w_1 \in L(\mathcal{A}_1 \downarrow \pi) = L(\mathcal{A}_1 \downarrow \pi_1)$ and $w_2 \in L(\mathcal{A}_2 \downarrow \pi) = L(\mathcal{A}_2 \downarrow \pi_2)$ such that w can be written as an asynchronous shuffle of w_1 and w_2 in the sense of definition 8. Hence, $L[\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi \subseteq L(\mathcal{A}_1 \downarrow \pi) \parallel L(\mathcal{A}_2 \downarrow \pi)$. The converse can be shown in a similar way. □

Refinement and Parallel Composition The definition of parallel composition naturally lends itself to a notion of addition of conformance mappings between design and requirement pairs. Consider FSMr's R_1, R_2 corresponding to two features f_1, f_2. Let D_1, D_2 be the corresponding FSMd's. Let ρ_1^r, ρ_2^r be the global predicates of R_1, R_2, and let ρ_1^d, ρ_2^d be the global predicates of D_1, D_2 respectively. Assume that $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$. Let $\rho^r = \rho_{12}^r \wedge \rho_1^r \wedge \rho_2^r$ be the global predicate of $R_1 \parallel R_2$; likewise, let $\rho^d = \rho_{12}^d \wedge \rho_1^d \wedge \rho_2^d$ be the global predicate of $D_1 \parallel D_2$. We now want to ask if $D_1 \parallel D_2$ conforms to $R_1 \parallel R_2$. This amounts to computing a conformance mapping between $D_1 \parallel D_2$ and $R_1 \parallel R_2$ given Φ_1, Φ_2. Consider any valid configuration π^d of $D_1 \parallel D_2$. By Lemma 5, we can write π^d as $\pi_1^d + \pi_2^d$, where π_1^d, π_2^d are valid configurations of D_1, D_2 respectively. Since $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$, there exists valid configurations $\pi_1^r \in \Phi_1(\pi_1^d)$ and $\pi_2^r \in \Phi_2(\pi_2^d)$ in R_1, R_2 respectively. Given this, the addition of Φ_1, Φ_2 is defined as follows:

Definition 10 (Addition of conformance mappings)**.**

The addition of conformance mappings Φ_1, Φ_2 *is defined to be a mapping* $\Phi = \Phi_1 + \Phi_2$ *as follows. For every valid configuration* $\pi^d = \pi_1^d + \pi_2^d$ *of* $D_1 \parallel D_2$,

$$\Phi(\pi^d) = \{\pi^r \mid \pi^r \textit{ is a valid configuration of } R_1 \parallel R_2, \pi^r = \pi_1^r + \pi_2^r \textit{ for valid configurations } \pi_1^r \in \Phi_1(\pi_1^d), \pi_2^r \in \Phi_2(\pi_2^d)\}$$

Note that by Definition 10, Φ could be empty: Consider a valid configuration $\pi^d = \pi_1^d + \pi_2^d$ of $D_1 \parallel D_2$. If there is no valid configuration π^r of $R_1 \parallel R_2$ which is a composition of valid configurations $\pi_1^r \in \Phi_1(\pi_1^d), \pi_2^r \in \Phi_2(\pi_2^d)$, then Φ is empty (or there is no conformance mapping Φ between $D_1 \parallel D_2$ and $R_1 \parallel R_2$). If Φ exists, then we can say the following:

Lemma 7 (Conformance of composition)**.** *Let* R_1 *and* R_2 *be two FSMrs corresponding to features* f_1, f_2*, and let* D_1 *and* D_2 *be the corresponding FSMds. Let* $D_1 \leq_{\Phi_1} R_1$ *and* $D_2 \leq_{\Phi_2} R_2$*. Let* $\Phi = \Phi_1 + \Phi_2$ *and* π^d *be a valid configuration of* $D_1 \parallel D_2$*. Then,* $\forall \pi^r \in \Phi(\pi^d)$*,* $L([(D_1 \parallel D_2) \downarrow \pi^d]) \subseteq L([(R_1 \parallel R_2) \downarrow \pi^r])$.

Proof. Given a valid configuration π^d of $D_1 \parallel D_2$, we can write it as $\pi_1^d + \pi_2^d$, where π_1^d, π_2^d are respectively valid configurations of D_1, D_2 (Lemma 5). Since $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$, there exist valid configurations $\pi_1^r \in \Phi_1(\pi_1^d)$ and $\pi_2^r \in \Phi_2(\pi_2^d)$ such that $L(D_1 \downarrow \pi_1^d) \subseteq L(R_1 \downarrow \pi_1^r)$ and $L(D_2 \downarrow \pi_2^d) \subseteq L(R_2 \downarrow \pi_2^r)$.

Since Φ has been computed, for every valid configuration π^d of $D_1 \parallel D_2$, there exists some valid configuration π^r of $R_1 \parallel R_2$, $\pi^r \in \Phi(\pi^d)$. As π^r is valid, $\pi^r \models \rho_{12}^r \wedge \rho_1^r \wedge \rho_2^r$; hence, π^r can be written as $\pi_1^r + \pi_2^r$, where π_1^r, π_2^r are respectively valid configurations of R_1, R_2 (Lemma 5), and $\pi_1^r \in \Phi_1(\pi_1^d)$, $\pi_2^r \in \Phi_2(\pi_2^d)$ by definition 10.

$L([(D_1 \parallel D_2) \downarrow \pi^d]) = L(D_1 \downarrow \pi_1^d) \parallel L(D_2 \downarrow \pi_2^d)$ by lemma 6. Similarly, $L([(R_1 \parallel R_2) \downarrow \pi^r]) = L(R_1 \downarrow \pi_1^r) \parallel L(R_2 \downarrow \pi_2^r)$. This along with the observation that $L(D_1 \downarrow$

$\pi_1^d) \subseteq L(R_1 \downarrow \pi_1^r)$ and $L(D_2 \downarrow \pi_2^d) \subseteq L(R_2 \downarrow \pi_2^r)$ gives $L([(D_1 \parallel D_2) \downarrow \pi^d]) \subseteq L([(R_1 \parallel R_2) \downarrow \pi^r])$. □

The ECPL case study: First, let us remind that in the feature *Door lock*, $\Phi(\langle Park \rangle) = \{\langle Auto, Park \rangle\}$ and $\Phi(\langle Speed \rangle) = \{\langle Manual, Speed \rangle,\ \langle Auto, Speed \rangle\}$. Also, in the feature *Door unlock*, $\Phi'(\langle Key \rangle) = \{\langle Manual, Key \rangle,\ \langle Auto, Key \rangle\}$ and $\Phi'(\langle Park \rangle) = \{\langle Auto, Park \rangle\}$. Note that the conformance mapping of *Door lock* is noted Φ whereas the one of *Door unlock* is noted Φ'.

In the FSMr $Req_{dl} \parallel Req_{du}$ with $\rho_r : Transmission_{dl} = Transmission_{du}$.

Any configuration with $Transmission_{dl} = Auto$ and $Transmission_{du} = Manual$ is invalid. However, $\Phi(\langle Park \rangle)$ contains a single configurations where $Transmission_{dl} = Auto$ and $\Phi'(\langle Key \rangle)$ contains a configuration where $Transmission_{du} = Manual$ and $\Phi(\langle Park \rangle) + \Phi'(\langle Key \rangle)$ is a valid configuration of $Des_{dl} \parallel Des_{du}$ with $\rho_d : true$.

Does the design conform to the requirement? Yes because $\Phi'(\langle Key \rangle)$ contains another configuration where $Transmission_{du} = Auto$ and $\Phi(\langle Park \rangle) + \Phi'(\langle Key \rangle)$ is not an empty set: $\Phi(\langle Park \rangle) + \Phi'(\langle Key \rangle) = \{\langle Auto, Park, Auto, Key \rangle\}$. Similarly, $\Phi(\langle Park \rangle) + \Phi'(\langle Park \rangle) = \{\langle Auto, Park, Auto, Park \rangle\}$,
$\Phi(\langle Speed \rangle) + \Phi'(\langle Park \rangle) = \{\langle Auto, Speed, Auto, Park \rangle\}$, and
$\Phi(\langle Speed \rangle) + \Phi'(\langle Key \rangle) = \{\langle Auto, Speed, Auto, Key \rangle, \langle Manual, Speed, Manual, Key \rangle\}$.

It may append that the addition of conformance mappings leads to an invalid design configuration i.e. a design configuration of the composed machine that does not conform to any requirement configuration. Let assume a variant of the ECPL case study where in the feature *Door unlock*,
$\Phi(\langle Speed \rangle) = \{\langle Manual, Speed \rangle\}$ and $\Phi(\langle Park \rangle) = \{\langle Auto, Park \rangle\}$.

Also, in the feature *Door unlock*,
$\Phi'(\langle Key \rangle) = \{\langle Manual, Key \rangle\}$ and $\Phi'(\langle Park \rangle) = \{\langle Auto, Park \rangle\}$.

In this hypothetical case, $\Phi(\langle Speed \rangle) + \Phi'(\langle Park \rangle)$ and $\Phi(\langle Park \rangle) + \Phi'(\langle Key \rangle)$ are empty because the corresponding requirement configurations are incompatible with respect to the composition constraint: $Transmission_{dl} = Auto$ and $Transmission_{du} = Manual$.

So the design does not conform to the requirement. However, if we consider the composition predicate $\rho_d : Cp1 = Park \Leftrightarrow Cp2 = Park$, then $\langle Speed \rangle$ and $\langle Park \rangle$ are not compatible anymore ($\langle Park \rangle$ and $\langle Key \rangle$ neither) and as a result the design conforms to the requirement: $\Phi(\langle Park \rangle) + \Phi'(\langle Park \rangle) = \{\langle Auto, Park, Auto, Park \rangle\}$ and $\Phi(\langle Speed \rangle) + \Phi'(\langle Key \rangle) = \{\langle Manual, Speed, Manual, Key \rangle\}$.

4.1.3 Feature Level Verification

In the last section, we introduced FSMv for modeling the behavior of a feature and illustrated how they can express variability at two different levels of description. We also defined a conformance relation to formally relate the variabilities at two different levels of abstraction. In this section, we will give an automatic procedure for checking the conformance relation.

Let f be a feature whose requirement and design level descriptions are given by FSMr Req_f and FSMd Des_f, respectively. The proposed verification method, directly following the definition of conformance, computes a mapping Φ such that $Des_f \leq_\Phi Req_f$. A straightforward approach to computation is to identify for each design configuration, one or more requirement configuration. The conformance checking fails if there does not exist any requirement configuration for a design configuration.

Algorithm 1, given below, presents a possible implementation using the standard automata containment algorithm [87], as implemented in the well-known model checking tool SPIN [14]. Algorithm 1 runs the full verification algorithm of SPIN for every pair (π_d, π_r) of design and requirement configurations (Details about the SPIN encoding is given below). SPIN(i.e. *pan(.exe)*) returns the list of pairs for which the conformance condition is violated. Every other pair is added to the conformance mapping Φ.

Algorithm 1 implements the conformance checking using SPIN.

Input : Des_f, Req_f.

Output : The mapping Φ when $Des_f \leq_\Phi Req_f$

1. Generate a ProMeLa file which contains Req_f, Des_f, the environment, the conformance condition expressed as a *never claim*, and the initialization sequence.

2. Launch the full verification algorithm of SPIN

3. Build the mapping Φ from the output of SPIN.

4. Conclude whether the design conforms to the requirement

if $\forall \pi_d \in \Pi(\rho_d)$, $\Phi(\pi_d) \neq \emptyset$ **then**

  return $true$ along with (Φ)

else

  return $false$ along with (π_d) {where π_d has no correspondence through Φ}

end if

4.1.4 SPIN Encoding

In this section, we tried to explain input models to SPLEnD using $User\ Interface$ feature from Banking Software Product Line (BSPL). Figure 4.4 is the FSMr for feature $User\ Interface$, which has UI as an event with global predicate $\rho = \{\neg(uip = Disable)\}$. There is only one boolean variable, $Var = \{uip : \{Enable, Disable\}\}$.

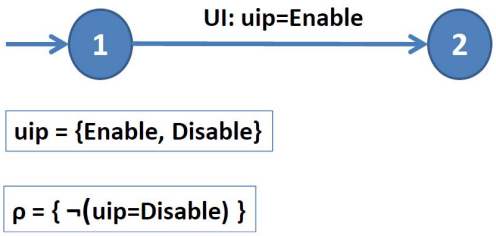

Figure 4.4: FSMr for feature: $UserInterface$.

Figure 4.5 is the FSMd for feature $User\ Interface$. This FSMd shares the event UI with the FSMr and has global predicate $\rho = \{(type = 2D \vee type = 3D)\}$. There are two variables, $Var = \{type : \{2D, 3D\}, graphics : \{Enable, Disable\}\}$.

SPLEnD represent any FSMvs in XML format. The XML file corresponding to the FSMr in Figure 4.4 is as below:

```
<?xml version='1.0'>
<FSMv>
<type>R</type>
```

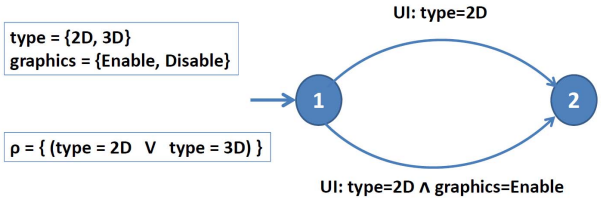

Figure 4.5: FSMd for feature: $UserInterface$.

```
<name>UserInterface_req</name>
<states>
<s initial=true>1</s>
<s>2</s>
</states>
<set_of_sets_of_final_states>
<group_of_final_states>
<s>2</s>
</group_of_final_state>
</set_of_sets_of_final_states>
<events>
<e>UI</e>
</events>
<variables>
```

```
<variable>
<v_name>uip</v_name>
<value>Enable</value>
<value>Disable</value>
</variable>
</variables>
<rho>
<conjunct>not(uip=Disable)</conjunct>
</<rho>
<vds>
<predicate>
<id>1</id>
<equ>(uip=Enable)</equ>
</predicate>
</vds>
<transitions>
<t>
<start>1</start>
<end>2</end>
<vdid>1</vdid>
<events>UI</events>
</t>
</transitions>
</FSMv>
```

The XML file for the FSMd in Figure 4.5 is as below:

```
<?xml version='1.0'>
<FSMv>
<type>D</type>
<name>UserInterface_des</name>
<states>
<s initial=true>1</s>
<s>2</s>
</states>
<set_of_sets_of_final_states>
<group_of_final_states>
<s>2</s>
</group_of_final_state>
</set_of_sets_of_final_states>
<events>
<e>UI</e>
</events>
<variables>
<variable>
<v_name>type</v_name>
<value>2D</value>
<value>3D</value>
</variable>
<variable>
<v_name>graphics</v_name>
```

```
<value>Enable</value>
<value>Disable</value>
</variable>
</variables>
<rho>
<conjunct>or(type=2D,type=3D)</conjunct>
</<rho>
<vds>
<predicate>
<id>1</id>
<equ>(type=2D)</equ>
</predicate>
<predicate>
<id>2</id>
<equ>and(type=3D,graphics=Enable)</equ>
</predicate>
</vds>
<transitions>
<t>
<start>1</start>
<end>2</end>
<vdid>1</vdid>
<events>UI</events>
</t>
<t>
```

```
<start>1</start>
<end>2</end>
<vdid>2</vdid>
<events>UI</events>
</t>
</transitions>
</FSMv>
```

These two XML files (one for FSMd and other for FSMr) along with predicate constraints file, are given as input to SPLEnD, which creates the ProMeLa model for the feature *User Interface*. Below is the format ProMeLa model create by SPLEnD to input SPIN.

```
#define  d_type_2D 0
#define  d_type_3D 1

#define  d_graphics_Enable 0
#define  d_graphics_Disable 1



#define  r_uip_Enable 0
#define  r_uip_Disable 1
```

```
/*The Events*/
#define  evt_UI 0


/*The states of the design model*/
#define  des_1 0
#define  des_2 1
#define des_error 2


/*The states of the requirement model*/
#define  req_1 0
#define  req_2 1
#define req_error 2



/*this channel is use to forward the event
in both des as well in req. from environment*/
chan evts_req= [0] of {byte};
chan evts_des= [0] of {byte};

/*State variable*/
byte req_state;
```

```
byte des_state;

/*Initialization  variables*/
byte vp_uip;
byte vp_type;
byte vp_graphics;
byte flag;



proctype environmentModel(){
do
::flag==0 -> flag=1; atomic{if
::(1)-> evts_des! evt_UI;   evts_req!evt_UI;
fi;}
od;
};


proctype requirementModel() {
mtype currentEvent;
req_state=req_1;
do
::flag==2-> evts_req?currentEvent;
if
```

```
::req_state== req_1-> if
::vp_uip==r_uip_Enable &&  currentEvent==
evt_UI-> req_state=req_2;
::else -> req_state= req_error;
fi;
::req_state== req_2-> if
::else -> req_state= req_error;
fi;
::else -> req_state = req_error;
fi;flag=0;
od;
};


proctype designModel() {
mtype currentEvent;
des_state=des_1;
do
::flag==1-> evts_des?currentEvent;
if
::des_state== des_1-> if
::vp_type==d_type_2D &&  currentEvent==
evt_UI-> des_state=des_2;
::(vp_type==d_type_3D && vp_graphics
==d_graphics_Enable)
```

```
&&  currentEvent== evt_UI-> des_state=des_2;
::else -> des_state= des_error;
fi;
::des_state== des_2-> if
::else -> des_state= des_error;
fi;
::else -> des_state = des_error;
fi;flag=2;
od;
};

/*never claim definintion*/
never {
TO_init:
if
::(flag==0 && req_state==req_error &&
des_state!=des_error)
->printf("vp_uip:%d,vp_type:%d,vp_graphics:%d\n",
   vp_uip,vp_type,vp_graphics);
goto accept_S9
::(1) -> goto TO_init
fi;
accept_S9:
if
::(1) -> goto TO_init
```

```
fi;
}

init{
flag=0; atomic{ if
:: (1)-> vp_uip=r_uip_Enable;
fi;}
atomic{
if
:: (1)->vp_graphics=d_graphics_Enable ;
vp_type=d_type_2D ;
:: (1)->vp_graphics=d_graphics_Disable ;
vp_type=d_type_2D ;
:: (1)->vp_graphics=d_graphics_Enable ;
vp_type=d_type_3D ;
:: (1)->vp_graphics=d_graphics_Disable ;
vp_type=d_type_3D ;
fi;}
atomic {
run environmentModel();
run requirementModel();
run designModel();
}
}
```

The above ProMeLa model is given as an input to SPIN. Internally SPIN generate

pan.c (c program) file corresponding to this ProMeLa model. SPIN execute pan.c to generate executable file called as pan.exe. pan.exe is executed to generate the conformance mapping. SPLEnD collect conformance mapping results from SPIN. Similarly, Φ is constructed for all other features.

After construction of Φ for all features, SPLEnD created the QBF for the composite FSMd's and FSMr's. This QBF is converted to 'qpro' format which is an input format for CirQit QBF solver. CirQit verified this QBF, and returned the results to SPLEnD.

Lemma 8 proves the correctness of Algorithm 1.

Lemma 8. *Given FSMd Des_f and FSMr Req_f for a feature f, let (π_d, π_r) be a pair of design and requirement configurations. Then, $L(Des_f \downarrow \pi_d) \not\subseteq L(Req_f \downarrow \pi_r)$ if and only if $\neg error(Des_f) \wedge error(Req_f)$.*

Proof. Assume $L(Des_f \downarrow \pi_d) \not\subseteq L(Req_f \downarrow \pi_r)$. Then there exists a word $w \in L(Des_f \downarrow \pi_d)$ which is prefixed by $u.e$, with u a finite prefix of a word in $L(Req_f \downarrow \pi_r)$, and e an event such that $u.e$ is not a prefix of any word in $L(Req_f \downarrow \pi_r)$. In such a situation, Des_f does not go to the error state but Req_f does.

Conversely, if $L(Des_f \downarrow \pi_d) \subseteq L(Req_f \downarrow \pi_r)$, then whenever Des_f is not in an error state, Req_f will also not be in an error state. In simple word, we found a word in $L(Des_f)$ which is not in $L(Req_f)$. Or in other words, we can say that, there is a path in design model which don't confirm with any of the path in requirement model. □

It must be noted that even though we are exhaustively checking whether every variant of the design conforms to some variant of the requirement. we are doing

it only at the feature level. Our experience with real systems is that the number of variants at the feature level is very small. Our experimental results shows that our approach scales well in realistic applications.

4.1.5 System Level Variability Verification

In this section, we extend the variability verification to a system consisting of two or more features. Let us consider the case when we have n features $f_1, f_2, \ldots, f_n$, whose designs and requirements are described by FSMd $D_1, D_2, \ldots, D_n$ and FSMr $R_1, R_2, \ldots R_n$ respectively. Further, let $D_i \leq_{\Phi_i} R_i$ for $1 \leq i \leq n$. Following Lemma 8, we can check that for each valid design configurations $\pi_{D_1 \| \cdots \| D_n}$ of $D_1 \parallel \cdots \parallel D_n$, the existence of a configuration $\pi_{R_1 \| \cdots \| R_n}$ of $R_1 \parallel \cdots \parallel R_n$ such that $\pi_{R_1 \| \cdots \| R_n}$ is a composition of valid requirement configurations computed via $\Phi_1, \ldots, \Phi_n$. We then have $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$ via a conformance mapping Φ which is nothing but $\Phi_1 + \cdots + \Phi_n$. The existence of conformance mapping can thus be shown by checking that Φ is non empty. This check is formulated naturally as satisfiability of a quantified boolean formula.

QBF Formulation : Given FSMd's $D_1, \ldots, D_n$ and FSMr's $R_1, \ldots, R_n$, (1) Let $Var(D_i) = \{x_{i1}, \ldots, x_{id_i}\}$ be the set of variables of design D_i, and $Var(R_i) = \{y_{i1}, \ldots, y_{ir_i}\}$, the set of variables of requirement R_i. Let $\pi_i^d : (x_{i1} = a_1, \ldots, x_{id_i} = a_{d_i})$ be a configuration of D_i. We denote this by $\pi_i^d(x_{i1}, \ldots, x_{id_i})$, which is the conjunction $\bigwedge_{l=1}^{d_i}(x_{il} = a_l)$;
(2) Given n FSMd's and n FSMr's check if D_i conforms to R_i for all $1 \leq i \leq n$ using Algorithm 1. This gives the map Φ_i. Assume D_i has m distinct configurations $\pi_{i1}^d, \ldots, \pi_{im}^d$. For $1 \leq j \leq m$, let $\Phi_i(\pi_{ij}^d) = \{\pi_{ij_1}^r, \ldots, \pi_{ij_k}^r\}$, where each of

$\pi^r_{ij_1}, \ldots, \pi^r_{ij_k}$ are configurations of R_i, that have been mapped by Φ_i to some configuration π^d_{ij} of D_i. $\Phi_i(\pi^d_{ij})$ can be written as the formula $\pi^r_{ij_1} \vee \cdots \vee \pi^r_{ij_k}$.
(3) The conformance mapping Φ_i between D_i and R_i has the form $\bigwedge_{j=1}^{m} \Phi_i(\pi^d_{ij})$.
(4) Let $\varphi^d_{i,j} = \rho^d \wedge \rho^d_i \wedge \rho^d_j$ and $\varphi^r_{i,j} = \rho^r \wedge \rho^r_i \wedge \rho^r_j$ represent respectively the propositional formula which ensure the consistency of the global predicates of D_i, D_j and R_i, R_j along with the compositional predicates ρ^d and ρ^r. Given a set $S \subseteq \{1,2,\ldots,n\}$, φ^d_S and φ^r_S can be appropriately written. For example, let $S = \{1,2,3\}$ then $\varphi^d_{1,2,3} = \rho^d \wedge \rho^d_1 \wedge \rho^d_2 \wedge \rho^d_3 \wedge \rho^d_{1,2} \wedge \rho^d_{1,3} \wedge \rho^d_{2,3}$
The QBF for conformance checking is given by

$$\Psi = \forall x_{11} \ldots x_{1d_1} x_{21} \ldots x_{2d_2} \ldots x_{n1} \ldots x_{nd_n} [\varphi^d_{1,2,\ldots,n} \Rightarrow \exists y_{11} \ldots y_{1r_1} y_{21} \ldots y_{2r_2} \ldots y_{n1} \ldots y_{nr_n} (\Phi_1 \wedge \cdots \wedge \Phi_n \wedge \varphi^r_{1,2,\ldots,n})]$$

The theorem below asserts that the QBF Ψ holds iff a conformance mapping Φ exists such that $D_1 \parallel \cdots \parallel D_n \leq_\Phi R_1 \parallel \cdots \parallel R_n$.

Theorem 9. *Given an SPL, let $\{f_1, \ldots, f_n\}$ be the set of features in a chosen product. Let D_i, R_i be the FSMd and FSMr for feature f_i. Then $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$ iff Ψ, as defined above, is satisfiable.*

Proof. Given $D_i \leq_{\Phi_i} R_i$, for $1 \leq i \leq n$, assume that $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$. Then, by definition of conformance, it means that for all valid configurations π^d of $D_1 \parallel \cdots \parallel D_n$, there exists a valid configuration π^r of $R_1 \parallel \cdots \parallel R_n$ such that $L([D_1 \parallel \cdots \parallel D_n] \downarrow \pi^d) \subseteq L([R_1 \parallel \cdots \parallel R_n] \downarrow \pi^r)$. Let Φ be the conformance mapping such that $\pi^r \in \Phi(\pi^d)$.

π^d is a valid configuration of $D_1 \parallel \cdots \parallel D_n$ implies that $\pi^d \models \bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho^d_S$, where ρ^d_S is the global predicate of $D_{i_1} \parallel \cdots \parallel D_{i_j}$, when $S = \{i_1, \ldots, i_j\}$. Using

Lemma 5 repeatedly, we can then say that $\pi^d = \pi^d_1 + \cdots + \pi^d_n$ for valid configurations π^d_i of D_i. Since $\pi^r \in \Phi(\pi^d)$, by definition of conformance mappings, π^r must be a valid configuration of $R_1 \parallel \cdots \parallel R_n$, hence $\pi^r = \pi^r_1 + \cdots + \pi^r_n$ (Lemma 5), such that $\pi^d_i \in \Phi(\pi^r_i)$, for valid configurations π^r_i of R_i. π^r is valid means $\pi^r \models \bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho^r_S$.

Given the above, we show that the QBF Ψ is satisfiable. The LHS of the QBF Ψ is the formula $\varphi^d_{1,2,\ldots,n}$, which is the conjunction ρ^d_S for all subsets S of $\{1,2,\ldots,n\}$. The *forall* quantifier outside would thus evaluate all configurations of $D_1 \parallel \cdots \parallel D_n$ that satisfy $\varphi^d_{1,2,\ldots,n}$; that is, which satisfy $\bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho^d_S$: hence, all valid configurations of $D_1 \parallel \cdots \parallel D_n$.

For the QBF to hold good, for all valid configurations of $D_1 \parallel \cdots \parallel D_n$ that have been evaluated on the LHS, we must find some configuration of $R_1 \parallel \cdots \parallel R_n$ that satisfies $\Phi_1 \wedge \cdots \wedge \Phi_n \wedge \varphi^r_{1,2,\ldots,n}$: (i) any configuration π of $R_1 \parallel \cdots \parallel R_n$ that satisfies $\varphi^r_{1,2,\ldots,n}$ would be valid; (ii) further, if it has to satisfy $\Phi_1 \wedge \cdots \wedge \Phi_n$, it must agree with $\pi^r_i \in \Phi_i(\pi^d_i)$ over $Var(R_i)$ for all $1 \leq i \leq n$. By Lemma 5, this means that π can be written as $\pi^r_1 + \cdots + \pi^r_n$. Thus, for the QBF to hold, we must be able to find for each valid configuration π^d of $D_1 \parallel \cdots \parallel D_n$, a valid configuration π^r of $R_1 \parallel \cdots \parallel R_n$ which can be written as $\pi^r_1 + \cdots + \pi^r_n$, where $\pi^r_i \in \Phi_i(\pi^d_i)$ for each i. But this is exactly what the mapping Φ which checks for conformance of $D_1 \parallel \cdots \parallel D_n$ with $R_1 \parallel \cdots \parallel R_n$ does. Since we assume that Φ exists, the QBF holds.

The converse can be shown in a similar way : that is, if the QBF Ψ holds, then $D_1 \parallel \cdots \parallel D_n$ will conform to $R_1 \parallel \cdots \parallel R_n$. $\square$

4.2 Experimental Results with SPLEnD

SPLEnD was used to verify an Entry Control Product Line (ECPL) that has 6-7 features and a Banking Software Product Line that has 100 features. Besides this, we experimented with several random SPLs with 5000 to 25000 features and all these were handled efficiently by SPLEnD.

We evaluated SPLEnD on a few case studies.

Entry Control Product Line (ECPL): This is an SPL from the automotive domain. ECPL had 7 features, each feature was modeled with an (FSMd, FSMr) pair with less than 10 states and 3-4 variables. The feature level conformance check was completed in all cases in less than or equal to 0.43 seconds. ECPL being a small case study, we tried checking conformance of the entire SPL by composing the individual FSMd, FSMr; this took 11 seconds to complete.

Banking Software Product Line (BSPL): This is an SPL controlling the software for ATM, Bank as well as mobile banking. This had 25 features. The FSMd, FSMr for all the features were modeled with less than 10 states and had 3-4 variables each. The conformance check was completed in all cases in less than or equal to 0.027 seconds; following this, SPLEnD synthesized the QBF to verify conformance at the SPL level. This check was completed in 0.005 seconds. QBF solving clearly outperforms the classical way of computing the product and checking for conformance, as can be seen in the ECPL versus BSPL case.

Scalability Check: Encouraged by the result from BSPL, we checked the scalability of our approach by generating random SPLs with 5000 to 25,000 features. The FSMd, FSMr in each case had 3 to 8 states and 2 variables each. For this experiment, we kept variables with binary domain only and this allow us to use them directly in QBFs. The conformance check at the feature level was completed in

less than 0.25 seconds in each case; the synthesized QBF formula had 10,000 (5k for design and 5k for requirement) to 50,000 variables. Let x denote the number of variables in the SPL design/requirement; Figure 4.6 shows the time taken in seconds in each case by SPLEnD.

x	10^4	2×10^4	3×10^4	4×10^4	5×10^4
Time(sec)	4.47	25.77	65.67	119.49	196.69

Figure 4.6: Execution time of QBF for Scalability

4.3 Modeling features and SPLs with time

Timed State Machine with variability: we propose a real-time extension of FSMv by permitting time-critical transitions. Following [16] we use clock variables to model time such that the clock variables grow uniformly within each state. Recall that we denote by $\mathcal{V}$ the finite set of variables used for modeling the variability. Without loss of generality, assume that the domain of each variable $x \in \mathcal{V}$ is finite and ranges over $\mathbb{N}$. Let $\text{Exp}(\mathcal{V})$ denote expressions of the form $\sum_{i=1}^{n} a_i x_i$ involving $x_i \in \mathcal{V}$ and $a_i \in \mathbb{Z}$. Let C be a finite set of clock variables. Let $c_i \in C$ and let $c \in \mathbb{N}$. The set $\Phi(C)$ of clock constraints can be inductively defined using the following grammar:

$$\varphi ::= c_i \sim \text{Exp} \mid c_i \sim c \mid \varphi \wedge \varphi$$

where $\sim \in \{<, >, \neq, =, \geq, \leq\}$. We are now in the position to define real-time extensions of FSMv.

Definition 10 (TSMv)**.** *A timed state machine with variability (TSMv) is a tuple* $\mathcal{A} = (L, s_0, \Sigma, \mathcal{V}, E, C, \rho)$ *where:*

- *L is a finite set of locations; s_0 is the initial location;*
- *Σ is a finite set of events;*
- *$\mathcal{V}$ is a finite set of variables;*
- *$E \subseteq L \times \Delta \times \Phi(C) \times R \times \Sigma \times L$ is the set of transitions;*
- *C is a finite set of clocks; and*
- *$\rho \in \Delta$ is a consistent global predicate.*

For a transition $e = (s, g, \varphi_g, R, a, s') \in E$ there is a transition from location s to location s' on event a; the predicate $g \in \Delta$ is called a guard of the transition e; g is consistent and defines the variability domain of the transition; $\varphi_g \in \Phi(C)$ is called the clock constraint associated with g; and $R \in 2^C$ is a set of clocks which are reset on transition e. The variables in $\mathcal{V}$ determine the variability allowed in the feature with each valuation of the variables corresponds to a variant.

An TSMv is deterministic iff for all locations l, whenever $e_1 = (l, g_1, \varphi, R_1, a, l_1)$ and $e_2 = (l, g_2, \psi, R_2, a, l_2)$ are two transitions from l, the clock constraints φ and ψ are mutually disjoint, that is $\varphi \wedge \psi$ is unsatisfiable. Generally, while capturing the requirements of a feature product line, the variability is at a high level of abstraction, and does not focus on implementability issues. The variability is expressed through constraints on the variation points. For example, in Figure 4.7, the global predicate $\neg(y \wedge z)$ restricts the possible configurations by a constraint on the variation points. On the other hand, in a design, the variability description is driven by efficiency, and ease of deployment. The level of abstraction in the variability is thus different in the requirement and the design of a product line. To capture this fact, we assume that the FSMv modeling the requirements as well as the design of a feature has disjoint variables and clocks. We distinguish the FSMv of the requirements and design by denoting them TSMv_r and TSMv_d respectively.

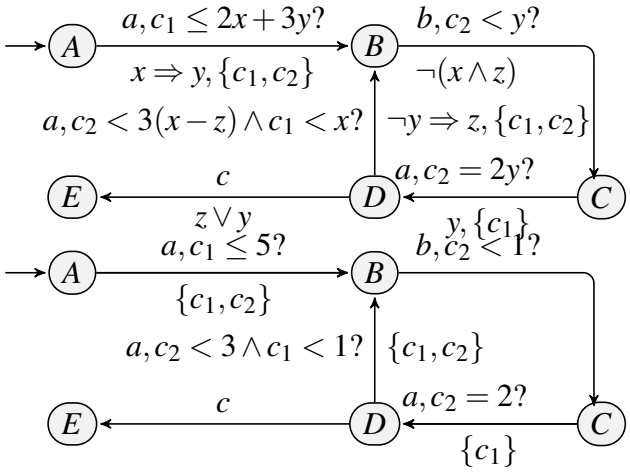

Figure 4.7: A TSMv $\mathcal{A}$ and its timed automata variant $\mathcal{A} \downarrow_{(x=1,y=1,z=0)}$

4.3.1 Design Conformance : Feature Level

Let $\mathcal{A}_i^d$ be the TSMv$_d$ modeling the behavior of the design of feature f_i, and let $\mathcal{A}_i^r$ be the TSMv$_r$ modeling the behavior of the requirements of feature f_i respectively. For simplicity, we assume that $\mathcal{A}_i^r$ is deterministic. Let $\mathcal{V}_i$ and C_i be the finite set of variables and clocks of $\mathcal{A}_i^d$, with $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ and $C_i \cap C_j = \emptyset$ for $i \neq j$. In a similar way, we assume that the variables and clocks of any two TSMv$_r$ are disjoint. Given $\mathcal{A}$, and a configuration π (an assignment of values to $\mathcal{V}$), we denote by $\mathcal{A} \downarrow \pi$, as in the untimed case, the variant of $\mathcal{A}$ with respect to π. In this case, $\mathcal{A}^d \downarrow \pi$ is a timed automata while $\mathcal{A}^r \downarrow \pi$ is a deterministic timed automata.

We say that an TSMv$_d$ $\mathcal{A}_d$ conforms to an TSMv$_r$ $\mathcal{A}_r$ if and only if the behavior of every variant of $\mathcal{A}_d$ is contained in the behavior of some variant of $\mathcal{A}_r$. This amounts to checking the language inclusion $A \subseteq B$, where A is a timed automata and B is a deterministic timed automata. This is known to be decidable [16].

Definition 11 (The conformance mapping Φ)**.** *Let $\mathcal{A}_r$ and $\mathcal{A}_d$ be a pair of TSMv$_r$*

and TSMv_d *respectively with global predicates* ρ^d *and* ρ^r. *Let* Π_d *and* Π_r *be the set of all design and requirement configurations. Then* $\mathcal{A}_d$ *conforms to* $\mathcal{A}_r$, *denoted* $\mathcal{A}_d \leq_\Phi \mathcal{A}_r$, *if there exists a mapping* $\Phi : \Pi_d(\rho^d) \rightarrow 2^{\Pi_r(\rho^r)}$ *such that* $\forall \pi_d \in \Pi_d(\rho^d), \exists \pi_r \in \Pi_r(\rho^r)$ *satisfying*

$$L(\mathcal{A}_d \downarrow \pi_d) \subseteq L(\mathcal{A}_r \downarrow \pi_r).$$

Φ *is called the conformance mapping.*

Note that the conformance mapping gives for each design configuration π^d, the set of all requirement configurations π^r such that the timed language $L(\mathcal{A}^d \downarrow \pi^d)$ is contained in the timed language $L(\mathcal{A}^r \downarrow \pi^r)$. The feature level conformance can thus be done by checking inclusion of the respective timed automata for design and requirement and computing the Φ mappings. As in the untimed case, we extend these mappings to check for conformance at the product line level. We first define the parallel composition of TSMv.

Consider the pairs $(\mathcal{A}_1^d, \mathcal{A}_1^r)$ and $(\mathcal{A}_2^d, \mathcal{A}_2^r)$ modeling the real time behavior of the design as well as the requirements of features f_1, f_2 respectively. Figure 4.8 depicts this. The variables used in $\mathcal{A}_1^d$ are $\{x_1, y_1\}$, along with clock variables $\{c_1, c_2\}$. The global predicate of $\mathcal{A}_1^d$ is $\rho_1^d = y_1 \Rightarrow x_1$. The variables used in $\mathcal{A}_2^d$ are $\{x_3, y_3\}$, along with clock variables $\{e_1, e_2\}$. The global predicate of $\mathcal{A}_2^d$ is $\rho_2^d = (x_3 \vee y_3) \wedge (y_3 \Rightarrow \neg x_3)$. The variables used in $\mathcal{A}_1^r$ are $\{x_2, y_2\}$, along with clock variable $\{d_1\}$. The global predicate of $\mathcal{A}_1^r$ is $\rho_1^r = \neg(x_2 \wedge y_2)$. The variables used in $\mathcal{A}_2^r$ are $\{x_4, y_4, z_4\}$, along with clock variable $\{f_1\}$. The global predicate of $\mathcal{A}_2^r$ is $\rho_2^r = (y_4 \Rightarrow \neg z_4) \wedge x_4$. It can be seen that $\mathcal{A}_1^d \leq_{\Phi_1} \mathcal{A}_1^r$ with the conformance mapping given by $\Phi_1[(1,1)] = \{(1,0)\}$, Note that $(0,1)$ is not a valid configuration of $\mathcal{A}_1^d$. Similarly, it can be seen that $\mathcal{A}_2^d \leq_{\Phi_2} \mathcal{A}_2^r$ with the conformance map-

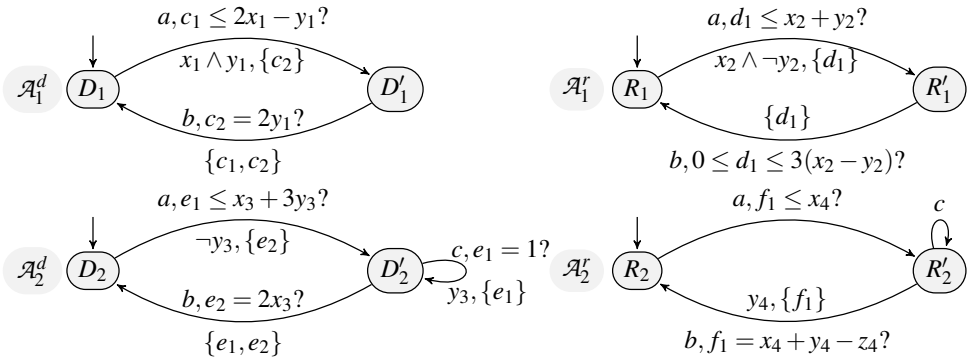

Figure 4.8: TSMv$_d$ $\mathcal{A}_i^d$ along with the corresponding TSMv$_r$ $\mathcal{A}_i^r$.

ping given by $\Phi_2[(1,0)] = \{(1,1,0)\}$, $\Phi_2[(0,1)] = \{(1,1,0),(1,0,1),(1,0,0)\}$. Note that (0,0) and (1,1) are not valid configurations of $\mathcal{A}_2^d$.

The parallel composition of two timed state machine with variability $\mathcal{A}_x = (Q_x, q_0^x, \Sigma_x, \mathcal{V}_x, E_x, C_x, \rho_x)$, $x \in \{1,2\}$ with $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$ and $C_1 \cap C_2 = \emptyset$, given a set $H = \Sigma_1 \cap \Sigma_2$ of handshake events is defined in the usual way. Let ρ_{12} be a predicate over $\mathcal{V}_1 \cup \mathcal{V}_2$, such that $\rho_{12} \wedge \rho_1 \wedge \rho_2$ is consistent. ρ_{12} is the composition predicate capturing the possible constraints between the variabilities of the two composed features. Let $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$. Transitions on H in the composition must respect the conjunction of guards on the respective transitions of $\mathcal{A}_x$, as well as the conjunction of clock constraints. Further, all valid configurations of the composition respect ρ. The composition of $\mathcal{A}_1, \mathcal{A}_2$ is denoted $\mathcal{A}_1 \parallel \mathcal{A}_2$. Consider two valid configurations $\pi_1 \in \Pi(\rho_1)$ and $\pi_2 \in \Pi(\rho_2)$ of $\mathcal{A}_1$ and $\mathcal{A}_2$. The composition of π_1, π_2, denoted π_{12} is a configuration over $\mathcal{V}_1 \cup \mathcal{V}_2$ such that π_{12} agrees with π_1 over $\mathcal{V}_1$, and agrees with π_2 over $\mathcal{V}_2$, and $\pi_{12} \models \rho$. π_{12} is a valid configuration of $\mathcal{A}$ and we denote it by $\pi_{12} = \pi_1 + \pi_2$. The parallel composition of TSMv's is such that each timed variant of the composition of two TSMv's is equal

to the composition of the timed variants of the individual TSMv's.

Lemma 12. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two TSMv's. For each valid configuration π of $\mathcal{A}_1 \parallel \mathcal{A}_2$, there are valid configurations π_1 of $\mathcal{A}_1$ and π_2 of $\mathcal{A}_2$ such that $\pi = \pi_1 + \pi_2$. Moreover,*

$$L([\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi) = L(\mathcal{A}_1 \downarrow \pi) \parallel L(\mathcal{A}_2 \downarrow \pi).$$

4.3.2 Refinement and Parallel Composition

The definition of parallel composition naturally lends itself to a notion of addition of conformance mappings between design and requirement pairs. Consider TSMv_r's R_1, R_2 corresponding to two features f_1, f_2. Let D_1, D_2 be the corresponding TSMv_d's. Let ρ_1^r, ρ_2^r be the global predicates of R_1, R_2, and let ρ_1^d, ρ_2^d be the global predicates of D_1, D_2 respectively. Assume that $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$. Let $\rho^r = \rho_{12}^r \wedge \rho_1^r \wedge \rho_2^r$ be the global predicate of $R_1 \parallel R_2$; likewise, let $\rho^d = \rho_{12}^d \wedge \rho_1^d \wedge \rho_2^d$ be the global predicate of $D_1 \parallel D_2$. $D_1 \parallel D_2$ conforming to $R_1 \parallel R_2$ amounts to computing a conformance mapping between $D_1 \parallel D_2$ and $R_1 \parallel R_2$, given Φ_1, Φ_2. Consider any valid configuration π^d of $D_1 \parallel D_2$. By Lemma 12, $\pi^d = \pi_1^d + \pi_2^d$, where π_1^d, π_2^d are valid configurations of D_1, D_2 respectively. Since $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$, there exists valid configurations $\pi_1^r \in \Phi_1(\pi_1^d)$ and $\pi_2^r \in \Phi_2(\pi_2^d)$ in R_1, R_2 respectively. Given this, the addition of Φ_1, Φ_2 is defined as follows: The addition of conformance mappings Φ_1, Φ_2 is defined to be a mapping $\Phi = \Phi_1 + \Phi_2$ as follows. For every valid configuration $\pi^d = \pi_1^d + \pi_2^d$ of $D_1 \parallel D_2$, $\Phi(\pi^d)$ is the set all valid configurations π^r of $R_1 \parallel R_2$, where $\pi^r = \pi_1^r + \pi_2^r$, and $\pi_1^r \in \Phi_1(\pi_1^d), \pi_2^r \in \Phi_2(\pi_2^d)$ are valid configurations of R_1 and R_2, respectively.

Lemma 13 (Composition Conformance)**.** *Let R_1 and R_2 be two TSMv_r machines*

corresponding to features f_1, f_2, and let D_1 and D_2 be the corresponding TSMv_d machines. Let $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$. Let $\Phi = \Phi_1 + \Phi_2$ and π^d be a valid configuration of $D_1 \parallel D_2$. Then, $\forall \pi^r \in \Phi(\pi^d)$, $L([(D_1 \parallel D_2) \downarrow \pi^d]) \subseteq L([(R_1 \parallel R_2) \downarrow \pi^r])$.

Let us now revisit the $(\mathsf{TSMv}_d, \mathsf{TSMv}_r)$ pairs seen in Figure 4.8. Consider $\mathcal{A}_1^d \parallel \mathcal{A}_2^d$ along with the global compositional predicate $\rho_{1,2}^d = y_1$, and $\mathcal{A}_1^r \parallel \mathcal{A}_2^r$, along with the global compositional predicate $\rho_{1,2}^r = x_2 \wedge (x_2 \Rightarrow y_4)$. It can be seen that $L(\mathcal{A}_1^d \parallel \mathcal{A}_2^d)$ is same as $L(\mathcal{A}_1^d)$ and $L(\mathcal{A}_2^d)$ for the configuration $(x_1 = 1, y_1 = 1, x_3 = 1, y_3 = 0)$, and is the empty set for all other valid configurations. Similarly, the parallel composition $L(\mathcal{A}_1^r \parallel \mathcal{A}_2^r) \supseteq L(\mathcal{A}_1^r)$ and $L(\mathcal{A}_2^r)$. The configurations $(x_2 = 1, y_2 = 0, x_4 = 1, y_4 = 0, z_4 = 0)$ and $(x_2 = 1, y_2 = 0, x_4 = 1, y_4 = 0, z_4 = 1)$ are not valid for $\mathcal{A}_1^r \parallel \mathcal{A}_2^r$, thanks to the compositional predicate $\rho_{1,2}^r = x_2 \wedge (x_2 \Rightarrow y_4)$. It can be seen that $\mathcal{A}_1^d \parallel \mathcal{A}_2^d \leq_\Phi \mathcal{A}_1^r \parallel \mathcal{A}_2^r$, where Φ is given by $\Phi[(x_1 = 1, y_1 = 1, x_3 = 1, y_3 = 0)] = \{x_2 = 1, y_2 = 0, x_4 = 1, y_4 = 1, z_4 = 0\}$, $\Phi(x_1 = 1, y_1 = 1, x_3 = 0, y_3 = 1)] = \{(x_2 = 1, y_2, x_4 = 1, y_4 = 1, z_4)\}$. Note that Φ is indeed the composition of Φ_1 and Φ_2, respecting the compositional predicates $\rho_{1,2}^d \wedge \rho_{1,2}^r$.

4.3.3 Conformance Checking

Lemma 13 shows how to extend the conformance mappings of individual features to multiple features, by pairwise composition. Assuming we have two features f_1, f_2, along with appropriate constraints on how the features interact when composed. Let R_i be the the TSMv_r modeling the expected behavior and variability of f_i, and D_i the TSMv_d extracted from the design of f_i. Let ρ_{12}^r and ρ_{12}^d be the compositional predicates for $R_1 \parallel R_2$ and $D_1 \parallel D_2$ respectively. Now we state the variability conformance problem for the composition as follows: Does there exist

a conformance mapping Φ such that $D_1 \parallel D_2 \leq_\Phi R_1 \parallel R_2$?

A compositional approach to solve the problem is to:

1. check if the design of f_i conforms to its requirement;
2. check if every valid configuration of $D_1 \parallel D_2$ can be mapped to a valid configuration of $R_1 \parallel R_2$.

This is the conformance condition.

Given the global predicates ρ_1^d, ρ_2^d as well as compositional global predicate $\rho_{1,2}^d$, let $\varphi_{1,2}^d = \rho_1^d \wedge \rho_2^d \wedge \rho_{1,2}^d$ be the global compositional predicate governing the composition of the TSMv_d D_1, D_2. In a similar way, we can define $\varphi_{1,2}^r$, the global compositional predicate governing the composition of TSMv_r R_1, R_2. Having computed the conformance mappings Φ_1, Φ_2 at the feature level, we can check the truth of the QBF formula

$$\Psi \stackrel{\text{def}}{=} \forall X[\varphi_{\{1,2\}}^d \Rightarrow \exists Y(\Phi_1 \wedge \Phi_2 \wedge \varphi_{\{1,2\}}^r)],$$

where X is the set of all the variables in D_1, D_2 while Y is the set of all the variables in R_1, R_2. Recall that the variables across any pair of TSMv_d or TSMv_r are disjoint. Hence, the satisfaction of pairwise conformance, along with compositional constraints ensure the existence of a solution. The QBF approach extends to arbitrary number of designs and requirements : $\varphi_S^d = \bigwedge_{i \in S} \rho_i^d \wedge \bigwedge_{i,j \in S}^d \rho_{i,j}^d$ captures the global compositional predicate for S.

Theorem 14. *Given a product line consisting of n features $\{f_1, \ldots, f_n\}$, let D_i, R_i be the TSMv_d and TSMv_r for feature f_i. Then $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$ if and only if Ψ holds.*

Power Window Controller Modeling: Figure 4.9 illustrates the modeling of an automotive example, the power window controller PWC [88] at requirement level. PWC, in response to user requests, controls the movement of one or more windows. The user can request basic up/down movement (*up, down*) of the windows, as well as express up and down movement (*Xup, Xdn*) which moves the windows to fully closed or open positions; these latter event happens when the user releases the up/down request before a fixed unit of time (*xpTR*). The Xup and Xdn are optional. The controller has an option of detecting an obstacle (like finger or hand) whenever it has the express up option, and retracting the window. The controller also checks the events of reaching the extreme ends of the window positions (*top, bottom*) before a fixed duration (*closeTR*). The variability in the model is expressed using the variables *ob, exp, xpTR, closeTR* where ob captures the presence/absence of obstacles detection feature, *exp* capture whether the feature express up/express down are active and d is the clock variable for measuring the elapse of time. $xpTime$ is a fixed unit time for up/down request. $closeTime$ is a fixed unit time to fully close or open the window. The global predicate $\rho = exp \rightarrow ob$ (means the presence of express up/down feature require the presence of obstacle detection feature), constraints the variability allowed in the system. Similarly, figure 4.10 represent the modeling of the PWC at design level.

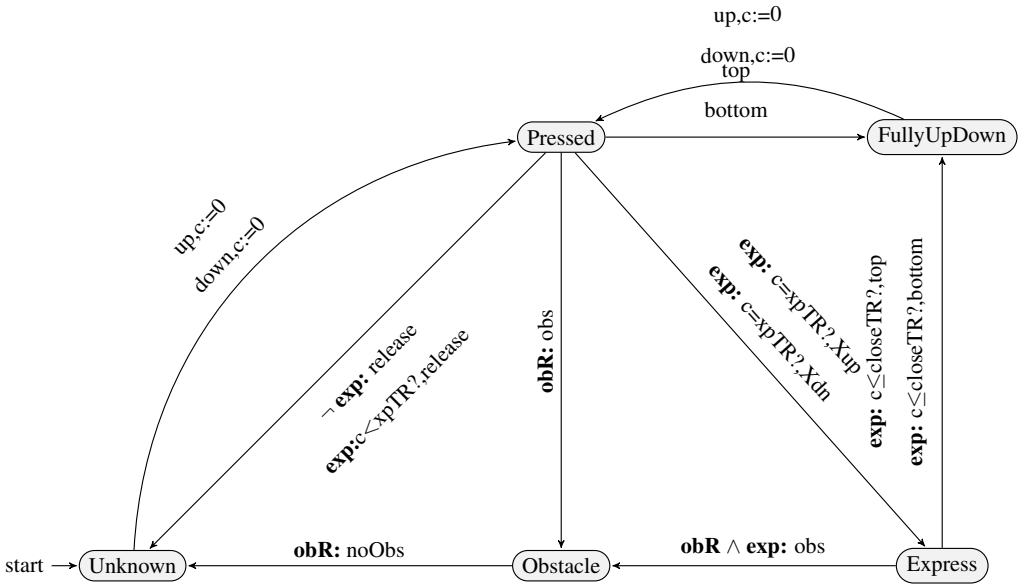

Figure 4.9: TSMv$_r$ for power window controller

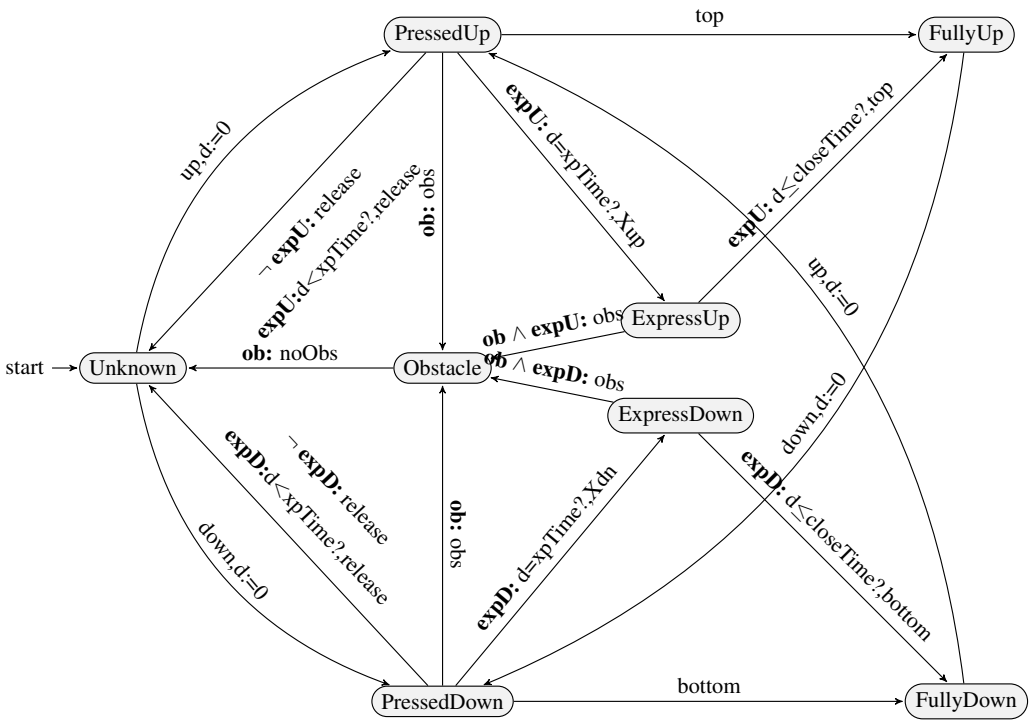

Figure 4.10: TSMv$_d$ for power window controller

Chapter 5

Tools

5.1 SPLAnE Tool

The Software Product Line (SPL) paradigm aims to jointly design a family of closely related software *products* in an efficient and cost-effective manner. In an SPL, there is a collection of features called the *scope* and a collection of reusable components called *core assets*, which are developed once for the entire product line family. Subsets of features from the scope specify the possible products in the family, which are then implemented by subsets of components from the core assets. The various products arising out of an SPL are specified as *variations* of one another. Some of these variability constraints [82], [5] are (i) specified using cross-tree constraints between features (components), like *include, exclude* relations, or (ii) by specifying whether a feature (component) is mandatory/optional. A mapping between the features and sets of components implementing them is called a *traceability relation*. A subset of features satisfying the feature level variability constraints is called a *specification*; the set of all specifications is called

a *PL specification*. Likewise, an *implementation* is a set of components satisfying the component level variability constraints; the set of all implementations is called a *PL implementation*. The SPL framework is defined [82] as a 3-tuple entity $\langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, where $\overline{\mathcal{F}}$ is a PL specification, $\overline{\mathcal{C}}$ is a PL implementation and $\mathcal{T}$ is the traceability relation. Integrating the variability across the specification and implementation levels and analyzing an SPL using the traceability relation is a challenging task : for instance, how do we check whether the *PL implementation* is adequate for providing implementation for all the specifications in the *PL specification*? In this report, we introduce a tool SPLAnE that is capable of analyzing several important and useful analysis operations on SPLs (see table 3.7 for a list of analysis operations).

SPLAnE (Software Product Line Analysis Engine) is designed and developed to analyze SPLs. Most of the existing state of the art tools in the product line domain (FaMa [37], FeatureIDE [89], FLAME [90], FAMILIAR [91] and SPLAR [81]) provide analysis operations based only on the PL specifications (given in the form of Feature Model/Orthogonal Variability Models/Domain-Specific Language). None of these tools have the capability to reason over PL specifications and PL implementations together.

FAMILIAR presents a Domain-Specific Language (DSL) which is used for management of large scale feature models and provides a script for reasoning over it [91]. *FeatureIDE* is designed as an IDE for the AHEAD tool suite [92], which integrates activities of all phases of software product line engineering. It uses the Feature-Oriented Software Development (FOSD) [92] approach for designing and implementing applications based on features. It is more about managing feature models visually, and no analysis over SPLs is provided. *FLAME* is a formal

framework for the specification of analysis operations on feature models [90]. It uses the Z specification language to encode feature models in Prolog. *FLAME* has automatically validated 18,000 test cases automatically generated using metamorphic testing techniques. *SPLAR* is the reasoning mechanism of SPLOT [81]. SPLOT is a web application for creating feature models online. *SPLAR* provides very few (3) analysis operations, namely *number of products*, *dead features* and *valid*.

SPLAnE takes as inputs (i) a PL specification given as a feature model, (ii) a PL implementation given as a component model, (similar to feature models, introduced in [82] for representing PL implementations), and (iii) a traceability relation. SPLAnE then parses the feature model, the component model and the traceability relation and synthesizes the SPL model. The user can now select any analysis operation (SPLAnE has 23 available ones). Based on the choice of the analysis operation, SPLAnE constructs a QBF formula Ψ. The formula Ψ constructed is such that Ψ is satisfied iff the chosen analysis operation holds good on the SPL model. SPLAnE then invokes the QSAT solver CirQit [35] in the back-end to check the satisfiability of Ψ. Ψ is encoded in the QPRO format, the standard input file format in non-prenex, non-CNF form for CirQit. Figure 5.1 shows the reasoning process in SPLAnE .

5.1.1 Features of SPLAnE

1. ***SPLAnE* as an extension of FaMa**: FaMa is the state of the art analysis engine used in the industry for feature models analysis. The FaMa framework is the core part of the FaMa tool which provides generic SPL related APIs which allow the inheritance of all the basic functionalities of FaMa

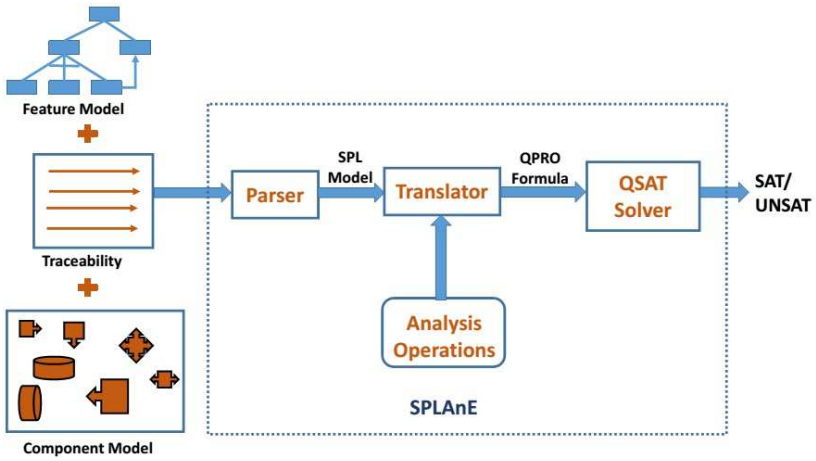

Figure 5.1: SPLANE reasoning process

to any extension. For example, FaMa framework provides support for reasoning over orthogonal variability models [76] and Debian based distributions [93]. Because of this, we chose the FaMa framework [94] as base for SPLAnE . The FaMa framework provides out of the box basic architecture for building feature model analysis tools while defining interfaces and standard implementation for some feature model operations[5] such as *"VoidFM"* or *"Products"*. From FaMa Tool, we specifically used feature model creater, reader and parser module in SPLAnE . On the other hand, FaMa does not provide any reasoning mechanism that takes as input more than one model. Therefore, different modifications have been addressed to bridge this gap. Concretely, i) we modified the FaMa framework to enable the triple (feature model,component model, traceability relation) as input ii) created a new module in FaMa framework for QSAT reasoners; iii) implemented several new analysis operations and re-implemented existing FaMa analysis operations, and iv) defined two new file formats to store and input

traceability relations and component models in SPLAnE .

Being part of the FaMa ecosystem allowed the reutilisation of some FaMa artefacts. This also allowed reutilisation of other FaMa ecosystem tools such as the FaMaTestSuite[95] and Betty [95] for detecting coding errors on SPLAnE . For the sake of re-utilization and coding efforts optimization, SPLAnE has been built on top of the FaMa framework. Like FaMa, multiple QSAT solvers can be plugged in SPLAnE . Thanks to the FaMa framework, SPLAnE can be (i) easily integrated (FaMa framework has a java interface, implementing a query based interaction), (ii) easily extendible, and (iii) easy to configure (FaMa fraework is configured by means of an unique XML file, easing its maintenance and configuration to adapt the tool to the user needs).

2. ***Analysis operations***: SPLAnE provides 23 analysis operations when reasoning over feature models, component models and traceability relations at the same time. That apart, SPLAnE can reason over a large set of existing analysis operations [5] using only feature models. For these operations on feature models, SPLAnE extends the existing FaMa framework adding QSAT reasoning capabilities.

3. ***Multiple File Formats***: One of the benefits of using FaMa framework is the reusability of code. In this case, the FaMa ecosystem enables SPLAnE to read and write into the most used file formats in the feature modelling area. Among others, FaMa is able to read different feature diagram notations such as xml or plain text, thus, enabling SPLAnE to cooperate with other feature modelling ecosystems. For example, SPLAnE can reason over the feature models stored in the splot[81] feature model repository by using the *.splx*

format.

4. ***Multiple Reasoners***: Depending on the analysis operation we are executing, different solvers may perform better. For example, for counting the number of products (this operation is called *#products*), a BDD based solver such as javaBDD [96] will perform faster than a CSP solver. Among other solvers, FaMa enables the use of SAT solvers for the reasoning of feature models while providing a concrete reasoning method that uses *SAT4j* as a solver. As an inherited capability, SPLAnE can plug and unplug various reasoners, thereby, using the best solver for each operation. An appealing use of this characteristic is to compare SAT vs QSAT for each analysis operation.

5. ***Scalability***: The QBF encoding easily outperforms the SAT approach. It must be noted that SPLAnE is the first tool that performs SPL anlaysis operations using the QBF encoding; all the previous tools use only the SAT encoding.

5.2 SPLEnD Tool

The requirements and designs which is input to SPLEnD are a kind of state machines, called *Finite State Machines with Variability (FSMv)* which is an extension of finite state machines, introduced in [97]. FSMv is capable of capturing variability information and is used to model the behavior and variability of a feature at the requirement and design level. The FSMv that models the design of a feature is called an FSMd, while the FSMv that models the requirements is called an FSMr. The theory of a tool and the modeling of SPLs is explained in [97].

Figure 5.2 summarizes the approach followed in SPLEnD. It shows an SPL composed of features f_1 to f_n. Each feature has an FSMv model of its requirements (FSMr) and an FSMv model derived from its design (FSMd). SPLEnD first checks whether the FSMd of every feature conforms to its FSMr (1^{st} check). The output of this step is a conformance relation between each pair of FSMr and FSMd. The obtained conformance relations are then used to check whether the design of the entire SPL conforms to the requirements (2^{nd} check). The first step is a standard model checking step where SPLEnD uses SPIN [14]. For the second step, SPLEnD constructs a Quantified Boolean Formula(QBF) and uses the QBF solver CirQit [15]. While the first step explicitly enumerates all feature variations and filters only those which satisfy the requirements, the second step avoids the enumeration explicitly and captures symbolically all the feature combinations using the QBF formula.

Figure 5.3 describes the overall architecture of SPLEnD. It takes as input the FSMds and FSMrs of a list of features along with the composition constraints at the requirement and design levels. The first check is delegated to SPIN [14] and the second check is done by the QBF solver CirQit [15]. The users of SPLEnD load an SPL project in the tool, which consists of two lists of XML files corresponding to the designs (FSMd) and requirements (FSMr) of all the features in the SPL. On loading the project, SPLEnD displays the table of (FSMd, FSMr) pairs, of all the features. Since 1-to-1, 1-to-*n*, *n*-to-1, and *n*-to-*m* mappings are possible between features of the design and features of the requirement, a *.property* file is given to express the mapping of each feature or group of features. When a mapping involves a group of features, this group is defined by i) a name, ii) a list of plain FSMvs that compose the group and a *group level composition constraint* that

limit the possible configurations of the group in order to maintain consistency.

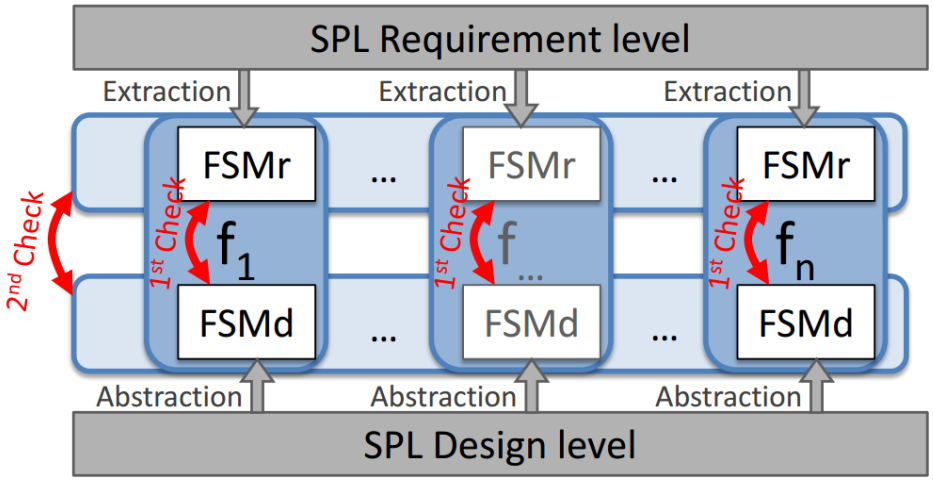

Figure 5.2: SPLEnD's framework and approach.

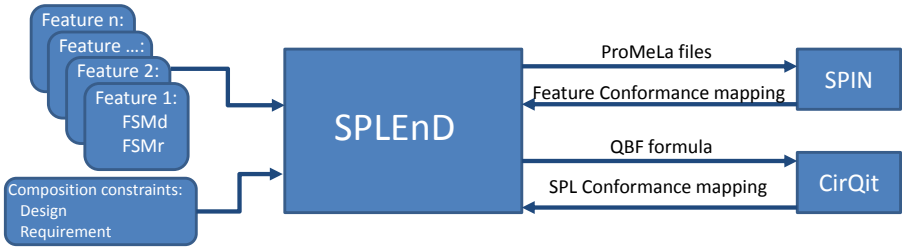

Figure 5.3: Architecture of SPLEnD

5.2.1 Feature Level Verification(1st Check)

On selection of a particular (FSMd, FSMr) pair in the table, users can see the FSMr and FSMd corresponding to each feature (Figure 5.5) and compute the conformance mapping. In Figure 5.6, SPLEnD displaying the conformance mapping for a pair of (FSMd,FSMr) for the case study ECPL. The time taken for the same is also shown. In case there is a non-conformance between a (FSMd, FSMr) pair,

the list of illegal configurations are displayed; the user can also see the FSMs corresponding to the violation, these FSMs are shown with their edges in red color.

5.2.2 SPL Level Variability Verification(2nd Check)

Once the conformance checking is successfully completed for all features independently (1st check), the next step is to check the conformance at the SPL level.

The SPL level variability is usually defined with a feature diagram [41] bringing constraints across features. These compositions constraints limit the possible configurations of the SPL at the requirement level and at the design level. Thus the SPL level conformance checks that every valid SPL design configuration conforms to a valid SPL requirement configuration.

To do so in SPLEnD,the user clicks the *SPL Conformance* button; this checks in one pass whether the design of all products in the SPL conform to the requirements. SPLEnD displays the result of the check as a YES/NO answer and the time taken for the same in the lower part of the window, see Figure 5.7. In case of non-conformance, SPLEnD gives a configuration of the design of the SPL that does not conform to any requirement's configuration.

5.3 Extra features of SPLEnD:

5.3.1 Addition of Features

The user can add features to an existing SPL by simply adding the XML files corresponding to the design and requirement of the new feature on the appropriate folders. The project needs to be reloaded, and conformance checking will be

carried out for the new (FSMd, FSMr) pair. Figure 5.4 shows the GUI when the SPL model is loaded in SPLEnD.

5.3.2 Converting predicate

One can give to SPLEnD a predicate defined using the vocabulary from the requirement level and SPLEnD translates it to the equivalent predicate at the design level using the individual conformance mapping. The purpose of this extra feature is to allow expressing properties (for model checking) at the requirement level where the variability is expressed naturally and get the automatic conversion to the design level vocabulary in order to check this property on the design code as shown in figure 5.8.

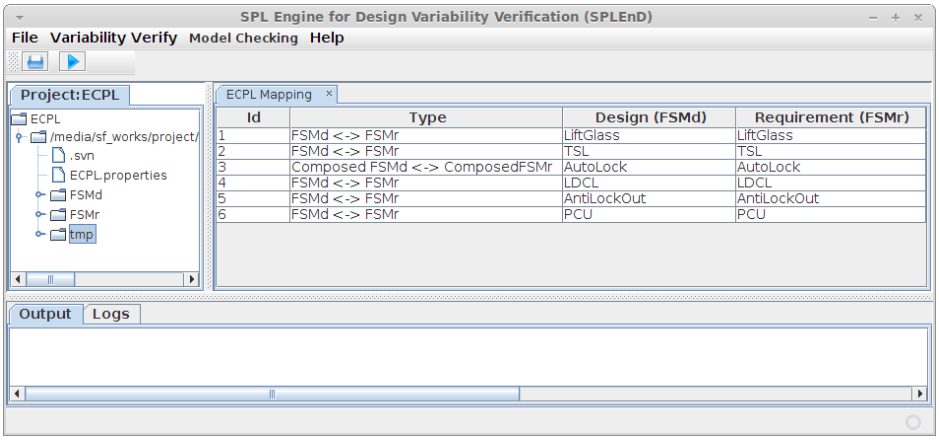

Figure 5.4: Snapshot of SPLEnD: Loading a project

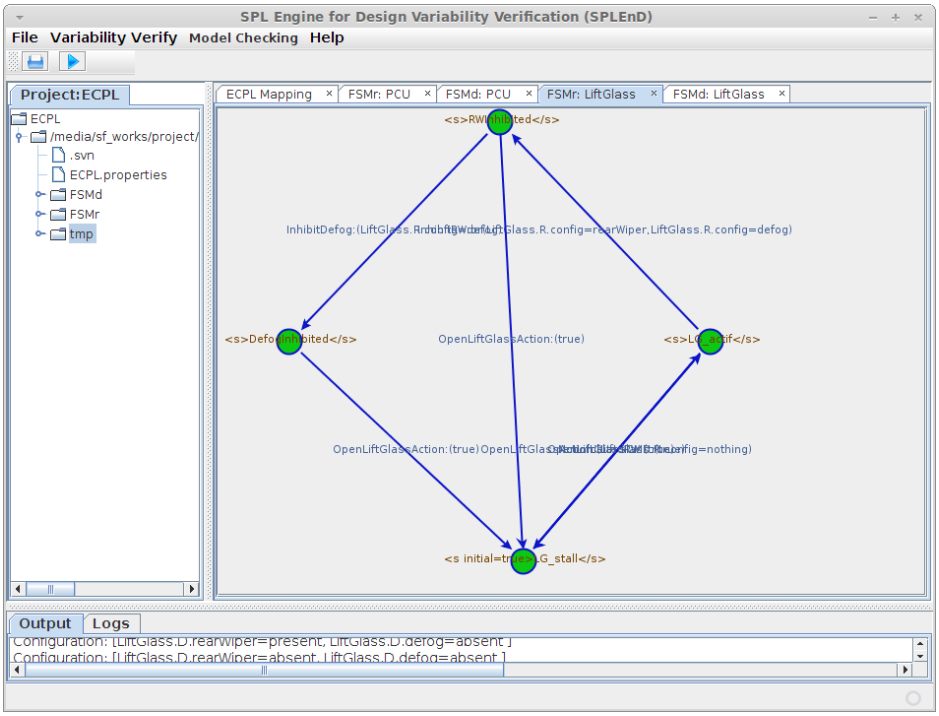

Figure 5.5: Snapshot of SPLEnD: The FSMr of the feature LiftGlass

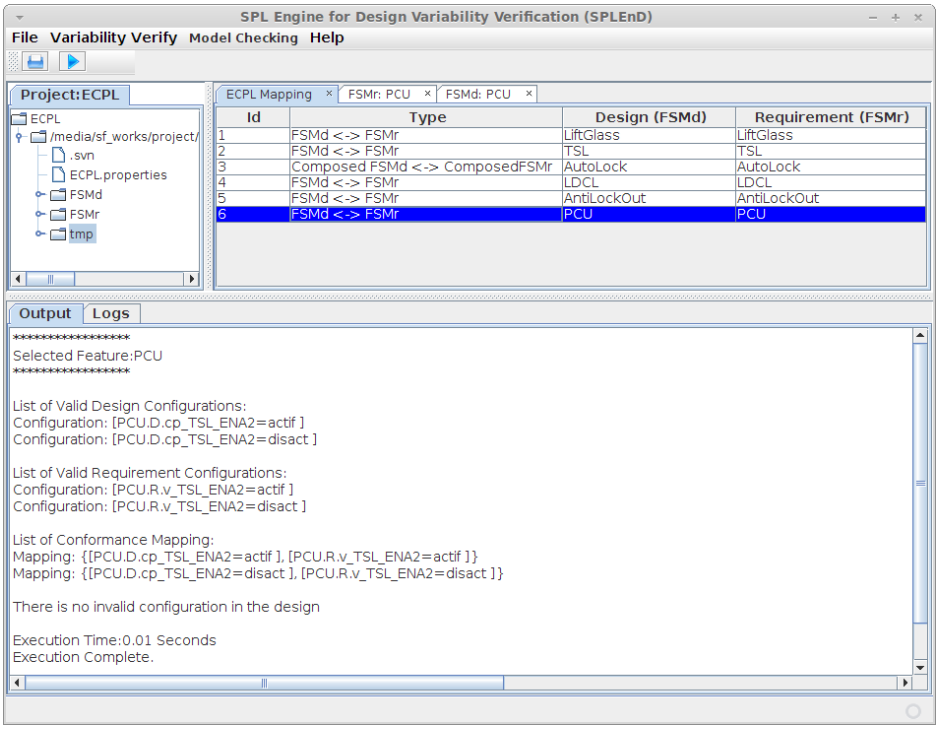

Figure 5.6: Snapshot of SPLEnD: conformance mapping

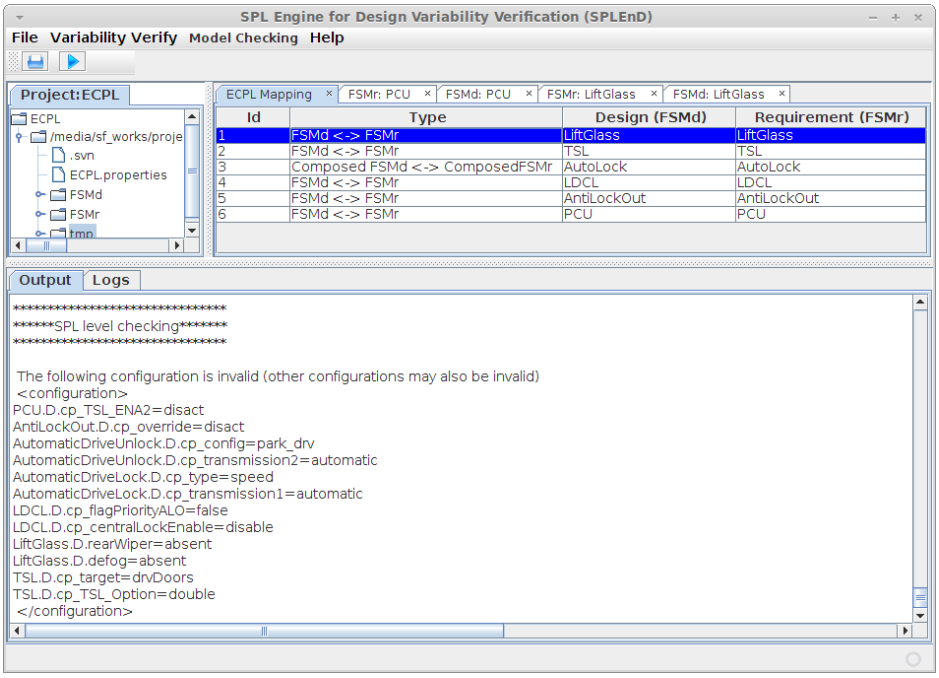

Figure 5.7: Snapshot of SPLEnD: SPL conformance failed: An invalid configuration.

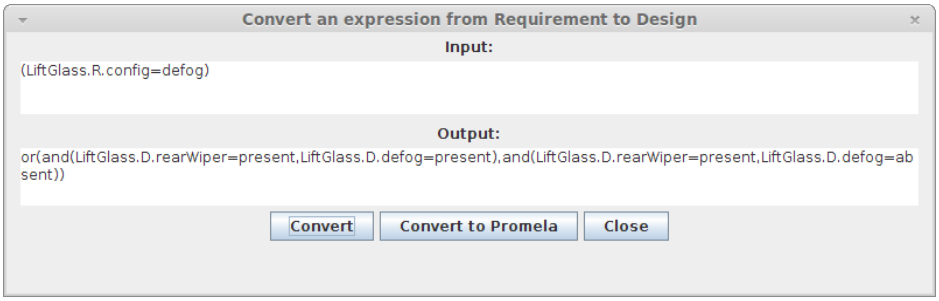

Figure 5.8: Snapshot of SPLEnD: Converting a predicate from Requirement to design.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

In this report, we stress the need to jointly analyze the specification and the implementation of SPLs. Thus, we have started from a formal definition of the notion of traceability and a set theoretical-based framework. We imported existing analyses and propose new analyses, such as superfluousness, explicitness, redundant, union, intersection, valid model, void product model, complete traceability, etc. The analysis problems have been translated into Quantified Boolean Formula and solved efficiently using a QBF solver. The approach is supported by a software tool called SPLAnE and integrated with the existing FaMa framework. We conducted a detailed experimentation with SPLAnE on: (i) large Debian models; (ii) randomly-generated models; and (iii) SPLOT models. We executed all analysis operations with five solvers, i.e., two QSAT solvers (CirQit and RaReQS) and three SAT solvers (Sat4j, PicoSAT and MiniSAT). Further, we experimented SPLAnE for scalability. The experiments are also conducted on the QSAT ap-

proach vs. the SAT approach. For scalability, we took the extended Betty tool and generated a random set of SPL models ranging from five to 50 percent of cross-tree constraints, with 10 different topology and from 10 features to 20,000 features. The scalability result shows that the tool SPLAnE was able to analyze such huge SPL models. The comparison between SAT vs. QSAT results clearly shows that our approach improved the performance by 70% over the SAT-based approach for the analysis operations, like *soundness* and *completeness*.

6.2 Future Work

In future work, we plan to focus on the following extension aspects of this report:

- *More solvers*: Currently, we have implemented SPLANE analysis operations using a reduced number of QSAT and SAT solvers. In the future we plan to add some SMT (Satisfiability Modulo Theories) solvers to this list and proceed with comparative study detecting the pros and cons of each approach.

- *Granularity*: In this report we have considered that the traceability relation exists at the level of features and components. However, A traceability relation can be extended to map a feature with a part of components or a component can be decompose into sub-component to perform a granular mapping or multi-level mapping.

- *Logic paradigms*: We have focused on SAT solving techniques, however, there are some other approaches such as BDD that are appealing for the same usage. In the future, we plan to do a comparison between a QSAT

approach presented in report and quantification over BDD with the implementation across all the analysis operation.

– *Experimentation*: In this report, we have evaluated our approach in a diverse set of scenarios however, we focused in examples containing only 1:m relationships. In the future work we plan to extend the experimentation to n:m relationships to see if this has implications in the scalability of our solution.

This work can be extended in different directions:

For now, the answer of SPLAnE when performing an analysis is close to being binary. Either the analysis succeeds or it fails, and a partial counterexample is given. SPLAnE returns valuations for which the analysis fails, but it is cumbersome for the end user to exploit this information. We would like to derive a diagnosis from this raw information that would orient the user to the source of the failure.

The proposed semantic model of the SPL treats specifications and implementations as sets of features and components, respectively. When richer structures, such as multi-sets are imposed on these elements, it will affect the definitions of *traceability* and *implements*. Then, the underlying logic has to be redesigned to handle the extra expressive power, which will have an implication on the analysis algorithms.

Existing works [98, 97] go beyond Boolean variability and suggest that a feature is not only present or absent, but can be parameterized and that constraints exist between the different configurations of the features. In [97], the authors provide a tool-supported method to relate the variability as it is expressed in the specification and the variability as it is realized in the implementation, for each feature

separately. These (traceability) relations could be integrated into SPLAnE and analyzed.

We also expect that the success of SPLAnE will be established when some requests for new analyses come from end users. The fifteen analyses defined in the article have been taken up from existing works or created by ourselves, but we hope that new analysis ideas will come from usage.

6.3 Future Work: Hybrid State Machine with variability

In this section we introduce an extension of FSMv to allow modeling hybrid systems. It is quite natural to consider situations where the behavior of a feature is governed by ODEs over continuous variables $x_1, \ldots, x_n$. In this case, the continuous variables evolve at different rates, possibly depending on different variation points.

Definition 15 (HSMv)**.** *A hybrid state machine with variability HSMv is a tuple* $\mathcal{A}$=$(L, s_0, \Sigma, \mathcal{V}, E, X, Fl, \rho)$ *where*

- $L, s_0, \Sigma, \mathcal{V}$ *and* ρ *as in the case of TSMv.*
- $E \subseteq L \times \Delta \times \Phi(X) \times R \times \Sigma \times L$ *is the set of transitions such that* $e = (s, g, \varphi_g, R, a, s')$ *represents a transition from location* s *to location* s' *on event* a*; the predicate* $g \in \Delta$ *is called a guard of the transition* e*;* g *is consistent and defines the variability domain of the transition;* $\varphi_g \in \Phi(X)$ *is called the constraint associated with* g*;* $R \in 2^X$ *is a set of variables which are reset on transition* e*;* X *is a finite set of continuous variables, and* $Fl : L \rightarrow Exp^{|X|}$ *is a flow*

function, that associates to each location, a vector of rates. The continuous variables in X grow at the rate specified by the flow function. Exp is an expression on the variables $\mathcal{V}$ as defined in section 4.3.

Note that the choice of values to the variables in $\mathcal{V}$ which decides the variant, also decides the rate of each continuous variable at each location. Every valid configuration gives rise to a variant, a singular hybrid automata [17]. Depending on how the Fl function is defined, we can obtain as variants, linear hybrid automata or rectangular hybrid automata. As seen in section 4.3, we obtain HSMv modeling the requirements as well as the design of a feature separately. The theory presented in the case of TSMv can be extended to the hybrid case, as long as the language inclusion problem for hybrid automata is decidable. This is easily seen, if we assume that the HSMv are all initialized, or have the strong reset condition. In these cases, we can still compute the conformance mappings at the feature level, and use it to check conformance at the feature product line level by synthesizing a QBF formula.

6.3.1 Improving SPLEnD GUI:

We plan to improve SPLEnD by introducing editors for drawing FSMr/FSMd. Also we would like to be able to export the design of the entire SPL including all the variants in the input format of SNIP. The idea is to allow the checking of LTL properties on every variants of the SPL in an efficient way from SPLEnD. The added value of SPLEnD would be to allow the user to express the property using the requirement vocabulary and then automatically translate it to the design vocabulary using the conformance mapping.

Bibliography

[1] Czarnecki K, Wasowski A. Feature Diagrams and Logics: There and Back Again. In: SPLC; 2007. p. 23–34.

[2] Berg K, Bishop J, Muthig D. Tracing software product line variability: from problem to solution space. In: SAICSIT '05. South African Institute for Computer Scientists and Information Technologists; 2005. p. 182–191.

[3] Czarnecki K, Eisenecker UW. Generative Programming: Methods, Tools, and Applications. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.; 2000.

[4] Clements PC, Northrop LM. Software product lines: practices and patterns. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.; 2001.

[5] Benavides D, Segura S, Cortés AR. Automated analysis of feature models 20 years later: A literature review. Inf Syst. 2010;35(6):615–636. Available from: http://dx.doi.org/10.1016/j.is.2010.01.001. doi:10.1016/j.is.2010.01.001.

[6] Metzger A, Heymans P, Pohl K, Schobbens PY, Saval G. Disambiguating the Documentation of Variability in Software Product Lines: A Separation

of Concerns, Formalization and Automated Analysis. In: RE; 2007. p. 243–253.

[7] Czarnecki K, Eisenecker UW. Generative programming - methods, tools and applications. Addison-Wesley; 2000.

[8] Metzger A, Pohl K. Variability Management in Software Product Line Engineering. In: ICSE Companion; 2007. p. 186–187.

[9] Tischer C, Boss B, Müller A, Thums A, Acharya R, Schmid K. Developing long-term stable product line architectures. In: Proceedings of the 16th International Software Product Line Conference - Volume 1. SPLC '12. New York, NY, USA: ACM; 2012. p. 86–95. Available from: http://doi.acm.org/10.1145/2362536.2362551. doi:10.1145/2362536.2362551.

[10] Cordy M, Schobbens PY, Heymans P, Legay A. Beyond boolean product-line model checking: dealing with feature attributes and multi-features. In: Notkin D, Cheng BHC, Pohl K, editors. ICSE. IEEE / ACM; 2013. p. 472–481.

[11] Flores R, Krueger CW, Clements PC. Second-Generation Product Line Engineering: A Case Study at GM.;. Systems and Software Variability Management pages 223–250, Springer, 2013.

[12] Classen A, Heymans P, Schobbens PY, Legay A. Symbolic Model Checking of Software Product Lines. In: Proceedings of ICSE 2011; 2011. p. 321–330.

[13] ter Beek MH, Gnesi S, Mazzanti F. VMC: A Tool for the Analysis of Variability in Software Product Lines. ERCIM News. 2013;2013(93).

[14] Holzmann GJ. The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional; 2003.

[15] Goultiaeva A, Bacchus F. Exploiting QBF Duality on a Circuit Representation. In: AAAI; 2010. .

[16] Alur R, Dill DL. A Theory of Timed Automata. Theor Comput Sci. 1994;126(2):183–235.

[17] Alur R, Courcoubetis C, Henzinger TA, Ho PH. Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems. In: Hybrid Systems I. vol. 736 of LNCS. Springer-Verlag; 1993. p. 209–229.

[18] Pohl K, Böckle G, van der Linden F. Software Product Line Engineering - Foundations, Principles, and Techniques. Springer; 2005.

[19] Pohl K, Metzger A. Variability Management in Software Product Line Engineering. In: Proceedings of the 28th International Conference on Software Engineering. ICSE '06. New York, NY, USA: ACM; 2006. p. 1049–1050. Available from: http://doi.acm.org/10.1145/1134285.1134499. doi:10.1145/1134285.1134499.

[20] Anquetil N, Grammel B, Galvao Lourenco da Silva I, Noppen JAR, Shakil Khan S, Arboleda H, et al. Traceability for Model Driven, Software Product Line Engineering. In: ECMDA Traceability Workshop Proceedings, Berlin, Germany. Norway: SINTEF; 2008. p. 77–86.

[21] Beuche D, Papajewski H, Schröder-Preikschat W. Variability Management with Feature Models. Sci Comput Program. 2004 Dec;53(3):333–352.

Available from: http://dx.doi.org/10.1016/j.scico.2003.04.005. doi:10.1016/j.scico.2003.04.005.

[22] Czarnecki K, Antkiewicz M. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Proceedings of the 4th International Conference on Generative Programming and Component Engineering. GPCE'05. Berlin, Heidelberg: Springer-Verlag; 2005. p. 422–437.

[23] Czarnecki K, Pietroszek K. Verifying Feature-based Model Templates Against Well-formedness OCL Constraints. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering. GPCE '06. New York, NY, USA: ACM; 2006. p. 211–220. Available from: http://doi.acm.org/10.1145/1173706.1173738. doi:10.1145/1173706.1173738.

[24] DeBaud JM, Schmid K. A Systematic Approach to Derive the Scope of Software Product Lines. In: Proceedings of the 21st International Conference on Software Engineering. ICSE '99. New York, NY, USA: ACM; 1999. p. 34–43. Available from: http://doi.acm.org/10.1145/302405.302409. doi:10.1145/302405.302409.

[25] Eisenbarth T, Koschke R, Simon D. A Formal Method for the Analysis of Product Maps. In: Requirements Engineering for Product Lines Workshop, Essen, Germany; 2002. .

[26] Satyananda TK, Lee D, Kang S, Hashmi SI. Identifying Traceability between Feature Model and Software Architecture in Software Product Line using Formal Concept Analysis. In: ICCSA Workshops; 2007. p. 380–388.

[27] Zhu C, Lee Y, Zhao W, Zhang J. A Feature Oriented Approach to Mapping from Domain Requirements to Product Line Architecture. In: Software Engineering Research and Practice; 2006. p. 219–225.

[28] Anquetil N, Kulesza U, Mitschke R, Moreira A, Royer JC, Rummler A, et al. A Model-driven Traceability Framework for Software Product Lines. Softw Syst Model. 2010 Sep;9(4):427–451. Available from: http://dx.doi.org/10.1007/s10270-009-0120-9. doi:10.1007/s10270-009-0120-9.

[29] Cavalcanti YaC, do Carmo Machado I, da Mota PA, Neto S, Lobato LL, de Almeida ES, et al. Towards Metamodel Support for Variability and Traceability in Software Product Lines. In: Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. VaMoS '11. New York, NY, USA: ACM; 2011. p. 49–57. Available from: http://doi.acm.org/10.1145/1944892.1944898. doi:10.1145/1944892.1944898.

[30] Ghanam Y, Maurer F. Extreme Product Line Engineering: Managing Variability and Traceability via Executable Specifications. In: AGILE; 2009. p. 41–48.

[31] Riebisch M, Brcina R. Optimizing Design for Variability Using Traceability Links. In: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems. ECBS '08. Washington, DC, USA: IEEE Computer Society; 2008. p. 235–244. Available from: http://dx.doi.org/10.1109/ECBS.2008.37. doi:10.1109/ECBS.2008.37.

[32] SAT4J-solver; 2010. http://www.sat4j.org/.

[33] MiniSAT-solver,;. http://minisat.se/.

[34] Biere A. PicoSAT;. http://fmv.jku.at/picosat/.

[35] Goultiaeva A, Bacchus F. CirQit; 2013. http://www.cs.utoronto.ca/~alexia/cirqit/.

[36] Janota M, Klieber W. RAReQS-NN;. http://sat.inesc-id.pt/~mikolas/sw/rareqs-nn/.

[37] Benavides D, Segura S, Trinidad P, Cortés AR. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In: VaMoS; 2007. p. 129–134.

[38] SPLAnE-Tool Website;. http://www.cse.iitb.ac.in/~splane/.

[39] Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute of Carnegie Mellon University; 1990. CMU/SEI-90-TR-21.

[40] Schobbens PY, Heymans P, Trigaux JC, Bontemps Y. Generic semantics of feature diagrams. Computer Networks. 2007;51(2):456–479.

[41] Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute of Carnegie Mellon University; 1990. CMU/SEI-90-TR-21.

[42] White Ja, Benavides Dc, Schmidt DCb, Trinidad Pc, Dougherty Bb, Ruiz-Cortes Ac. Automated diagnosis of feature model configurations. Journal of Systems and Software. 2010;83(7):1094–1107. Cited By 24. Available from: http://www.scopus.com/

inward/record.url?eid=2-s2.0-77953132059&partnerID=40&md5=2f87ba959d35ef385ba7e9a2f3c75379. doi:10.1016/j.jss.2010.02.017.

[43] Bagheri E, Di Noia T, Ragone A, Gasevic D. Configuring software product line feature models based on stakeholders' soft and hard requirements. In: SPLC. vol. 6287 LNCS; 2010. p. 16–31. Cited By 5.

[44] Soltani Sa, Asadi Ma, Hatala Ma, Gašević Db, Bagheri Eb. Automated planning for feature model configuration based on stakeholders' business concerns. In: ASE; 2011. p. 536–539. Cited By 4. doi:10.1109/ASE.2011.6100118.

[45] Borba P, Teixeira L, Gheyi R. A theory of software product line refinement. Theor Comput Sci. 2012;455:2–30.

[46] Mendonca M, Wasowski A, Czarnecki K. SAT-based analysis of feature models is easy. In: Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings; 2009. p. 231–240. Available from: http://doi.acm.org/10.1145/1753235.1753267. doi:10.1145/1753235.1753267.

[47] She S, Lotufo R, Berger T, Wasowski A, Czarnecki K. Reverse engineering feature models. In: Taylor RN, Gall H, Medvidovic N, editors. Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. ACM; 2011. p. 461–470. Available from: http://doi.acm.org/10.1145/1985793.1985856. doi:10.1145/1985793.1985856.

[48] Janota M, Kiniry J. Reasoning about Feature Models in Higher-Order Logic. In: Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings; 2007. p. 13–22. Available from: http://doi.ieeecomputersociety.org/10.1109/SPLINE.2007.36. doi:10.1109/SPLINE.2007.36.

[49] Acher M, Collet P, Lahire P, France RB. Separation of concerns in feature modeling: support and applications. In: Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012; 2012. p. 1–12. Available from: http://doi.acm.org/10.1145/2162049.2162051. doi:10.1145/2162049.2162051.

[50] Sven Apel PW Hendrik Speidel, Rhein A, Beyer D. Detection of feature interactions using feature-aware verification. In: ASE; 2011. p. 372–375.

[51] Apel S, Hutchins D. A calculus for uniform feature composition. ACM Trans Program Lang Syst. 2010;32(5).

[52] Harry C Li SK, Fisler K. Verifying cross-cutting features as open systems. In: FSE; 2002. p. 89–98.

[53] Dario Fischbein VB Sebastian Uchitel. A Foundation for Behavioural Conformance in software product line architectures. In: ROSATEA; 2006. p. 39–48.

[54] Cordy M, Classen A, Perrouin G, Schobbens PY, Heymans P, Legay A. Simulation-based abstractions for software product-line model checking. In: ICSE; 2012. p. 672–682.

[55] Patrizia Asirelli SG Maurice H terBeek, Fantechi A. Formal Description of Variability in Product Line Families. In: SPLC; 2011. p. 130–139.

[56] Ina Schaefer DG, Soleimanifard S. Compositional Algorithmic Verification of Software Product Lines. In: FMCO; 2010. p. 184–203.

[57] Ali Gondal MP, Butler M. Composing Event-B specifications - case study experience. In: Software Composition; 2011. p. 100–115.

[58] Mannion M. Using Firts-Order Logic for Product Line Model Validation. In: SPLC; 2002. p. 176–187.

[59] Batory D. Feature Models, Grammars, and Propositional Formulas. In: Proceedings of the 9th International Conference on Software Product Lines. SPLC'05; 2005. p. 7–20.

[60] Larsen KG, Nyman U, Wasowski A. Modal I/O automata for interface and product line theories. In: Proceedings of the 16th European conference on Programming. ESOP'07. Berlin, Heidelberg: Springer-Verlag; 2007. p. 64–79. Available from: http://dl.acm.org/citation.cfm?id=1762174.1762183.

[61] Raclet JB, Badouel E, Benveniste A, Caillaud B, Legay A, Passerone R. Modal interfaces: unifying interface automata and modal specifications. In: EMSOFT; 2009. p. 87–96.

[62] Fantechi A, Gnesi S. Formal Modeling for Product Families Engineering. In: SPLC'08, editor. SPLC. IEEE Computer Society; 2008. p. 193–202.

[63] Gruler A, Leucker M, Scheidemann K. Calculating and Modeling Common Parts of Software Product Lines. In: SPLC; 2008. p. 203–212.

[64] Gomaa H, Olimpiew EM. Managing Variability in Reusable Requirement Models for Software Product Lines. In: ICSR; 2008. p. 182–185.

[65] Jörges S, Lamprecht AL, Margaria T, Schaefer I, Steffen B. A constraint-based variability modeling framework. STTT. 2012;14(5):511–530.

[66] Maurice H terBeek FM, Sulova A. VMC: A Tool for Product Variability Analysis. In: FM; 2012. p. 450–454.

[67] Maurice H terBeek SG, Mazzanti F. Demonstration of a Model Checker for the analysis of product variability. In: SPLC; 2012. p. 242–245.

[68] Krishnamurthi S, Fisler K. Foundations of incremental aspect model-checking. ACM Trans Softw Eng Methodol. 2007;16(2).

[69] Liu J, Basu S, Lutz RR. Compositional model checking of software product lines using variation point obligations. Autom Softw Eng. 2011;18(1):39–76.

[70] Cordy M, Schobbens P, Heymans P, Legay A. Behavioural modelling and verification of real-time software product lines. In: 16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 1; 2012. p. 66–75. Available from: http://doi.acm.org/10.1145/2362536.2362549. doi:10.1145/2362536.2362549.

[71] Lauenroth K, Metzger A, Pohl K. Quality Assurance in the Presence of Variability. SSE, Institut fur Informatik und Wirtschaftsinformatik, univertitat Duisburg Essen; 2011.

[72] Gruler A, Leucker M, Scheidemann K. Modeling and Model Checking Software Product Lines. In: Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems. FMOODS '08. Berlin, Heidelberg: Springer-Verlag; 2008. p. 113–131. Available from: http://dx.doi.org/10.1007/978-3-540-68863-1_8. doi:10.1007/978-3-540-68863-1_8.

[73] Cordy M, Schobbens PY, Heymans P, Legay A. Behavioural modelling and verification of real-time software product lines. In: SPLC (1); 2012. p. 66–75.

[74] YICES; 2010. http://yices.csl.sri.com/.

[75] BDDSolve; 2010. http://www.win.tue.nl/~wieger/bddsolve/.

[76] Roos-Frantz F, Galindo JÁ, Benavides D, Ruiz-Cortés A. FaMa-OVM: A Tool for the Automated Analysis of OVMs. In: Proceedings of the International Software Product Line Conference, SPLC 2012. ACM. ACM; 2012. p. 250–254. doi:10.1145/2364412.2364456.

[77] Galindo JA, Benavides D, Segura S. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In: Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications, Antwerp, Belgium, September 20, 2010; 2010.

p. 29–34. Available from: http://ceur-ws.org/Vol-688/acota2010_paper5_galindo.pdf.

[78] Seidl M. QPRO Format;. http://qbf.satisfiability.org/gallery/qpro.pdf.

[79] Gallery Q. QCIR-G14: A Non-Prenex Non-CNF Format for Quantified Boolean Formulas;. http://qbf.satisfiability.org/gallery/qcir-gallery14.pdf.

[80] Peschiera C, Pulina L, Tacchella A, Bubeck U, Kullmann O, Lynce I. The Seventh QBF Solvers Evaluation (QBFEVAL'10). In: SAT; 2010. p. 237–250.

[81] Mendonca M, Branco M, Cowan D. S.P.L.O.T.: Software Product Lines Online Tools. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications. OOPSLA '09. New York, NY, USA: ACM; 2009. p. 761–762. Available from: http://doi.acm.org/10.1145/1639950.1640002. doi:10.1145/1639950.1640002.

[82] Mohalik S, Ramesh S, Millo JV, Krishna SN, Narwane GK. Tracing SPLs precisely and efficiently. In: SPLC (1); 2012. p. 186–195.

[83] Segura S, Galindo JA, Benavides D, Parejo JA, Cortés AR. BeTTy: benchmarking and testing on the automated analysis of feature models. In: VaMoS; 2012. p. 63–71.

[84] Thüm T, Batory DS, Kästner C. Reasoning about edits to feature models. In: 31st International Conference on Software Engineering, ICSE

2009, May 16-24, 2009, Vancouver, Canada, Proceedings; 2009. p. 254–264. Available from: http://dx.doi.org/10.1109/ICSE.2009.5070526. doi:10.1109/ICSE.2009.5070526.

[85] Milner R. A Calculus of Communicating Systems. Secaucus, NJ, USA: Springer-Verlag New York, Inc.; 1982.

[86] Hoare CAR. Communicating Sequential Processes. Commun ACM. 1978 Aug;21(8):666–677. Available from: http://doi.acm.org/10.1145/359576.359585. doi:10.1145/359576.359585.

[87] Vardi MY, Wolper P. An automata-theoretic approach to automatic program verification. In: Proceedings of LICS 1986; 1986. p. 322–331.

[88] Mathwork/Matlab. Automotive Power Window System;. http://in.mathworks.com/products/simulink/model-examples.html?file=/products/demos/simulink/PowerWindow/html/PowerWindow1.html.

[89] Kastner C, Thum T, Saake G, Feigenspan J, Leich T, Wielgorz F, et al. FeatureIDE: A tool framework for feature-oriented software development. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE; 2009. p. 611–614.

[90] Durán A, Benavides D, Segura S, Trinidad P, Ruiz-Cortés A. FLAME: FAMA Formal Framework (v 1.0). Seville, Spain; 2012. ISA–12–TR–02.

[91] Acher M, Collet P, Lahire P, France R. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. Science of Computer Programming (SCP) Special issue on programming languages. 2013;p. 55. doi:http://dx.doi.org/10.1016/j.scico.2012.12.004.

[92] Batory D, Sarvela JN, Rauschmayer A. Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering. 2004;30(6):355–371. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2004.23.

[93] Galindo JA, Benavides D, Segura S. Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis. In: ACoTA; 2010. p. 29–34.

[94] Trinidad P, Benavides D, Ruiz-Cortés A, Segura S, Jiménez A. FAMA Framework - Poster. In: 12th Software Product Lines Conference (SPLC). IEEE Computer Society Press. Limerick, Ireland: IEEE Computer Society Press; 2008. p. 359. Available from: http://doi.ieeecomputersociety.org/10.1109/SPLC.2008.50. doi:10.1109/SPLC.2008.50.

[95] Team F. http://www.isa.us.es/fama/?FaMa_Current_Projects; 2012.

[96] Vahidi A. JDD: a pure Java BDD and Z-BDD library; 2015. https://bitbucket.org/vahidi/jdd.

[97] Millo JV, Ramesh S, Krishna SN, Narwane GK. Compositional Verification of Software Product Lines. In: IFM; 2013. p. 109–123.

[98] Cordy M, Schobbens PY, Heymans P, Legay A. Beyond Boolean Product-line Model Checking: Dealing with Feature Attributes and Multi-features. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13. Piscataway, NJ, USA: IEEE Press; 2013. p. 472–481. Available from: http://dl.acm.org/citation.cfm?id=2486788.2486851.