lero THE IRISH SOFTWARE ENGINEERING RESEARCH CENTRE

Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines

Mikoláš Janota
Lero – The Irish Software Engineering Research Centre
School of Computer Science and Informatics,
University College Dublin, Dublin, Ireland

Joseph Kiniry
Lero – The Irish Software Engineering Research Centre
School of Computer Science and Informatics,
University College Dublin, Dublin, Ireland

Goetz Botterweck
Lero – The Irish Software Engineering Research Centre
Department of Computer Science and Information Systems,
University of Limerick, Limerick, Ireland

12 March 2008

Lero Technical Report Lero-TR-SPL-2008-02

Contact

Address ..	Lero International Science Centre University of Limerick Ireland
Phone	+353 61 233799
Fax	+353 61 213036
E-Mail	info@lero.ie
Website ..	http://www.lero.ie/

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303-1.

Formal Methods in Software Product Lines: Concepts, Survey, and Guidelines

Mikoláš Janota, Joseph Kiniry
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
{mikolas.janota, joseph.kiniry}@ucd.ie

Goetz Botterweck
Lero, University of Limerick
Limerick, Ireland
goetz.botterweck@lero.ie

Abstract

Based upon our survey of the literature, software product lines is a fertile research field for the application of formal methods. Most computer scientists and software practitioners, including software product lines researchers, are not exploiting the powerful tools and techniques available in modern formal methods. This paper (i) summarizes the core of modern applied formal methods, (ii) discusses software product lines from a formal methods point-of-view, (iii) surveys the application of formal methods to software product lines research, and, most importantly, (iv) highlights key research opportunities and challenges for joint work between formal methods and software product lines researchers.

1 Introduction

Applying mathematics in software development is today known as "Formal Methods." It has a long tradition in Computer Science—especially given that the early decades of Computer Science would today be known as formal methods. A notable example is the early work of Floyd and Hoare in the 1960s that resulted in what is called *Hoare logic* [15] today, which is a means of connecting program code and its logic specifications. Another common example is specification languages, such as Z or VDM [27], that are used to rigorously describe software-intensive systems. This year's Turing Award winners also work in formal methods—they invented model checking, which is used to reason about the correctness of hardware circuits and software systems.

Although, compared to Formal Methods (FM), the research on Software Product Lines (SPL) [9] is a younger and developing field, it has roots in many earlier software engineering areas. For instance, ideas such as mass produced software components [24], software families [28], software reuse [19], domain engineering, domain-specific software architectures [26], and feature modeling [22] have influenced SPL research or provide techniques used in SPL approaches.

So what insights do we gain if we consider these two fields, SPL and FM, *in conjunction*? Many of the challenges brought up by SPL research and practice can be addressed by techniques available in FM—and in part that is already happening. However, we believe that there is still great potential to make even better use of these opportunities. First, there is a whole spectrum of *standard* FM techniques with software tools ready to be used. SPL researchers should be aware of this "toolkit". Second, there are challenges still waiting to be picked and solved. FM researchers should know about these questions waiting to be answered.

Consequently, with this article we focus on the use of of formal methods in the domain of Software Product Lines (SPL). We strive to both raise awareness for available techniques and identify opportunities for further research.

This article brings several contributions: (i) it provides a snapshot of modern formal methods, particularly with respect to the most useful, tool-supported methods, (ii) it summarizes some facets of SPL from an FM point-of-view, (iii) it is a survey of the current use of FM in SPL, and (iv) it highlights opportunities and challenges in SPL research for the application of FM.

Hence, a FM reader will find opportunities for research within the area of SPL, and an SPL reader will find hints and guidelines that will help him to chose the appropriate FM tool or technique for his particular problem.

2 Formal Methods Overview

There are several tomes, hundreds of pages in length, that summarize formal methods [18]. Thus, any summary of modern applied formal methods of a few pages in length does the field injustice. This section focuses instead upon the key facets of *applied formal methods*—those subfields of formal methods that have shown to be, in recent years, the most useful and applicable, have the most supportive and active communities, and have excellent tool support.

The steps of the general process of using FM comprise of the following: 1) create a *formal model* of the studied system, 2) formally express the desired *properties* of that system, 3) attempt prove (show) that the model satisfies these properties, and 4) if the proving process failed, analyze the cause so the model can be amended or the property adjusted.

To create a *formal model*, one needs an underlying *formal system* in which to build the model. Most readers will be familiar with one or more *formal languages* used to define such models, but in the end, each of these languages' semantics is expressed in a logic. Thus, the foundational system used in FM are *logics* of various kinds. Consequently, in these logics, one defines models, expresses properties about models, and then performs reasoning about the model and its properties. In the following sections we discuss some of the most relevant (to the SPL researcher) logics, reasoning techniques and tools, and standardized languages used to express models and properties in these logics for such tools.

2.1 Logics

Logics are the languages of mathematics used to formally capture the concepts about which one wishes to reason. Often, FM practitioners are not writing in a logic directly—rather they are using a language tailored for the particular domain. Nevertheless, it is some form of logic that eventually formalizes the semantics of such a language.

There are two main properties of a logic that determine its suitability for a particular problem. One property is its *expressiveness*, i.e., which concepts are expressible in that logic and how easily such is done. A second property is the difficulty of constructing proofs for statements in that logic.

Propositional logic is a logic with two-values for each variable[1], *true* and *false*, and the formulas are typically expressed using Boolean connectives ($\vee$, $\wedge$, etc.) and negation ($\neg$). Propositional logic is *decidable*, meaning that there exists an algorithm that decides whether a given formula is a tautology (evaluates to *true* under any valuation of variables, also known as being *valid*) or not. Nevertheless, this problem is a co-NP complete problem for CNF (conjunctive normal form) formulas; dually deciding CNF formula's satisfiability is NP-complete.

First-order logics (FOL) are propositional logics embellished with quantifiers used to express things such as "multiplication by 0 yields 0"—$(\forall x)(0 * x = x * 0 = 0)$. FOL are (in the general case) *semi-decidable*, meaning that there exists an algorithm that, for a given formula, terminates if that formula is valid. But, in addition, there is no algorithm that for any formula terminates and determines whether that formula is valid or not (a direct result of Gödel's incompleteness theorems).

Higher-order logics (HOL) [31], of which *second-order logics* are a common variant, enable quantification over functions and predicates, rather than just over variables.

Even though the general FOL is only semi-decidable, there exist *fragments of FOL* that are decidable. An exemplar of such fragment is *Presburger arithmetic*—arithmetic without multiplication. This led to the development of algorithms and tools for deciding the validity of formulas in these fragments, see Sections 2.2, 2.3 for more details.

A variety of logics have been introduced to facilitate writing expressions in different certain domains. For example, the aforementioned temporal logics are logics specialized to express properties of processes evolving over time.

2.2 Reasoning

Logics are used to describe the objects about which one wants to reason. The reasoning itself is an attempt to construct a *proof* of the desired claim. The term proof has a rather wide scope and for the purpose of this text we just assume that a proof is either a set of steps that adhere to the pertaining calculus or it is an exhaustive enumeration of all the possible valuations.

There is a variety of options how a proof is constructed. A traditional *pen and paper* approach offers freedom and flexibility to the mathematician and it is a popular choice for proofs attempted for the first time. The disadvantage is that working by hand does not take advantage of *tool support*.

So how do tools help with proving? Generally speaking, tools provide two functions: *automation* of the proving process, and *validation* of the steps performed by the human. Complementing the palette of logics, FM provide us with a number of tools with the support of different logics, varying in the degree of automation and time requirements, etc.

Table 1 provides an overview of some of the main tools supporting reasoning. The following provides an overview of several kinds of reliable, useful, popular tools that appear in FM. Several concrete examples of these tools are also provided, and nearly all of them are freely available online.

SAT solvers operate on propositional logic and attempt to decide whether the given formula, typically in CNF, is satisfiable or not. Due to the NP-completeness of satisfiability of a CNF formula we are guaranteed termination, but the resource requirements may be unwieldy. The good news is that, in practice, SAT solvers are very capable of solving problems with thousands of free variables. Examples of

[1]Many other useful logics exist that have more than two truth values, including several three-valued logics (with a third truth value of *unknown*), a bevy of many-valued logics (with either a finite, infinite and enumerable, or infinite and non-enumerable number of truth values), temporal logics and logics of causality, fuzzy logics, etc.

SAT solvers include CHAFF, MINISAT, SATZILLA[2].

Finding a satisfying evaluation of a formula is a special case of the *Constraint Satisfaction Problem* (CSP). CSP is given as a set of domains and constraints on these domains and we are interested in the states that satisfy all the constraints. *Constraint Satisfaction Optimization Problem* (CSOP) is an extension of CSP where we are interested in states that apart from satisfying the constraints, maximize (resp. minimize) the so-called *objective function*. *Constraint solvers* are tools that answer queries about the given CSPs or CSOPs. Examples of constraints solvers are JACOP, MINION, VALCSP[3].

Model checking is a fully automated technique targeting properties of finite systems by the exhaustive (explicit or symbolic) search through all possible executions of the system. The modeled system is often described with a state transition system, and the desired properties in a temporal logic (LTL, CTL, etc.). The domain of search is finite but very large. Examples include Bogor and SPIN.

Tools that decide the validity of formulas of decidable fragments of FOL are typically referred to as SMT solvers (SMT stands for *satisfiability modulo theories*). Popular SMT solvers include Yices, CVC3, MathSAT[4].

Fully automated tools that handle first-order formulas are called *automated theorem provers*. As FOL is undecidable, an automated theorem prover is not guaranteed to terminate. Popular automated provers include CVC, Simplify, Vampire, Yices, Z3.

Proof assistants (also known as *proof checkers* or sometimes *logical frameworks*) are tools that let the user determine the steps of the proof, while checking that these adhere to the pertaining calculus. Moreover, they provide automation for certain types of steps. Proof assistants typically support higher-order languages and thus are very expressive. Examples include Coq, HOL, Isabelle, PVS.

Finally, some tools are somewhere halfway between automated provers and interactive proof assistants. Many of them support specialized logics, such as rewriting logic in the case of Maude, or reasoning techniques, such as induction in FOL, in the case of ACL2.

Apart from reasoning tools, some programming languages are close to logic. *Logic programming* (e.g., Prolog) enables expressing logic formulas as programs in a straightforward fashion.

2.3 Standardized Languages

Many of the input languages of the tools just discussed have been de facto standardized. It is a very important fact that a language has been standardized; such avoids tool "lock-in" and enables the use of multiple tools to solve (various different parts of) a problem. Here we provide several important examples of such standardized languages.

Tool category	**Logic**
SAT solvers	propositional logic
constraint solvers	constraints over finite domains
model checkers	temporal logic and state transition systems
SMT solvers	decidable fragments of FOL
automated theorem provers	FOL
proof assistants	HOL

Table 1. Tools for Mechanized Reasoning.

Specification languages are languages used to describe systems (hardware, software, business, etc.). They are used to abstract away from the implementation details of the system and reason about its overall properties. The languages B, JML, VDM, Z are examples of popular specification languages with excellent tool support.

For automated reasoning tools, several regular competitions expedited a widespread of standard problem formats. In SAT solving, the most common format is the DIMACS format, designed for the DIMACS Implementation Challenges and used for the aforementioned SAT competition.

Thousand Problems for Theorem Provers (TPTP) is a competition and a set of benchmarks for first-order theorem provers; this format is now widely supported.

The Satisfiability Modulo Theories (SMT) initiative focuses on automated reasoning for decidable fragments of FOL. SMT provers are very popular and powerful, as the logics supported by such provers are primarily useful in the fields of hardware and program verification.

3 Software Product Lines

Some core concepts of software product lines are now introduced. This overview is based on an ongoing survey of SPL literature, however, here SPL concepts are described from a FM point-of-view. By taking this fresh perspective, specific SPL approaches are generalized using FM concepts (e.g., as sets or functions). The foundation provided here facilitates a review of some approaches that apply FM in SPL (Section 4) and the identification of several open research challenges (Section 5).

3.1 Motivation: Economic production

The major motivation behind SPL is that, under certain conditions, it is more efficient to treat a group of similar

[2] See the results of the SAT competition for more information.
[3] See the results of the CSP Solver Competition for more information.
[4] See the results of the SMT Competition for more information.

software systems as a whole, rather than handling each individual system on its own. In this sense Parnas defines *program families* as

> ... sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. [28]

To avail of these commonalities, SPL engineering applies techniques that were known before, e.g., *software reuse* and *software components*. In contrast to other approaches, SPL engineering strives to achieve reuse in a strategic, prescribed way and to use a managed set of features satisfying the needs of a particular market segment [9].

Consequently, in comparison to Single System Engineering, SPL Engineering requires additional upfront investment to establish the product line (see ❶ in Figure 1). Examples of such investment include the definition of a product line's scope, the development of reusable assets, and the creation of a production plan that describes how products are derived.

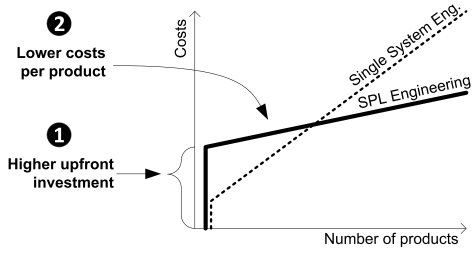


Figure 1. Costs of SPL Engineering.

If this investment is to pay off, a sufficient number of products and a lower costs per product is necessary to leverage the economy of scale (see ❷ in Figure 1).

Such an approach is applied to FM in SPL as well. For instance, if a model analysis technique that optimizes product derivation is created, the technique and its related tools must be introduced to an organization ❶, but the marginal costs of such an introduction is reduced by performing product-related processes more efficiently ❷.

3.2 SPL Artifacts

To support the efficient execution of SPL processes with FM, it helps if we first understand the types of artifacts used in SPL and the relationships between these artifacts. Later on, we will see how we can exploit these relationships with FM, for instance by checking the conformance of one model against constraints given in another model.

SPL engineering employs artifacts very similar to those used in general software engineering like specifications, models, and source code. However, SPL artifacts are different and distinct in some ways. Some of these differentiating aspects are summarized in Figure 2.

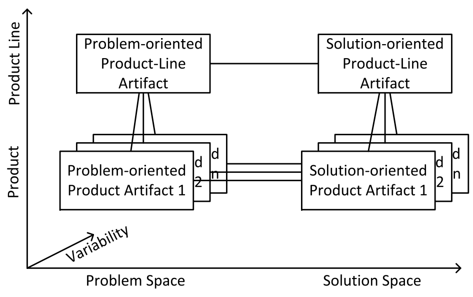

Product Line
Product
Problem-oriented Product-Line Artifact
Solution-oriented Product-Line Artifact
Problem-oriented Product Artifact 1
d 2
d n
Solution-oriented Product Artifact 1
d 2
d n
Variability
Problem Space
Solution Space

Figure 2. SPL Artifacts.

3.2.1 Product line vs. Product

SPL approaches distinguish between two levels of system development: the *product line* and its *products* (see vertical dimension in Figure 2). Often, there is *one* product line artifact describing an aspect for the whole product line, and *many* product-specific artifacts describing that aspect of the individual products. For example, consider a feature model with many product-specific feature configurations, or a product line architecture (PLA) with many product-specific architectures, based upon this PLA.

We see this introduction of an artifact that *explicitly* describes an aspect (e.g., features or architecture) for the whole software family as a major contribution of SPL Engineering. Such a product line artifact serves multiple purposes: (i) it describes the products by modeling overall structures and constraints, (ii) it provides guidance when creating a product-specific instance, and (iii) it enables conformance checking of instances with some given constraints.

3.2.2 Variability

As a direct consequence of the structure *"one product line, many products,"* SPL engineering has to provide a means to describe the commonalities and variability between different products (see the diagonal dimension in Figure 2). This is, for instance, implemented by extending existing modeling languages, or by introducing a separate variability model that contains references to elements in other artifacts.

3.2.3 Mapping from Problem to Solution

Another dimension through which one categorizes and relates SPL artifacts is seen in the distinction between *problem* and solution (see horizontal dimension in Figure 2).

An example of a problem-oriented SPL artifact is *feature models*, which describe each individual problem as a combination of *features*; where a feature is ...

> ... a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system. [22]

On the other hand, an example of a solution-oriented artifact is architecture description languages (ADLs). Traditional ADLs now support variability, for example, by supporting the description of multiple architectures [12].

Consequently, an interesting research area is product line models that integrate both problem- and solution-oriented aspects—fertile ground for further research. Section 4 provides several references to such models.

3.3 Interpretation from a FM Perspective

In the preceding section, relationships between SPL artifacts was discussed. Here, these relationships are examined from a FM perspective (see Figure 3).

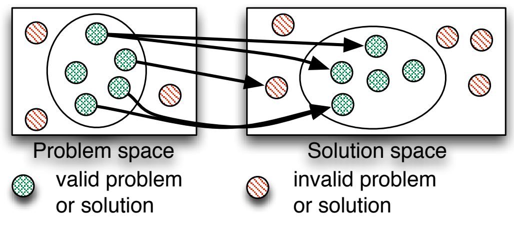


Figure 3. Product line concepts.

A *problem space* is the domain of problems under consideration, and a *solution space* is the range of potential solutions. These spaces determine the scopes of the problems and solutions within a given domain, respectively.

For a particular product line, one needs to determine the set of problems that the product line covers. These problems are referred to as the *valid problems*. In the context of the preceding section, this determination corresponds to a product-line artifact that defines a set of instances at the product level.

Analogously, *valid solutions* that are realizable by the product line must be identified. An example of a *solution space* is a set of programs written in the `C` programming language, and *valid solutions* in this context are exactly those programs which compile.

Furthermore, to link the two domains, a *mapping from problems to solutions* is necessary. This mapping, depicted as arrows in Figure 3, determines which elements of the solution space are solutions for a particular given problem. Such a mapping is, for example, realized by a program generator that takes a feature configuration as input and produces a valid program as output.

3.4 Processes

Product development with a SPL encompasses numerous processes and roles. Figure 4 offers a somewhat simplified version thereof.

When establishing the product line, an expert *develops* a problem-space model and the reusable assets.

The customer is then presented only with the view of the problem space model, representing the set of available products from which he or she may choose. By *configuration*, the customer chooses a member of the problem space, which in turn defines the desired properties of the product. If the set of valid problems is large and has complex dependencies between the configurable parts, this configuration process may require interaction with the application expert.

Finally, the product itself must be *assembled* according to the product description. As discussed earlier in Section 3.1, the success of the product line hinges on this particular point since, if product assembling is not efficient, one might just as well develop product by product. Hence, one might say that the 'holy grail' for a product line is a fully automated assembly.

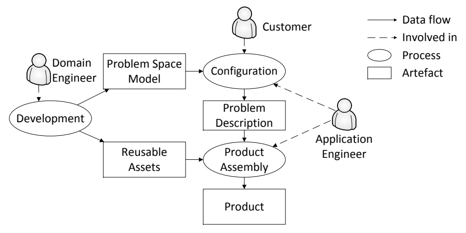


Figure 4. SPL Processes.

3.5 Properties of a Product Line

The mathematical point of view on a product line outlined in Section 3.3 enables us to explicitly point out the desirable, resp. undesirable, properties of the relevant concepts. This section discusses such properties and later Section 4 demonstrates how various researchers have targeted these properties with formal analyses.

First, we ask what are the desired, resp. undesired, properties of the mapping? On reflection, we have identified the following properties of such mappings:

1. If a solution is classified as invalid, it must never be derived. Hence, a valid problem must not be mapped to an invalid solution. *(Validity is preserved across the mapping.)*

2. We must have a solution for all problems under consideration. Hence, each valid problem must have a solution, and furthermore that solution must be mapped

to from that particular valid problem.
(The restriction of the mapping to the valid problems must be total.)

3. If two problems are distinct, the solutions must be distinct as well. Hence, no two distinct problems may map to the same solution. The violation of this property is not as alarming as for the previous two properties, but still, the violation is worth exploring.
(The restriction of the mapping to the valid problems must be injective.)

We do not prohibit mapping a problem to multiple solutions *(the mapping need not be a function)*, as leaving a non-determinacy in a system's description is a common and useful practice in program/system specification.

Another important property of the (problem and solution) spaces is that none of them should be empty. Hence, the product-line models that describe spaces must be *consistent*. (Were they to be inconsistent, then any solution would satisfy any problem.)

Finally, the *absence of redundancies* is a desirable property of the individual models and the mappings between them. For instance, for a given feature model, a feature that cannot be selected into any product—a so-called *dead feature*—is redundant. Dead features must either be removed, or the underlying constraints must be reconsidered.

4 Survey

This section summarizes some recent research that lies within the intersection between FM and SPL. It is by no means complete. Specific papers were chosen because they highlight the clear and straightforward application of FM to a variety of SPL problems.

4.1 Feature Models

To express the semantics of feature models in propositional logic is rather straightforward. Features are represented as propositional variables, and the semantics of a feature model is a propositional formula, e.g., $f_1 \Rightarrow f_2$ for feature f_1 *requires* f_2. Thus, a feature configuration, expressed as a variable valuation, conforms to the feature model if and only if it satisfies the formula defined by the feature model.

This approach is extensively studied by Schobbens et al. [30]. In this work, they provide a survey of the variants of FODA-like feature models, their semantics, investigates their expressiveness, and studies the complexity of several feature model analyses.

Jonge and Visser [13] propose the use of context-free grammars for capturing the semantics of feature models. Feature configurations conforming to a feature model are defined as the sentences generated by the pertaining grammar. This idea is extended by Batory [2] by combining grammars and propositional logic.

Czarnecki et al. [10] also use context-free grammars to capture the semantics of feature models. Their meta model is more expressive than the feature models covered by the aforementioned articles, as their meta model enables *cardinalities*, similar to ER-multiplicities, and *feature cloning*.

Höfner et al. [16] take an algebraic approach to feature modeling. In this approach, a product family is a set of sets of features, and the language of algebra is used to abstract from the sets. Dependencies, such as exclusion, are expressed as additional equalities. What makes this approach interesting is that it enables to use multisets (sets with repetitions of an element, aka *bags*) instead of sets. This extension permits the modeling of feature models with cloning. Even though this approach seems interesting, no other publications along this line of research are known.

In our own work [21] we use the proof assistant PVS to reason in and about feature models. Thanks to the expressive power of the PVS language and HOL, this approach enables reasoning about the full bevy of feature models—with cloning or attributes. Furthermore, it supports reasoning about feature modeling mechanisms, such as: "is this operation a specialization?".

4.2 Analyses

Section 3.5 introduced several desirable properties of a product line (and the related models). This section discusses the research focused on identifying the violation of some of these properties.

Let us recall some of the recurring terminology. A feature model is *consistent* if and only if there is at least one configuration permitted by the model. A *dead feature* is a feature that cannot be selected into any configuration.

Probably the first documented analysis of a feature model was done by Mannion who uses Prolog to check for the consistency and the absence of dead features [23].

This line of research was followed by Batory [2] who not only applies a SAT solver (see Section 2.2) to detect inconsistencies of a feature model, but also to *debug* a feature model, i.e., to test whether certain feature combinations that are expected to be allowed, resp. disallowed, really are.

Along this line, Benavides et al. [7] apply constraint programming to analyze feature models. Constraint programming enables features to have attributes of finite domains. Furthermore, one can ask a variety of queries, such as model consistency, number of models, or to find a product with certain properties (filtering).

For further reference, see a survey on the automated analyses of feature models by Benavides et al. [6] or a set of challenges for feature model analyses identified by Batory et al. [3].

The work of Sinz et al. [32] focuses on general product configuration in the automotive industry, a domain that, until recently, did not necessarily deal with software. However, this domain bares a strong resemblance to SPL, and software-intensive subsystems are gaining importance.

The authors operate in a two tiered model: *configuration at the engineering stage* and *configuration at the manufacturing stage*. The engineering stage is concerned with the capabilities of the intended products. The manufacturing stage needs to deal with things like availability of the individual parts. The scope of intended products is expressed as a set of rules, and several consistency criteria for these rules are described. A SAT solver of their own making is used which supports some specialized constructs useful for their particular problem, e.g., $choose_l^k$ operator[5]. This formalism enables the assignment of *time frames* to the rules in the manufacturing stage, and these time frames determine when the pertaining rule is in effect.

An important part of consistency analysis is to provide the *source* of an inconsistency, if one is found. Research on this topic is still somewhat scarce, but the work of Trinidad et al. [34] shows some promising results.

4.3 Aiding the Configuration Process

Once the feature model is expressed in a formal (logical) representation, tool support assists the user during the configuration process, inferring the consequences of the user-made selection thus far (*decision propagation*).

In the case of feature selection, the consequences of the user's selection are other features that must be selected or deselected. Batory applied a *Logic Truth Maintenance System* to implement such support [2]; the implementation is part of the AHEAD tool suite [5] as the tool `guidsl`.

For further discussion on this topic, see Section 5.2.

4.4 Product Line Modeling

This section focus on approaches that model and analyze the product line as a whole (see Figure 3).

Feature oriented programming (FOP) represents a line of research where programs are composed on demand from some specified features. Hence, in FOP, features are mapped, in a 1-to-1 fashion, to pieces of program code [5]. In the FOP context, Batory and Geraci [4] specify rules for the validity of component composition and attribute grammars are used to implement the checks of these rules.

In the context of the FOP framework AHEAD [5], Thaker et al. [33] use a SAT solver to validate that a feature model does not allow feature combinations which result in illegal code.

Similarly, Van der Storm proposes a model where features are mapped to components with dependencies. Subsequently, one validates whether any feature configurations violate these dependencies [35].

Another example along this same line is work by Czarnecki and Pietroszek [11] who use UML to describe both the problem and the solution space. Additionally, they use the Object Constraint Language (OCL) to specify properties of the valid solution space in order to show that a valid problem will not be mapped to an invalid solution.

All of this summarized research are different variants for detecting the aforementioned property: that a valid problem must not be mapped to a invalid solution (Section 3.5).

Kumbang [1] is a framework, implemented as an Eclipse plugin, that provides support for modeling in two-tiers—*features* and *components*. A reasoning engine is used to facilitate the configuration process.

In our own work [20], a generic formalism for reasoning about product line models is provided. This formalism mathematically captures the idea illustrated by Figure 3. Furthermore, it enables the formalization and study of similar approaches that also deal with models of features and their implementation.

We conclude this section with a reference to the domain of program generation. Effectively, a program generator realizes the mapping between the problem and solution space. Huang et al. [17] use an automated theorem prover to verify that a given template will only generate valid Java code (i.e., code that will compile, in their case). In our terminology, this generation mechanism ensures that no valid problem is mapped to an invalid solution.

4.5 Component Composition

The work of Blundell et al. [8] utilizes model-checking in the SPL context. Components, modeled as state machines, correspond to individual features. Constraints on the model are expressed using temporal logic (CTL). Feature composition corresponds to adding transitions between the relevant state machines.

The work of Scheidemann [29] is somewhat more general. The author presents algorithms for choosing a (minimal) subset of all possible configurations to verify a whole family. These algorithms exploit the fact that certain requirements apply only to a subset of components.

5 Recommendations and Challenges

This section provides several guidelines for those who wish to employ FM in their research or engineering pro-

[5] $choose_l^k(t_1, \ldots, t_n)$ evaluates to true if and only if at least l and at most k out of $t_1, \ldots, t_n$ evaluate to true.

cesses. We use the marker 🅲 to denote open research challenges in SPL via FM.

5.1 Model Analyses

As we have mentioned before, models are analyzed to uncover undesired properties. One popular form of analysis is what is known as *sanity checking*.

Figure 5 illustrates a potential procedure for such analyses. We first have to interpret the initial model ❶, for instance by translating f_1 *requires* f_2 as $f_1 \Rightarrow f_2$. This yields a formal model in the chosen logic ❷, that is represented in a concrete syntax in a standardized format ❸. Using this data as input to an automated tool, the desired analysis results are obtained ❹.

We will now provide guidelines for choosing the appropriate means (to perform such analyses) according to the form of the model. A bulk of the SPL research focuses on the feature models and so does this section. Nevertheless, these guidelines apply to other types of models so long they are expressible in the corresponding logic.

For the sake of brevity, we introduce the following notation: Let $\mathcal{M}$ be a feature model then $[\![\mathcal{M}]\!]$ is a formula corresponding to $\mathcal{M}$'s semantics. Additionally, we say that a model is consistent if and only if the formula expressing its semantics, i.e., $[\![\mathcal{M}]\!]$, is satisfiable.

By *model debugging* we mean that we expect the model to enable or disable certain specific configurations. For instance, we expect that a feature model does not allow one to select both a manual and an automatic gearshift in an automobile.

The procedure to verify such a requirement is to first translate the requirement to its logical representations, say a formula ϕ, and then check for the satisfiability resp. unsatisfiability of $[\![\mathcal{M}]\!] \wedge \phi$. E.g., for the given example we require that the following formula is unsatisfiable:

$$[\![\mathcal{M}]\!] \wedge (\mathit{manual} \wedge \mathit{automatic}) \tag{1}$$

Recall that dead features are features that do not appear in any product. Mathematically speaking, a feature represented by the variable f is dead in the model $\mathcal{M}$ if and only if the following formula is unsatisfiable.

$$[\![\mathcal{M}]\!] \wedge f \tag{2}$$

For models expressible in propositional logic, as most variants of FODA are, SAT solvers are a natural choice for *consistency checking* and *model debugging*. To use a SAT solver, one either chooses to export the problem into the DIMACS format, or directly uses the solver's program interface. For instance, MiniSat provides such an interface [14].

To determine whether a feature is dead, a SAT solver is called repeatedly on formula 2 for each feature. We are not aware of a documented, more efficient technique and we believe that minimizing the number of calls to the SAT solver to find dead features is a research challenge. 🅲

For models whose semantics is not propositional, but which still have finite domains of values, constraint solvers are a good choice. Using SMT solvers is also a possibility and using such increases the expressiveness of constraints on features and attributes. Nevertheless, we are unaware of any research using SMT solvers in the feature modeling domain. 🅲

Even thought there has been research in the semantics of feature models that enable cloning of features [10, 16, 21], we are not aware of a semantics that is readily used with automated tools. We believe that constraint solvers are useful when there is no support for cloning arbitrary many times, as the domain is still finite. For feature models where a feature can be cloned arbitrary many times, more powerful techniques are required. 🅲

As we can see, to solve problems that lead to satisfiability is rather straightforward and off-the-shelf tools are useful. However, there are problems that take into account *all* the configurations of the model, such as counting the number of all configurations. Finding the number of valid configurations does not necessarily uncover a problem (unless the number is 0). However, this number is an important sanity check: if the number is too large, the constraints might be too loose; if the number is too small, the constraints might be too restrictive or too specific for the products developed so far. One can imagine other kinds of similar analyses, such as an average price of all valid products. 🅲

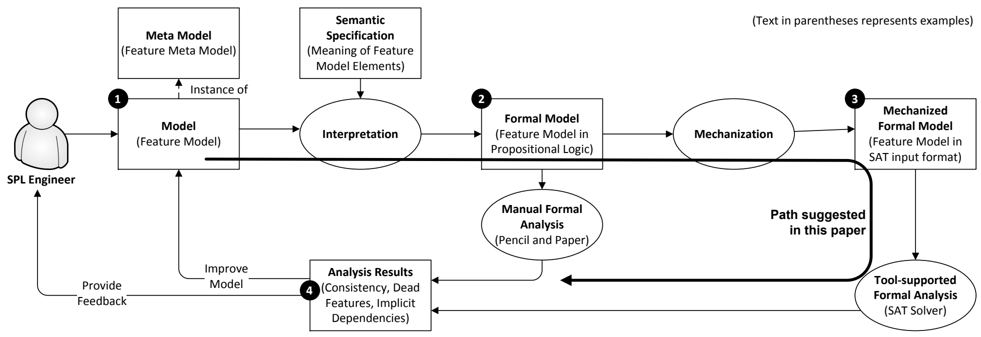
Constraint solvers typically have the support for returning all possible solutions. Once solutions are obtained, further analysis can be performed. Naturally, this technique is problematic once the set of solutions becomes unwieldy.

One way to store and build the whole set of configurations for the propositional logic formulas are Binary Decision Diagrams (BDDs) [25]. BDDs are essentially efficient representations of sets of satisfying valuation of a given formula. Libraries with BDD support are easily accessible, e.g., the JavaBDD library. BDDs, however, suffer from the space-explosion problem. Moreover, BDDs are not readily applicable to models whose semantics is not propositional.

Hence, we believe that there is utility in looking for dedicated algorithms for such analyses, like counting numbers of configurations. We should note that recent advances in the SAT community make counting the number of satisfying valuations for a propositional formula possible, including estimates, see e.g., [36]. 🅲

5.2 Configuration Process

During the configuration process firstly we want to make sure that the user does not make decisions that violate the



(Text in parentheses represents examples)
Meta Model (Feature Meta Model)
Semantic Specification (Meaning of Feature Model Elements)
Instance of
1
Model (Feature Model)
Interpretation
2
Formal Model (Feature Model in Propositional Logic)
Mechanization
3
Mechanized Formal Model (Feature Model in SAT input format)
SPL Engineer
Manual Formal Analysis (Pencil and Paper)
Path suggested in this paper
Improve Model
Provide Feedback
4
Analysis Results (Consistency, Dead Features, Implicit Dependencies)
Tool-supported Formal Analysis (SAT Solver)

Figure 5. Data flow for model analysis.

feature model's restrictions. Secondly, we would like to help the user with making the decisions.

5.2.1 Completing a Configuration

Given a model $\mathcal{M}$ and a set of decisions expressed by the formula ϕ, the first question to ask is whether it is possible to complete the configuration process while keeping these decisions. That is a satisfiability problem of the formula $[\![\mathcal{M}]\!] \wedge \phi$ and hence, the recommendations for satisfiability given above apply here as well.

A step further is to look for an actual configuration that respects the decisions already made. In other words, we wish to automatically complete a partial configuration. This is desirable if, e.g., the customer expressed some parameters for the configuration and is indifferent to the setting of the rest of the features.

If we are looking for *any* completion of the given configuration and we are using a constraint or SAT solver, the query for satisfaction of $[\![\mathcal{M}]\!] \wedge \phi$ is sufficient, as these types of solvers also provide the actual solution.

The problem gets trickier if we are looking for a completion that is optimal in some sense, e.g., for the cheapest solution. Here, SAT solvers typically would not help, even though MINISAT+ (a modification of MINISAT) does allow for minimizing an expression. Nevertheless, most constraint solvers enable one to specify the so-called objective function, a function that the solution should minimize/maximize (i.e., CSOP specification, recall Section 2.2).

5.2.2 Decision Automation

The previous subsection discussed how to complete a partial configuration by a single call to a tool that provides all the missing decisions. However, tools can also help the person throughout an interactive configuration process by automatically making necessitated decisions. Primarily, this means 1) preventing the user from making the decisions that are no longer possible, as they would violate some constraint 2) automatically making the decisions that must be made if the configuration process is to succeed.

In the propositional case, the points 1) and 2) collapse into one as they both mean enforcing a particular value (*true* or *false*) on a certain variable. For instance, once a is selected (set to *true*) in the presence of the constraint $\neg(a \wedge b)$, b must be deselected (set to *false*).

In the case of variables with larger domains, 1) corresponds to removing elements from a particular domain that violates the model's constraints and 2) corresponds to automatically selecting a value that is the only one left in the particular domain.

As we have mentioned before (Section 4.3), a Logic Truth Maintenance System is a natural choice for the propositional case, see e.g., Batory [2].

It is easy to see how 1) and 2) are implemented for attributes with finite domains. In fact, constraint solvers contain such decision propagation. Nevertheless, we are not aware of a documented and available application of decision automation for non-propositional constraints. 🅲

6 Conclusions

By providing a comprehensive survey of the application of FM in the software product line (SPL) literature, we have shown that many formal methods approaches have been adopted into the state-of-the-art techniques. Besides this overview, our article identifies what these approaches have in common and identifies numerous important concepts.

Furthermore, we have provided guidelines for the application of formal methods for particular tasks in the SPL domain, coupled with pointers to relevant tools and literature.

Last but not least, the article identifies open challenges for research of formal methods in SPL.

7 Acknowledgements

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1.

References

[1] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21, 2007.

[2] D. Batory. Feature models, grammars, and propositional formulas. In H. Obbink and K. Pohl, editors, *Proceedings of the 9th International Software Product Line Conference (SPLC)*, 2005.

[3] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006.

[4] D. Batory and B. J. Geraci. Validating component compositions in software system generators. In *Proceedings of the 4th International Conference on Software Reuse (ICSR '96)*. IEEE Computer Society, 1996.

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[6] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, 2006.

[7] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *CAiSE'05*, 2005.

[8] C. Blundell, K. Fisler, S. Krishnamurthi, and P. Van Hentenryck. Parameterized interfaces for open system verification of product lines. In *IEEE International Symposium on Automated Software Engineering*, 2004.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison–Wesley Publishing Company, 2002.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In R. L. Nord, editor, *Proceedings of Third International Conference on Software Product Lines (SPLC)*, 2004.

[11] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proceedings of the 5th International Conference on Generative Programming And Component Engineering*, 2006.

[12] E. M. Dashofy and A. van der Hoek. Representing product family architectures in an extensible architecture description language. In *Proceedings of PFE '01*, 2001.

[13] M. de Jonge and J. Visser. Grammars as feature diagrams. Presented at the Generative Programming Workshop 2002, Austin, Texas, Apr. 2002.

[14] N. Eén and N. Sörensen. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT '03)*. Springer-Verlag, 2003.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[16] P. Höfner, R. Khedri, and B. Möller. Feature algebra. In *Proceedings of the 14th International Symposium on Formal Methods (FM)*, LNCS. Springer-Verlag, 2006.

[17] S. S. Huang, D. Zook, and Y. Saragdakis. Statically safe program generation with SafeGen. In R. Glück and M. Lowry, editors, *GPCE '05*, LNCS. Springer-Verlag, 2005.

[18] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, second edition, 2004.

[19] I. Jacobson, M. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[20] M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. To appear in FASE 08, 2008.

[21] M. Janota and J. Kiniry. Reasoning about feature models in higher-order logic. In P. Kellenberger, editor, *Proceedings of SPLC*, 2007.

[22] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA), feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie Mellon University, Nov. 1990.

[23] M. Mannion. Using first-order logic for product line model validation. In G. J. Chastek, editor, *Proceedings of SPLC '02*, 2002.

[24] D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, 1968.

[25] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD-foundations and applications*. Springer-Verlag, 1998.

[26] E. Mettala and M. Graham. The domain-specific software architecture program. Technical Report TRCMU/SEI-92SR-9, Carnegie Mellon SEI, June 1992.

[27] N. Nissanke. *Formal Specification: Techniques and Applications*. Springer-Verlag, 1999.

[28] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 1976.

[29] K. D. Scheidemann. Optimizing the selection of representative configurations in verification of evolving product lines of distributed embedded systems. In L. O'Brien, editor, *Proceedings of SPLC '06*, 2006.

[30] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proceeding of 14th IEEE International Requirements Engineering Conference (RE)*, 2006.

[31] S. Shapiro. *Foundations Without Foundationalism: A Case for Second-Order Logic*. Oxford University Press, 1991.

[32] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17:75–97, 2003.

[33] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE '07*, 2007.

[34] P. Trinidad, D. Benavides, A. Durán, A. Ruiz-Cortés, and M. Toro. Automated error analysis for the agilization of feature modeling. *Journal of Systems and Software*, 2007.

[35] T. van der Storm. Generic feature-based software composition. In *Proceedings of 6th International Symposium on Software Composition*, 2007.

[36] W. Wei and B. Selman. A new approach to model counting. In *Theory and Applications of Satisfiability Testing (SAT '05)*. Springer-Verlag, 2005.