Tracing SPLs Precisely and Efficiently

Ramesh S,
Swarup Mohalik*,
Jean-Vivien Millo†
India Science Lab
General Motors
TCI, Bangalore, India
ramesh.s@gm.com

Shankara Narayanan Krishna
Department of Computer Science and Engineering
IIT Bombay, Powai
Mumbai, India
krishnas@cse.iitb.ac.in

Ganesh Khandu Narwane
Homi Bhabha National Institute
Anushakti Nagar
Mumbai, India
ganeshk@cse.iitb.ac.in

ABSTRACT

In a Software Product Line (SPL) comprising specifications (feature sets), implementations (component sets) and traceability between them, the definition of product is quite subtle. Intuitively, a strong relation of implementability should be established between implementations and specifications due to traceability. Various notions of traceability has been proposed in the literature : [13], [17], [8], [9]; but we found in our experience that they do not capture all situations that arise in practice. One example is the case where, an implementation, due to packaging reasons, contains additional components not required for a particular product specification. We have defined a general notion of traceability in order to cover such situations. Moreover, state-of-the-art satisfiability based notions lead to products where the implementability relation does not exist. Therefore, in this paper, we propose a simple, set-theoretic formalism to express the notions of traceability and implementability in a formal manner. The subsequent definition of SPL products is used to introduce a set of analysis problems that are either refinements of known problems, or are completely novel. Last but not the least, we propose encoding the analysis problems as Quantified Boolean Formula (QBF) constraints and use Quantified SAT (QSAT) solvers to solve these problems efficiently. To the best of our knowledge, the QBF encoding is novel; we prove the correctness of our encoding and demonstrate its practical feasibility through our prototype implementation *Software Product Line Engine (SPLE)*.

1. INTRODUCTION

Software Product Line (SPL) is a development framework to jointly design a family of closely related software *products* in an efficient and cost-effective manner. The basis of an SPL is a collection of features, called the *scope* and a collection of reusable components called the *core assets*, which are developed once for the entire family [18]. The possible products in the family are *specified* through subsets of features from the scope, and are *implemented* by subsets of components selected from the *core assets*.

In the literature, the structure and analysis of SPL specifications (feature sets) have been studied in great depth. Since the products of an SPL are closely related, it is natural to present their specifications as variations of each other. The variations include choices (termed *variation points*) and dependency constraints among the variation points, such as exclusivity (if f is present then g should not be present), mandatory (if f is present, then g must also be present) and more complex cardinal constraints. Managing variability in large industrial SPLs is quite complex and has given rise to a number of analysis problems, such as detecting the common and dead elements, counting the number of variants etc. Therefore, this has been the focus of SPL research in the recent years. A comprehensive survey of these analysis problems and their solutions can be found in Benavides et al.[5].

On the other hand, since the core assets include reusable components and the products are closely related, it is expected that the implementations (subsets of components) for the products are also closely related and hence can be specified through the variability constraints over the core assets. Therefore, notations such as FODA (Feature-Oriented Domain Analysis) diagrams, used to represent feature variability, can also be used to select variant implementations. Such an approach has been taken in [17].

Recently, the analysis problems around variability of specifications have been cast in propositional logic constraints and SAT-solvers have been used to solve the problems [3]. This is due to the fact that propositional logic provides a uniform and expressive framework to formulate the analysis problems and also because of the tremendous progress in SAT-solving in recent years which makes it possible to address industrial scale problems. The same techniques can be applied to the variability of implementation as well, for similar kinds of analysis.

However, due to the fact that the traceability relation connects two different sets of artifacts, namely, specifications and implementations, certain issues arise naturally in the

*swarup.mohalik@gm.com
†jean-vivien.millo@gm.com

full SPL consisting of specifications, implementations and traceability between the two. These issues include the very definition of traceability, the induced *implements* relation from implementations to specifications, the new SPL analysis problems arising because of traceability and efficient methods of solving them. In the literature, there have been some attempts to identify and address these problems ([10, 21, 2, 7, 6, 11, 25, 17, 8, 9]). In this paper, we refine and enhance the results of these works through the following salient contributions:

- We propose a simple and abstract set-theoretic semantics of SPL with variability and traceability constraints which captures the core concepts precisely and succinctly, without the clutter usually associated with rich notational languages such as FD (Feature Diagram) [14], OVM (Orthogonal Variability Model) [23] etc.
- We provide a simple, intuitive and usable definition of traceability. This is used to define when an implementation *implements* a feature (also, sets of features). We show that existing definition of products as satisfying instances of SPL constraints is inadequate and therefore propose a tighter definition of products.
- We define a number of SPL analysis problems some of which are standard for feature variability but are now lifted to the entire SPL with traceability (e.g. live, common and dead elements), some others which arise because of the dichotomy of specifications and implementations (soundness and completeness, redundant, superfluous, extraneous etc).
- It is known that propositional satisfiability is not an efficient way of solving some of the analysis problems, since in certain cases one has to enumerate all the specifications and implementations. We show that the costly enumeration can be avoided by encoding these problems as Quantified Boolean Formulae (QBF), which are then checked for satisfiability using QSAT (Quantified SAT) solvers. We also provide some evidence of the scalability of QSAT for analysis problems in large SPLs : we consider the case study of *Electronic Shopping* [16] which has 290 features and more than a billion valid products.

1.1 Related Work

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 [7]. A variability management model that focuses on the traceability aspect between the notion of problem and solution spaces is presented in [6]. Anquetil et al.[2] formalize the traceability relations across problem and solution space and also across domain and product engineering. In [10], 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 [11]. Zhu et al.[25] define a traceability relation from requirements to features and also from features to architectures with consistency analysis. [21] presents a method to identify the traceability between feature model and architecture model. Czarnecki's work [13], [8], [9] on giving semantics to features in feature models by mapping them to other models has been found useful at the requirements level. In the above works, the treatment is either informal or does not address the role of traceability in the implementability aspect of SPL.

Borba's work [19] builds on the idea of automatic generation of products from assets, by relying on feature diagrams and configuration knowledge (CK) [13]. A CK relates features to assets specifying which assets specify or implement possible feature combinations. [19] lays theoretical foundations on refining and evolving SPLs. The notion of traceability in [19] is general; however, unlike [19], the focus of our paper is on the implementability of SPL. In [8] and [9], the authors propose a template-based approach for mapping feature models to annotated models expressed in UML/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. [9] 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 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 outperforms SAT based procedures (Table 3).

The paper that is closest to our work is that by Metzger et al. [17] and deserves a detailed comparison. In this paper, *PL variability* refers to the variations among the features of the system and *software variability* refers to the variations among the software system artifacts. In our paper, we follow a different terminology to bring out the product line hierarchy clearly (shown in Figure 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 paper, we give a simple set-theoretic semantics to SPL's in lieu of the visually appealing notations such as FD, VFD and OVM. The advantage is that in this semantics the core concepts, analysis problems and the solution methods can be expressed in a more clear, unambiguous and concise way.

The traceability among PL and software variability is represented in Metzger et al. using X-links. One type of X-links

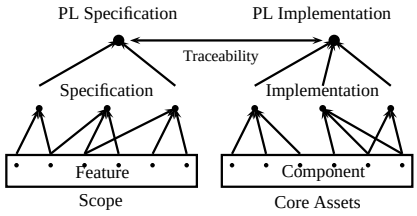

Figure 1: Product Line Hierarchy.

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 that paper are general propositional constraints. However, not all propositional constraints provide the intuitive and strong implementability relation between the implementations and specifications. The definition of traceability in our paper captures the above-mentioned class of constraints and is used to define a reasonable notion of an *implements* relation between implementations and specifications. 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. In 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 $Covers$ relation in our paper.

Metzger et al. propose a number of analysis problems; in the terminology of that paper, they are *realizability, internal competition, usefulness, flexibility* and *common and dead elements*. We have redefined these in our paper in 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 paper obviates this problem and gives efficient solution methods scalable to large, real-life case studies. Table 3 gives a comparison of SAT and QSAT approaches for the analysis operation *soundness* on a case study (MPPL in section 4) which had 38 components and 25 features. Soundness checks if each of the 2^{38} PL implementations covers some PL specification.

1.2 Outline of The Paper

In the following section, we give a formal definition of an SPL with traceability. It introduces the central notion of *implements* relation and the analyses we would like to carry out in SPL. In Section 3, we show how the analysis problems can be encoded in QBF and solved using QSAT solvers. We present in Section 4 some results of the analyses carried out using the QSAT solver tool CirQit [12] on two case studies : (i) Mobile Phone Product Line (MPPL) and (ii) Electronic Shopping Product Line (ESPL). Finally, we conclude in Section 5 with a summary of the paper and some future directions. The proof of the main result relating the analysis problems and QBF formulae is given in [1].

2. MODEL OF SPL : TRACEABILITY AND IMPLEMENTATION

Consider an SPL with three features, namely, Anti-Brake Skidding (ABS), Electronic Stability Control (ESC) and Lane Centering (LC). The set $\{ABS, ESC, LC\}$ forms the *scope* of the SPL. Variants of vehicles may be derived from different combination of these features. These different subsets, called *specifications* of a vehicle, are often decided by business reasons and form the *PL Specification* of the SPL. For example, the PL specification may contain $\{ABS, LC\}$, $\{ABS, ESC\}$ or $\{ABS, ESC, LC\}$. These features need several components for their implementation, e.g., feature specific sensors, control softwares, ECUs (Electronic Control Unit) and actuators. All these components together form the *core assets* of the SPL. A specific *implementation* consists of a set of components. For example, a skid sensor, a brake sensor, ABS control software and ABS ECU, brake actuator, lane sensor, lane centering control software and steering actuator together constitute an implementation. All the possible implementations (or, variants) in the SPL constitute the *PL implementation*.

Specification and Implementation. The set of all features found in any of the products in a productline defines the *scope* of the productline. We denote the scope of a productline by $\mathcal{F}$ with possible subscripts. A scope $\mathcal{F}$ consists of a set of features, denoted by small letters $f, g \ldots$. A specification is subset of features in the scope and denoted by $F, G, \ldots$. The core assets of a product line is denoted by $\mathcal{C}$.

Traceability. A feature is implemented using a non-empty subset of components in the core asset $\mathcal{C}$. For example, the feature ABS is implemented by a skid sensor, brake sensor, brake controller ECU and a brake actuator. 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.

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.*

EXAMPLE 2. *Consider the SPL $\Psi = (\overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T})$ with 3 features f_1, f_2, f_3 and 4 components c_1, c_2, c_3 and c_4, shown*

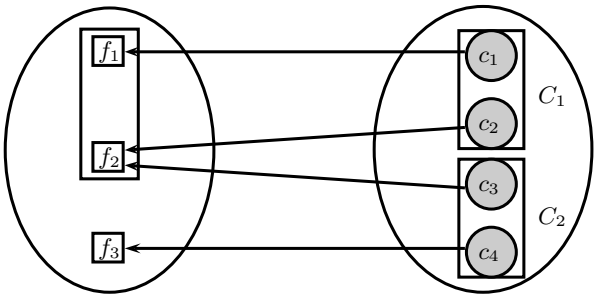

Figure 2: The Example SPL.

pictorially in Figure 2. The solid rectangles denote specifications and implementations. The hyper-edges from sets of components to features denote the traceability relation.

- $\overline{\mathcal{F}}$=$\{F_1 : \{f_1\}, F_2 : \{f_2\}, F_3 : \{f_1, f_2\}, F_4 : \{f_3\}\}$,
- $\overline{\mathcal{C}} = \{C_1 = \{c_1, c_2\}, C_2 = \{c_3, c_4\}\}$,
- $\mathcal{T}$: $f_1 \rightarrow \{\{c_1\}\}$, $f_2 \rightarrow \{\{c_2\}, \{c_3\}\}$, $f_3 \rightarrow \{\{c_4\}\}$.

From $\mathcal{T}$, it is clear that f_1 requires c_1 for its implementation, f_3 requires c_4 for its implementation, while c_2 or c_3 can implement f_2.

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 Example 2, f_1 is implemented by implementations C_1, f_2 is implemented by C_1 and C_2, and f_3 is implemented by C_2 but not by C_1.

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 Example 2, $Provided\_by(C_1) = \{f_1, f_2\}$ and $Provided\_by(C_2) = \{f_2, f_3\}$. With the basic definitions above, we can now define when an implementation exactly implements a specification.

Definition 3 (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_1 realizes the specification $\{f_1, f_2\}$, but it does not realize $\{f_1\}$ even though it provides the implementation of f_1. In many real-life usecases, due to the constraints on packaging of components, the exactness may be restrictive. For example, a *roll-over control* component is not necessary for an ABS feature but may be packaged in the stability control module by a component provider. In the absence of a choice, the integrator company has to buy the roll-over control which provides more features than is decided for a variant[1]. Hence, we relax the definition of *Realizes* in the following.

Definition 4 (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 that the scope $\mathcal{F}$ in Example 2 consisted of only two specifications $\{f_1\}$ and $\{f_2\}$. This models two variants with mutually exclusive features. The implementation C_1 implements $\{f_1\}$. Without the proviso, we would have $Covers(C_1, \{f_1\})$. However, since $Provided\_by(C_1) = \{f_1, f_2\}$, it actually implements both the features together, thus violating the requirement of mutual exclusion.

The set of products of the SPL are now defined as the specifications, and the implementation covering them through the traceability relation.

Definition 5 (SPL Products). *Given an SPL $\Psi = \langle \overline{\mathcal{F}}, \overline{\mathcal{C}}, \mathcal{T} \rangle$, the products of the SPL denoted $Prod(\Psi)$, is the set of all specification-implementation pairs $\langle F, C \rangle$ where $Covers(C, F)$.*

Thus, in Example 2, we see that among the potential 8 products (4 specifications $\times$ 2 implementations) the valid products are $\langle C_1, F_1 \rangle$, $\langle C_1, F_2 \rangle$, $\langle C_1, F_3 \rangle$, $\langle C_2, F_2 \rangle$ and $\langle C_2, F_4 \rangle$.

2.1 SPL Level Properties

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

1. 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) for the SPL is adequate to provide implementation 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 $Covers(C, F)$.

2. 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 $Covers(C, F)$.

3. Given a specification, we want to find out all the variant implementations in the architecture such that they cover the specification. This is given by a function $FindCovers(F) = \{C | Covers(C, F)\}$.

[1]This example is hypothetical. Usually, such extra components can be disabled by calibration parameters either by the provider or by the integrator.

4. At times, it is necessary for a premier set of features to be provided exactly for some product variants. In this case, we want to find out if there is an implementation which *realizes* the specification. A specification is *existentially explicit* if there is an implementation C such that $Realizes(C, F)$. Dually, it is *universally explicit* if for all implementations $C \in \overline{\mathcal{C}}$, $Covers(C, F)$ implies $Realizes(C, F)$.

5. A given specification may be implemented by multiple implementations. This may be a desirable criterion of the PL implementation 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 $|FindCovers(F)| = 1$.

6. Identification of common, live and dead elements in an SPL are some of the basic analyses identified 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 $\langle F, C\rangle \in Prod(\Psi)$, $e \notin F \cup C$.

7. There are 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 w.r.t. Ψ, whereas if either $\{a\}$ or $\{b\}$ is removed from the PL implemetation, the remaining implementation ($\{b\}$ or $\{a\}$) is not superfluous anymore (w.r.t. the reduced SPL).

8. A component is *redundant* if it does not contribute to any feature in any implementation in the PL implementation. 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.

9. 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 $C \in \overline{\mathcal{C}}$, $(c \notin C \Rightarrow \neg implements(C, f))$. This definition can be extended to specifications as well: a component c if *critical* for a specification F, if for all implementations $C \in \overline{\mathcal{C}}$, $(c \notin C \Rightarrow \neg Covers(C, F))$.

10. 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 difference choices of implementation and extraneous features pairs :
    $Extra(F) \equiv \{\langle C, Provided\_by(C) \setminus F\rangle | Covers(C, F)\}$.

3. SPL ANALYSIS PROBLEMS

In the literature, different analysis problems in SPL are usually encoded as satisfiability problems for propositional constraints[3] and SAT solvers such as Yices[24], Bddsolve[4] are used to solve them. As has been noted in [17], 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 and that the more expressive logic formalism of Quantified Boolean Formulae (QBF) is necessary to encode the analysis problems. QBF is generalized form of propositional formulae with quantification (existential and universal) over the propositional symbols. The boolean satisfiability problem for propositional formulae is then naturally extended to QBF satisfiability problem (QSAT).

Let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be the core assets and let $\mathcal{F} = \{f_1, \ldots, f_m\}$ be the scope of the SPL. Each feature and component x is encoded as a propositional variable p_x. Given an implementation C, $\widehat{C}$ denotes the formula $\bigwedge_{c_i \in C} p_{c_i}$, and $\bar{C}$ denotes a bitvector where $\bar{C}[i] = 1$ (TRUE) if $c_i \in C$ and 0 (FALSE) otherwise. Similarly, for a specification F, we have $\widehat{F}$ and $\bar{F}$.

Let CON_F (resp. CON_I) denote the set of constraints over the propositional variables capturing the PL specification (resp. PL implementation). Given the PL specification and implementation as sets, it is straightforward to get these constraints. When one uses richer notations like FD etc, one can extract these constraints following [3]. For the traceability, the encoding CON_T is as follows. Let f be a feature and let $\mathcal{T}(f) = \{C_1, C_2 \ldots, C_k\}$. We define $formula\_\mathcal{T}(f)$ as $\bigvee_{j=1..k} \bigwedge_{c_i \in C_j} p_{c_i}$. If the set $\mathcal{T}(f)$ is undefined(empty), then $formula\_\mathcal{T}(f)$ is set to FALSE. CON_T is then defined as $\bigwedge_{f_i \in \mathcal{F}} [formula\_\mathcal{T}(f_i) \Rightarrow p_{f_i}]$ for the features f_i for which $\mathcal{T}(f_i)$ is defined and FALSE if $\mathcal{T}(.)$ is not defined for any feature.

The implementation question whether $implements(C, f)$ is now answered by asking whether the formula $\widehat{C}$ for the set of components C alongwith the traceability constraints CON_T can derive the feature f. This is equivalent to asking whether $\widehat{C} \wedge \text{CON}_T \wedge (\neg p_f)$ is UNSAT. Since it is evident that only $\mathcal{T}(f)$ is used for the implementation of f, this can further be optimized to $\widehat{C} \wedge (formula\_\mathcal{T}(f) \Rightarrow p_f) \wedge (\neg p_f)$. However, as we will see later, since $implements(.,.)$ is used as an auxiliary function in the other analyses, we want to encode it as a formula with free variables. Thus, $form\_implements_f(x_1, \ldots, x_n)$ is a formula which takes n boolean values (0 or 1) as arguments, corresponding to the bitvector $\bar{C}$ of an implementation C and evaluates to either TRUE or FALSE.

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

This forms the core of encoding for all the other analyses. Hence the correctness of this construction is crucial. Lemma 1 states the correctness result. The proof is given in [1].

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.*

In order to extend the construction to encode $Covers$, we construct a formula $f\_covers(x_1, \ldots, x_n, y_1, \ldots, y_m)$ where, the first n boolean values encode an implementation C and the subsequent m boolean values encode a specification F. The formula evaluates to TRUE iff $Covers(C, F)$ holds.
$f\_covers(x_1, \ldots, x_n, y_1, \ldots, y_m) =$
$\bigwedge_{i=1}^{m}[y_i \Rightarrow form\_implements_{f_i}(x_1, \ldots, x_n)]$.

Similarly, we have $f\_realizes(x_1, \ldots, x_n, y_1, \ldots, y_m) =$

$$\bigwedge_{i=1}^{m}[y_i \Leftrightarrow form\_implements_{f_i}(x_1, \ldots, x_n)].$$

Notice the replacement of $\Rightarrow$ in $f\_covers(..)$ by $\Leftrightarrow$ in $f\_realizes(..)$.

In Example 2, we ask whether $Covers(\{c_1, c_2\}, \{f_1\})$. Since there are 4 components c_1, c_2, c_3 and c_4, and 3 features f_1, f_2 and f_3, this translates to the formula $f\_covers(1, 1, 0, 0, 1, 0, 0)$, which, after simplification boils down to $form\_implements_{f_1}(1, 1, 0, 0)$. Since $formula\_\mathcal{T}(f_1) = p_{c_1} \wedge p_{c_2}$, after simplification we get $\forall p_{c_1} \ldots p_{c_4}\{(1 \Rightarrow p_{c_1} \wedge 1 \Rightarrow p_{c_2}) \Rightarrow (p_{c_1} \wedge p_{c_2})\}$. Since this is true, we conclude that $Covers(\{c_1, c_2\}, \{f_1\})$ holds.

In order to demonstrate a negative example, we ask whether $Covers(\{c_1, c_2\}, \{f_3\})$. We simplify the encoded formula $f\_covers(1, 1, 0, 0, 0, 0, 1)$, with $formula\_\mathcal{T}(f_3) = p_{c_4}$. This yields $form\_implements_{f_3}(1, 1, 0, 0)$ and at last, $\forall p_{c_1} \ldots p_{c_4}\{(1 \Rightarrow p_{c_1} \wedge 1 \Rightarrow p_{c_2}) \Rightarrow p_{c_4}\}$. Since this is FALSE (for say, $p_{c_1} = 1$, $p_{c_2} = 1$ and $p_{c_4} = 0$), we conclude correctly that $Covers(\{c_1, c_2\}, \{f_3\})$ does not hold.

We encode the other analysis problems as QBF formulae as shown in Table 1. The correctness of the encoding is asserted by the following theorem. In the theorem, for the constraint CON_I, $\text{CON}_I[q_{c_1}, \ldots, q_{c_n}]$ denotes the same constraint where each propositional variable p_{c_i} has been replaced by a new propositional variable q_{c_i}.

THEOREM 6. *Given an SPL Ψ, each of the properties listed in Table 1 holds good iff the corresponding formula evaluates to true.*

PROOF. The proof can be seen at [1]. □

4. IMPLEMENTATION, CASE STUDIES

In order to verify the satisfiability of the QBF formulae encoding the analysis problems, we represent them in a tool input format called QPRO [22]. The format is a standard input file format in non-prenex, non-CNF form. We use the tool CirQit [12] to check the satisfiability of QBFs for SPL analysis. 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 [20]. It accepts QBFs in QPRO format and returns TRUE if the formula is satisfiable, and FALSE otherwise. We have developed a prototype implementation that takes as input an SPL; any of the given analysis operations can be selected thereafter: the corresponding formulae are automatically generated in QPRO format. The CirQit tool is then invoked to check the satisfiability. Our prototype implementation *Software Product Line Engine (SPLE)* can be seen at [15], while Figure 3 gives a screen shot of SPLE.

In order to illustrate the SPL analysis method described in the paper, we considered two examples : (i) Mobile Phone Product Line (MPPL) [16] used in the Mobile Phone companies, as well as (ii)Electronic Shopping Product Line (ESPL) [16]. MPPL was taken from [16] (the Feature Diagram : Mobile Phone). This gives us the scope $\mathcal{F}$ as well as the set of PL specifications $\overline{\mathcal{F}}$ required by our notion of an SPL. To provide the PL implementation $\overline{\mathcal{C}}$ corresponding to this, we designed an architecture diagram (see below) as well as a traceability relation.

MPPL : Feature Diagram. Figure 4 presents the feature diagram of the MPPL. Rectangles are used to represent features of the MPPL. The *mandatory* features are represented by dark rectangles and filled circles on top. The optional features are represented by dotted rectangles and hollow circles on top. The features *Utility Functions*, *Setting*, *OS*, etc., are mandatory features and *Connectivity*, *media*, etc., are optional features. The *alternative* features are represented by hollow arc relationship between parent features and child features. In an alternative relationship, when a parent feature is part of any specification, then its mandatory to select exactly one child feature. For example, in case the *OS* feature is selected, it is mandatory to select exactly one from *Symbian* or *WinCE*. The *Or* features are represented by filled arc relationship between parent features and child features. In an Or relationship, when a parent feature is part of any specification, it is mandatory to select at least one child feature. If *connectivity* feature is part of any specification, one must select at least one feature from *Bluetooth*, *USB* or *Wifi*. The *requires* cross-tree constraint between the *Games* feature and *Java Support* feature denotes that the product with the *Games* feature must include the *Java Support* feature. The *excludes* cross-tree constraint between *MP3* and *MP4* denotes that the product with the *MP3* feature must not select the *MP4* feature and vice versa.

MPPL : Architecture Diagram. Figure 5 represents the architecture of MPPL. The given architecture diagram is a kind of feature diagram. It consists of *mandatory* and *optional* components. The mandatory components are represented by solid rectangles and optional components are represented by dotted rectangles. The components *Transmitter*, *Receiver*, etc. are mandatory components and *GPRSApps*, *MP3Apps*, etc. are optional components. The mandatory relationship between two components is represented by solid lines and optional relationship is represented by dotted lines.

Properties	Formula
$Implements(C, f)$ $\bar{C} = (v_1, \ldots, v_n)$	$form\_implements_f(v_1, \ldots, v_n)$
$Covers(C, F)$, $C = (v_1, \ldots, v_n)$ $Realizes(C, F)$, $\bar{F} = (u_1, \ldots, u_m)$	$f\_covers(v_1, \ldots, v_n, u_1, \ldots, u_m)$ $f\_realizes(v_1, \ldots, v_n, u_1, \ldots, u_m)$
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	$\forall p_{c_1} \ldots p_{c_n}[form\_implements_{f_j}(p_{c_1} \ldots p_{c_n}) \Rightarrow p_{c_j}]$

Table 1: Properties and Formulae

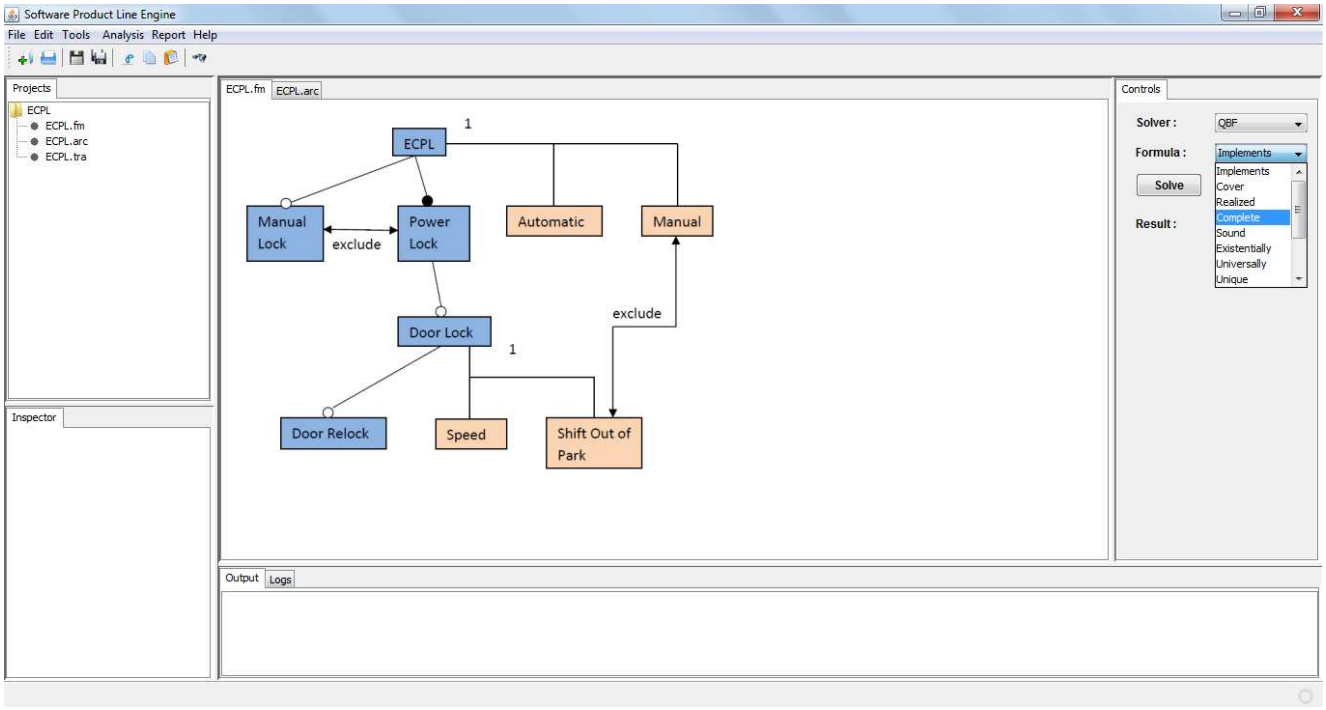

Figure 3: A Screenshot of SPLE

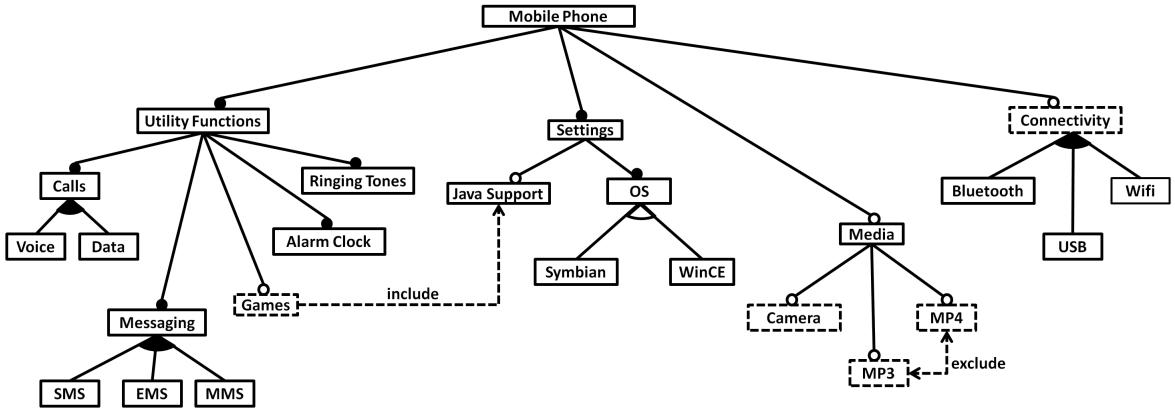

Figure 4: Feature Diagram of MPPL.

The relationship with value *1* denotes that we have to select exactly one component from the given set of components. In the given architecture diagram, we can select only one *OS setup* i.e. $SymbianOSSetup$ or $WinCEOSSetup$. The relationship with value *1+* denotes that we have to select at least one component from the given set of components. If we select $ConnectivityInterface$ and $ConnectivityAdapter$ components then it is mandatory to select at least one component from $\{BluetoothHwd, USBHwd, WifiHwd\}$. In the MPPL, there are a total of 25 features, 38 components and more than 7000 valid products.

MPPL : Traceability relations. Table 2 presents the traceability relation of MPPL.

Feature	Components
$UtilityFunctions$	$\{\{UtilInterface, UtilAdapter\}\}$
$Calls$	$\{\{Transmitter, Receiver\}\}$
$Voice$	$\{\{Keypad, Mice, Speaker\}\}$
$Data$	$\{\{Keypad, GPRSApps\}\}$
$Messaging$	$\{\{MessageInterface, MessageAdapter\}\}$
SMS	$\{\{SMSApps\}\}$
EMS	$\{\{EMSApps\}\}$
MMS	$\{\{MMSApps\}\}$
$Games$	$\{\{GamesInterface, GamesAdapter\}\}$
$AlarmClock$	$\{\{AlarmApps\}\}$
$RingingTones$	$\{\{RingTonesApps\}\}$
$Settings$	$\{\{SettingInterface, SettingAdapter\}\}$
$JavaSupports$	$\{\{MIDP2.0\}, \{MIDP3.0\}\}$
OS	$\{\{SymbianOSSetup\}, \{WinCEOSSetup\}\}$
$Symbian$	$\{\{SymbianOSSetup\}\}$
$WinCE$	$\{\{WinCEOSSetup\}\}$
$Media$	$\{\{MediaInterface, MediaAdapter\}\}$
$Camera$	$\{\{VGA\}, \{MP1.3\}, \{MP2.0\}, \{MP3.2\}, \{MP5.0\}\}$
$MP3$	$\{\{MP3Apps\}\}$
$MP4$	$\{\{MP4Apps\}\}$
$Connectivity$	$\{\{ConnectivityInterface, ConnectivityAdapter\}\}$
$Bluetooth$	$\{\{BluetoothHwd\}\}$
USB	$\{\{USBHwd\}\}$
$Wifi$	$\{\{WifiHwd\}\}$

Table 2: Traceability Relation of MPPL.

Table 3 compares the performances of SAT and QSAT based encoding of the property *soundness*, which checks whether every PL implementation covers some PL specification. The QBF formulae for soundness can be seen in Table 1 : there is a block of 38 universal quantifiers corresponding to the 38 components in MPPL; this when unrolled in SAT would require us to check the property for all the 2^{38} implementations. Table 3 shows that encoding in QBF is far more efficient: note that it took just .023 s in QSAT to answer this, while we had to abort after a day's wait in the case of the SAT encoding.

Number of PL implementations	SAT Time(sec)	QSAT Time(sec)
1	0.016s	$\cong 0$
10	0.022s	$\cong 0$
500	0.237s	$\cong 0$
1000	0.350s	$\cong 0$
2000	0.433s	$\cong 0$
5000	0.848s	$\cong 0$
10000	1.571s	$\cong 0$
20000	2.701s	$\cong 0$
40000	7.384s	$\cong 0$
60000	14.134s	$\cong 0$
2^{38}	> 1 day	0.023s

Table 3: MPPL soundness : SAT vs. QSAT performances

The Electronic Shopping Product Line (ESPL) is a much bigger product line (the Feature Model : Electronic Shopping from [16]) with 290 features, and more than a billion valid products. We enhanced this with an archirecture diagram comprising of 290 components, and a 1-1 traceability relation, thereby preserving all the valid products. On both MPPL and ESPL, we checked the analysis operations; all the analyses were carried out using the QBF encoding of the previous section and application of the CirQit tool. We have recorded the time required to check the satisfiability of the formulae in Table 4. To compare with the SAT based approach, we unrolled the QBF formulae into SAT formulae for the analysis operations other than *Implements, Realizes* and *Covers* and checked their satisfiability using a SAT solver; in the case of ESPL, we had to abort after a day's wait for all the operations. This was expected, after looking at Table 3 for the operation of *soundness* in the case of MPPL.

5. CONCLUSION

In this paper, we have proposed a semantic formulation for Software Product Lines using elementary set theory, in order to give precise and unambiguous definitions for traceability between the specifications and implementations. Based on this, a new definition for products in an SPL is proposed. We show that our definition is different from the satisfiability based definition and that it captures the intricacies of the implementation relation in the correct way. With this foundation, we define a set of analysis problems for the SPLs.

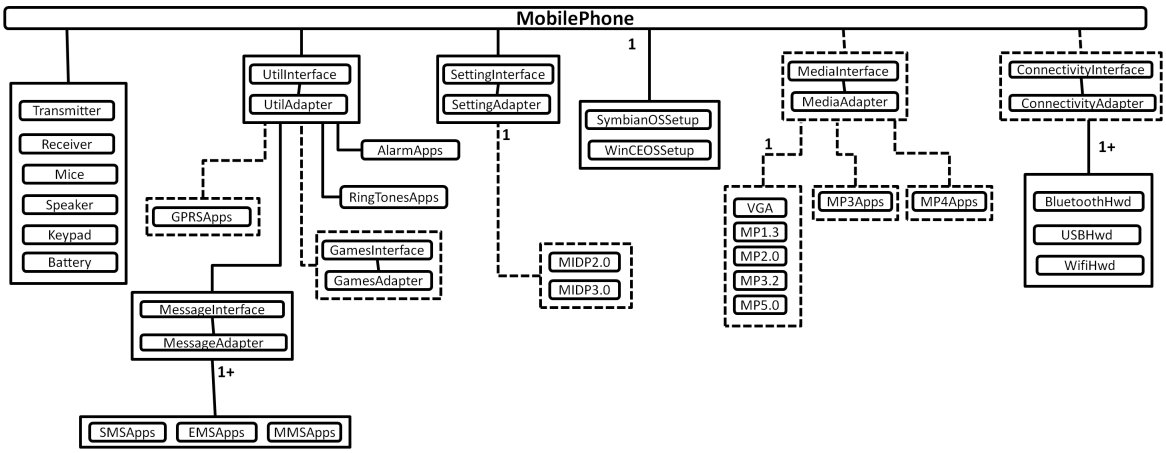

Figure 5: Architecture Diagram of MPPL.

Properties	MPPL Avg. Time(sec)	ESPL Avg. Time(sec)
Implements	.0017	.014
realizes	.043	.423
covers	.021	.018
complete	.025	4.68
sound	.023	4.22
existentially explicit	.061	0.862
universally explicit	.127	5.42
common	.007	0.154
live	.0226	0.145
dead	.008	0.153
critical	.0013	.015

Table 4: Time Complexity for Properties and Formulae

We show that these problems can be formulated as QBF satisfiability and can be solved using QSAT tools such as CirQit.

We have demonstrated the feasibility of our approach through a small fragment of an industrial SPL. Since QSAT problem is PSPACE-complete, generic QSAT solving may not scale well. However, one observes that the formulas for the SPL analyses have very specific structure which can be exploited for efficient QSAT solving. In fact, while a detailed study of the scalability of our approach is under way, our preliminary experiments give optimistic results.

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

6. REFERENCES

[1] http://www.cse.iitb.ac.in/∼krishnas/tr2012.pdf, 2012.

[2] N. Anquetil, B. Grammel, I. G. L. da Silva, J. A. R. Noppen, S. S. Khan, H. Arboleda, A. Rashid, and A. Garcia. Traceability for model driven, software product line engineering. *ECMDA Traceability Workshop Proceedings, Berlin, Germany*, pages 77–86, Norway, June 2008. SINTEF.

[3] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[4] BDDSolve. http://www.win.tue.nl/ wieger/bddsolve/, 2010.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortï£¡s. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6):615–636, 2010.

[6] K. Berg, J. Bishop, and D. Muthig. Tracing software product line variability: from problem to solution space. *SAICSIT '05: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 182–191, , Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[7] D. Beuche, H. Papajewski, and W. Schrï£¡der-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333 – 352, 2004. Software Variability Management.

[8] K. Czarnecki and M. Antkiewicz. A template based approach based on superimposed variants. *Proceedings of GPCE'05*, pages 422–437, 2005.

[9] K. Czarnecki and K. Pietroszek. Verifying feature based model templates against well-formedness ocl constraints. *Proceedings of GPCE'06*, pages 211–220, 2006.

[10] J.-M. DeBaud and K. Schmid. A systematic approach to derive the scope of software product lines. *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 34–43, New York, NY, USA, 1999. ACM.

[11] T. Eisenbarth, R. Koschke, and D. Simon. A formal method for the analysis of product maps. *Requirements Engineering for Product Lines Workshop, Essen, Germany*, 2002.

[12] A. Goultiaeva and F. Bacchus. http://www.cs.utoronto.ca/ alexia/cirqit/, 2010.

[13] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[14] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. A feature-oriented reuse method with domainspecific reference architectures. *Annals of Software Engineering*, 5(1):143âĂŞ168, 1998.

[15] http://www.cse.iitb.ac.in/∼krishnas/SPLE.zip, 2012.

[16] D. C. Marcilio Mendonca, Moises Branco. s.p.l.o.t. - software product lines online tools. *Proceedings of OOPSLA'09*, 2009.

[17] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 243–253, 2007.

[18] S. E. I. of Carnegie Mellon University. Software product line web site, - 2010.

[19] P. Borba, L. Teixeira and R. Gheyi. A theory of software product line refinement. *Theoretical Computer Science*, 2012. doi:10.1016/j.tcs.2012.01.031.

[20] C. Peschiera, L. Pulina, A. Tacchella, U. Bubeck, O. Kullmann, and I. Lynce. The seventh qbf solvers evaluation (qbfeval'10). In O. Strichman and S. Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 237–250. Springer, 2010.

[21] T. K. Satyananda, D. Lee, S. Kang, and S. I. Hashmi. Identifying traceability between feature model and software architecture in software product line using formal concept analysis. *Computational Science and its Applications, International Conference*, 0:380–388, 2007.

[22] M. Seidl. http://www.qbflib.org/format_qpro.pdf, 2009.

[23] P. Stan Bühne, Kim Lauenroth and Klaus Pohl. Modelling requirements variability across product lines. *Proceedings of RE'05*, pages 41–52, IEEE Computer Society, 2005.

[24] YICES. http://yices.csl.sri.com/, 2010.

[25] C. Zhu, Y. Lee, W. Zhao, and J. Zhang. A feature oriented approach to mapping from domain requirements to product line architecture. In *Software Engineering Research and Practice*, pages 219–225, 2006.