Safe Composition of Configuration Knowledge-based Software Product Lines

Leopoldo Teixeira, Paulo Borba
Informatics Center
Federal University of Pernambuco
Email: {lmt,phmb}@cin.ufpe.br

Rohit Gheyi
Department of Computing Systems
Federal University of Campina Grande
Email: rohit@dsc.ufcg.edu.br

***Abstract*—Feature models and configuration knowledge drive product generation in a Software Product Line (SPL). Mistakes when specifying these models or in the implementation might result in ill-formed products — the safe composition problem. This work proposes an automated approach for verifying safe composition for SPLs with explicit configuration knowledge models. We translate feature models and configuration knowledge into propositional logic and use SAT Solvers to perform the verification. We evaluate our approach using seven releases of the MobileMedia SPL, which generate up to 272 products in the 7th release. We report safe composition problems related to non-conformity with the feature model, bad specification of the configuration knowledge, and implementation not envisioning the full SPL scope, that affect over 40% of the products in the 7th release.**

I. Introduction

A software product line (SPL) is defined as a set of software systems that share common characteristics, but are sufficiently distinct from each other [1]. In a SPL, products are generated from reusable assets [1]. To accomplish this, SPL approaches use, in most cases, Feature Models (FMs) [2], which describe a domain by representing common and variable features of an SPL; and Configuration Knowledge (CK) [3], which relates features to implementation assets, driving product generation. We can explicit such knowledge in a model [4], [5], [6], in annotations over SPL assets [7], [8], or it can be implicit by the implementation mechanism followed, such as feature modules or containment hierarchies [9], [10], [11], [12]. Errors when specifying this model can result on incorrect products. Thus, the safe composition problem [8], [10], [13], [14].

Safe composition is related to safe generation and the verification of properties for SPL assets: i.e., providing guarantees that the product derivation process generate products with particular properties [8], [10], [13], [14]. To keep the SPL consistent and to detect errors early, it is important to make sure that all products obey such properties. Since SPLs can quickly scale to hundreds of products, it is often impractical, time-consuming and error-prone to manually inspect FM, CK, and implementation to understand dependencies between assets for all products. Another option is to generate all products, compile, and test them. While this is an useful and safe approach, it does not scale. For example, in release 7 of the MobileMedia SPL [15], it would take around 4 hours *only to compile* all 272 products. Additionally, anytime that something changes, we would need to compile all products again.

Some approaches avoid the need for synthesizing the entire SPL to verify safe composition [10], [11], [12], [13]. However, they are tailored to specific languages for implementing SPLs. To enable multiple languages, we can use a dedicated CK model. Such model relates features and their combinations to assets [4], [5]. Assets might be classes, aspects, configuration files, and so on. CK evaluation against a valid feature selection drives product generation. Such evaluation yields the set of assets needed to build a product. However, mistakes made when specifying this model might result in safe composition problems. For example, missing entries in the CK lead to missing assets in the resulting product.

In this paper we present an approach to verify safe composition of SPLs with such dedicated CK models, that we call CK-based SPLs. To enable the verification, we use explicit interfaces in the CK, expressing dependencies between assets. Such dependencies are extracted from the assets, and need to be satisfied for all SPL products. To actually perform the verification, we use the formal specification language Alloy [16] and its associated tool support — Alloy Analyzer — which provides automatic analysis of Alloy specifications. Our tool automatically translates the FM and CK to our Alloy encoding. When it detects safe composition problems in a SPL, it reports the list of problematic products based on counterexamples returned by the Alloy Analyzer.

We evaluate our approach using the first seven releases of the Mobile Media SPL [15][1], which handles different media types in mobile devices. The FM is available from documentation and the CK is defined in the build files used for compiling SPL products. For releases 1-5 which are smaller SPLs (in release 5: 16 products, 14 features, where 4 are optional), the tool does not report safe composition problems. However, for releases 6 and 7, containing 48 and 272 products, respectively, it identifies safe composition problems. Problems relate to different sources: non-conformity with the FM, bad CK specification, and implementation not envisioning the full SPL scope. In release 6, time for verifying and reporting all problematic products is around 4 seconds, while in release 7, it takes 34 seconds, which is less than the average time for

[1]Detailed results at http://www.cin.ufpe.br/~lmt/sbes2011/

compiling a single MobileMedia product. Errors found affect almost 17% of the products in release 6 and almost 43% of the products for release 7.

In our earlier work, we presented a general theory for SPL refinement [6]. In it, we define an SPL as a tuple formed by FM, CK, and code assets, in which all products that can be generated are well-formed — safe composition. In this work, we propose and implement a way to check well-formedness. This is important as an initial step towards tool support for SPL refactoring.

In summary, the main contributions of this work are the following:

- Extension of an existing CK model [4], [5] to enable safe composition verification (Section III);
- An approach for verification of CK-based SPLs (Section IV-A);
- Tool support implementation, using the Alloy Analyzer, to detect errors in an SPL and to report ill-formed products (Section IV-B);
- Evaluation of the proposed approach using seven releases of an SPL (Section V).

II. Motivating Example

To enable automatic generation of products, we need to relate features to assets. This is established by the *Configuration Knowledge* (CK) [3]. For instance, consider the Expression Product Line (EPL), an SPL where products consist of interpreters for evaluating arithmetic expressions [17]. Figure 1 displays the FM for EPL. A FM represents, in terms of features, the possible choices that the customer can make for a product. That is, the possible products which the SPL can generate. It In this case, it allows 6 product configurations, which are valid feature selections. Each has at least one operation (**Add** or **Sub**) and must work with one specific type only (**Integer** or **Double**).

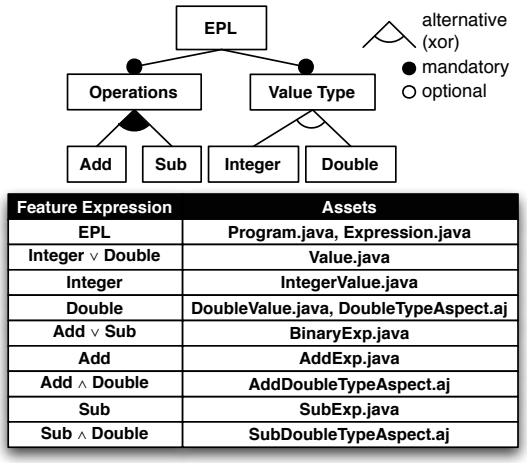
We use a separated model (CK) to relate *feature expressions* to assets [4], [5]. Figure 1 presents the EPL CK. In this case, assets are classes and aspects. For instance, in the seventh row of the CK in Figure 1, we see **AddDoubleTypeAspect** related to the joint selection of the **Add** and **Double** features. This aspect intercepts the program during the **Add** operation, changing the type from **Integer** to **Double**. CK Evaluation against a product configuration yields a set of assets, used to build the product. As an example, evaluating this CK for the product configuration

{**EPL, Value Type, Double, Operations, Add**}

yields the following set of assets:

{**Program.java,Expression.java,Value.java, DoubleValue.java,DoubleTypeAspect.aj,AddExp.java, BinaryExp.java,AddDoubleTypeAspect.aj**}.

Depending on the approach adopted for SPL development [18], there are different times and roles in charge of specifying the CK. For example, if we use the proactive approach [18], where a good portion of the SPL is designed



Feature Expression	Assets
EPL	Program.java, Expression.java
Integer ∨ Double	Value.java
Integer	IntegerValue.java
Double	DoubleValue.java, DoubleTypeAspect.aj
Add ∨ Sub	BinaryExp.java
Add	AddExp.java
Add ∧ Double	AddDoubleTypeAspect.aj
Sub	SubExp.java
Sub ∧ Double	SubDoubleTypeAspect.aj

Fig. 1. Feature Model and Configuration Knowledge for the Expression PL.

up front, an architect specifies in the CK how assets implement features and their interaction. In the case of reactive and extractive approaches [18], the developer responsible for implementing the feature also specifies and updates the CK.

If not properly specified, a CK might lead to safe composition problems [10], [11], [13], [14]. For example, we might forget to include an entry, include incompatible entries in it. In the set above, **AddDoubleTypeAspect** references the **AddExp** class, which depends on **IntegerValue**. Figure 2 illustrates this dependency, relating assets to their respective feature expression. From the FM, we know that products cannot contain both integer and double types. Consequently, products with the **Double** feature miss the **IntegerValue** class, since it is only associated with the **Integer** feature. We consider them ill-formed products, since an asset name (**AddExp**) present in the set yielded by CK evaluation references other asset name (**IntegerValue**), absent in that set.

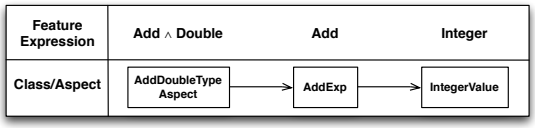


Fig. 2. Dependency diagram between some classes and aspects of the Expression Product Line and the relation with feature expressions on the CK.

It is undesirable to generate ill-formed products, but it might happen due to mistakes when designing and implementing a SPL, such as mentioned above. A brute-force approach of generating all SPL products for verifying safe composition is often impractical, since there are SPLs that can generate thousands of products [19]. Even in SPLs with fewer products, this might be expensive, as we discuss in Section V-E. To ensure productivity and avoid unexpected maintenance costs when developing a SPL, it is important to verify safe composition of SPLs without synthesizing the entire SPL [10], [13]. Next section describes our general approach for safe composition of CK models, illustrating how we could use it to detect the

problem presented in this section.

III. Safe Composition of CK-based Software Product Lines

To detect problems such as the ones mentioned in the previous section, we now introduce our approach for safe composition of CK-based SPLs. We use explicit interfaces for CK items (rows) to enable the verification. Therefore, for each CK item we have provided and required interfaces. Interfaces are explicit to comprehend semantic dependencies and other restrictions, like the inclusion of an asset depending on the final size of the generated product, besides typing dependencies, which can be inferred from existing assets, as in [10], [13]. Figure 3 illustrates another view of the CK in Figure 1. We now have columns for the provided and required interfaces. In this example, interfaces are asset names[2], representing dependencies between assets. When evaluating the CK against a product configuration, we can also yield the provided and required interfaces.

Feature Expression	Assets	Provided	Required
EPL	Program.java, Expression.java	Program, Expression	Expression, Value
Integer ∨ Double	Value.java	Value	Expression
Integer	IntegerValue.java	IntegerValue	Value
Double	DoubleValue.java, DoubleTypeAspect.aj	DoubleValue, DoubleTypeAspect	DoubleValue, Expression, Value
Add ∨ Sub	BinaryExp.java	BinaryExp	Expression
Add	AddExp.java	AddExp	BinaryExp, Expression, IntegerValue, Value
Add ∧ Double	AddDoubleTypeAspect.aj	AddDoubleTypeAspect	DoubleTypeAspect, AddExp, DoubleValue, Expression, Value
Sub	SubExp.java	SubExp	BinaryExp, Expression, IntegerValue, Value
Sub ∧ Double	SubDoubleTypeAspect.aj	SubDoubleTypeAspect	DoubleTypeAspect, SubExp, DoubleValue, Expression, Value

Fig. 3. Extended CK for the Expression Product Line, with required and provided interfaces.

So, for EPL, in the case of the product configuration

{**EPL, Value Type, Double, Operations, Add**},

discussed in Section II, CK evaluation yields the following set of provided interfaces:

{**Program, Expression, Value, DoubleValue, DoubleTypeAspect, BinaryExp, AddExp, AddDoubleTypeAspect**}.

The set of required interfaces yielded by CK Evaluation for the same product configuration is:

{**Expression, Value, DoubleValue, BinaryExp, *IntegerValue*, DoubleTypeAspect, AddExp**}.

When comparing yielded interfaces for this product, it is noticeable that it requires ***IntegerValue***, which is not provided. Therefore, we consider this product ill-formed, due to this non-satisfied dependency. A possible solution to this problem would be to refactor the implementation to remove this dependency. Another solution that would not involve changing the implementation would be to associate the ***IntegerValue*** class with the feature expression **Add** $\wedge$ **Double**.

[2]For simplicity and space, we remove file extensions from names on the provided and required columns.

With such interfaces, we are able to verify safe composition of a product without building it. We only need to verify that, after evaluating the CK for the product, all required interfaces are being provided. Therefore, according to previous works [5], [6], we consider an SPL well-formed when all of its products are well-formed.

A good part of these interfaces can be automatically derived from assets, as previous works already discuss [10], [13], [14], [12]. The derivation process is an orthogonal concern, and it is not the focus of this work. Such process can also have limitations, when dealing with semantic dependencies. For instance, in the case of dynamic class loading, where we only know values at runtime. Besides that, we can implement an SPL using different kinds of files, such as configuration files, property files, where it might not be possible to extract all interfaces automatically. Finally, these interfaces also depend on the variability implementation mechanism used in the SPL. Therefore, this work is not concerned with interfaces extraction, since we adopt a CK notion that supports different implementation languages and artifacts. To perform the verification, we assume that these interfaces are available, whether automatically derived from the assets or manually informed.

IV. Verification of CK-based Software Product Lines

This section presents our approach for the automated verification of safe composition of CK-based SPLs, formalizing the concepts presented in Section III. We translate both the feature model (FM) and the configuration knowledge (CK) into propositional formulae to evaluate whether a given SPL is well-formed. We use Alloy [16], and the Alloy Analyzer to encode FM and CK, perform the verification and report the list of ill-formed products.

A. Intuition

While the FM gives us the domain constraints, represented by the set of product configurations, the CK gives us the implementation constraints, represented by provided and required interfaces. Figure 3 presents the extended CK for EPL. As discussed, it associates feature expressions, like **Add** $\wedge$ **Double**, to assets, such as **AddDoubleTypeAspect**. We can translate the CK to a proposition. For each CK item, we generate provided and required propositions. Figure 4 illustrates the codification rules. For example, for the CK item associating the joint selection of **Add** and **Double** features, the proposition representing the provided interface is

$$(Add \wedge Double) \Rightarrow AddDoubleTypeAspect.$$

The idea is that if the feature expression $Add \wedge Double$ evaluate to true, $AddDoubleTypeAspect$ is provided. We translate required interfaces likewise. For the same CK item, the proposition representing the required interfaces is

$$(Add \wedge Double) \Rightarrow (DoubleTypeAspect \wedge AddExp \wedge DoubleValue \wedge Expression \wedge Value).$$

We use the conjunction ($\bigwedge$) of the provided propositions for each CK item to express the provided interfaces for the entire

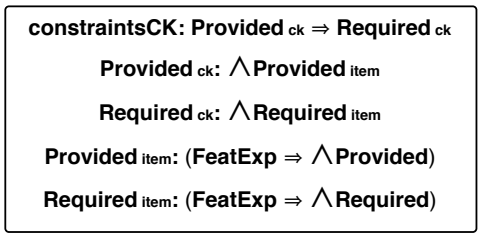


Fig. 4. Propositional logic codification rules for CK.

CK. We build the required proposition in the same way. We then relate both propositions to represent the CK constraints (*constraintsCK*). The idea is that for all products, required interfaces should be provided.

Using this codification, a product is well-formed when it satisfies $constraintsCK$. We can use propositions to represent product configurations, where we express non-selected features with the negation symbol. For example, using the FM in Figure 1, we can represent the product configuration

{**EPL, Value Type, Integer, Operations, Add**}

with the proposition below:

$$EPL \wedge Operations \wedge Add \wedge \neg Sub \wedge ValueType \wedge Integer \wedge \neg Double.$$

The *constraintsCK* proposition for EPL, when checked against this product configuration, simplifies to

$$(Program \wedge Expression \wedge Value \wedge IntegerValue \wedge BinaryExp \wedge AddExp) \Rightarrow (Expression \wedge Value \wedge BinaryExp \wedge IntegerValue).$$

Notice that after the simplification, for feature expressions evaluated to false, required and provided interfaces are eliminated from the proposition. For example, since we did not selected the **Double** feature, $AddDoubleTypeAspect$ does not appear in the left side of the implication — provided interfaces. We also see that all interfaces required for this product (right side of the implication) are provided (left side of the implication). Therefore, the whole proposition evaluates to true, which means that assets can be safely composed for the product. Thus, it is considered a well-formed product.

For the same SPL, if we evaluate the product configuration

{**EPL, Value Type, Double, Operations, Add**},

represented by the logical proposition

$$EPL \wedge Operations \wedge Add \wedge \neg Sub \wedge ValueType \wedge \neg Integer \wedge Double,$$

the *constraintsCK* proposition is then simplified to

$$(Program \wedge Expression \wedge Value \wedge DoubleValue \wedge DoubleTypeAspect \wedge BinaryExp \wedge AddExp \wedge AddDoubleTypeAspect) \Rightarrow (Expression \wedge Value \wedge BinaryExp \wedge \textbf{IntegerValue} \wedge DoubleTypeAspect \wedge AddExp \wedge DoubleValue).$$

We now see that **IntegerValue** appears on the right side of the implication. This means that this product configuration requires this asset. However, **IntegerValue** does not appear on the left side of the implication, since it is not being provided. So, we cannot prove this proposition. Assets cannot be safely composed for this product, since there is a non-satisfied dependency. Therefore, this is an ill-formed product.

These examples illustrate how we can verify safe composition for single products using our approach. To check the entire SPL, we use the FM, since it represents the set of product configurations which are of interest to customers. Rules for translating an FM into propositions have been discussed previously [20]. Every feature relationship (root, optional, mandatory, alternative) has a specific translation to propositional logic, representing its semantics So, we represent the EPL FM in Figure 1 with the following proposition:

$$semanticsFM : EPL \wedge (EPL \Leftrightarrow Operations) \wedge (EPL \Leftrightarrow ValueType) \wedge (Operations \Leftrightarrow (Add \vee Sub)) \wedge (ValueType \Leftrightarrow (Integer \vee Double)) \wedge (ValueType \Leftrightarrow \neg(Integer \wedge Double)).$$

To perform safe composition verification for a CK-based SPL, we relate these two propositions — *semanticsFM* and *constraintsCK*. Domain constraints (FM) must satisfy implementation constraints (CK), as proposed in previous works [8], [10], [20]. The proposition we then need to check is the following:

$$semanticsFM \Rightarrow constraintsCK.$$

If this proposition evaluates to true, we consider the SPL well-formed, since all product configurations respect the provided and required interfaces. If it does not, we know that there are safe composition problems in the SPL. The next section describes how we formalize this intuition in Alloy, using the Alloy Analyzer tool support to detect and report which products are ill-formed.

B. Formalization

Based on the encoding presented in the previous section, there are different ways that we can actually perform the verification using SAT solvers. We use Alloy and the Alloy Analyzer [16] due to its tool support, which can perform automatic analysis over specifications. It also has a formal semantics, and we can use it with different off-the-shelf SAT solvers. We also had previous experience using it in the context of FMs and SPLs, performing analysis over thousands of features [21], [22]. In this section we illustrate how we encode FMs and CK in Alloy. It is important to highlight that translation of such models to Alloy is automatically performed.

An Alloy specification contains a number of signature paragraphs. Each signature denotes a set of objects (similar to a UML class), which we can associate to other objects by relations declared in the signature body. So, a signature paragraph introduces a new type. Alloy provides a restricted set of primitive signatures, so it does not have boolean types natively. However, we can use an idiom for expressing booleans. We

define an abstract signature **Bool**, and two signatures extending it, **True** and **False**. The **extends** keyword means that these are subtypes of **Bool**. The **one** keyword denotes that there is always exactly one instance of a signature. Therefore, every **Bool** is either **True** or **False**. We define features and assets by defining subset signatures (see the **in** keyword). This means that they are subsets of **Bool**. They can assume **True** or **False** values. We do it in this way since features and assets can be selected (**True**) or not (**False**), depending on the product configuration. The following fragment illustrates signature declarations for the boolean idiom we use and EPL features and assets.

```
abstract sig Bool {}
one sig True, False extends Bool {}
one sig EPL, Operations, Add, Sub, ···Double in Bool {}
one sig Program, Expression, Value ···in Bool {}
```

In Alloy, we use predicates (**pred**) to package reusable formulae. If all constraints listed in the body are satisfied, the predicate evaluates to true. Otherwise it evaluates to false. We use predicates to represent the FM and CK as propositions, encoded using the rules discussed in the previous section. The fragment below specifies the FM and CK for EPL. The predicates for specifying FM relationships, such as root and optional, have been previously specified, since we use a theory for encoding FMs in Alloy [21]. For example, we declare the alternative relationship using two arguments: the parent feature, and the set of alternative features (declared using the + keyword to denote the set union operator). We specify cross-tree constraints likewise. We specify the provided and required predicates according to Figure 4, as detailed in Section IV-A[3].

```
pred semanticsFM[] {
  root[EPL]
  mandatory[ValueType,EPL]
  mandatory[Operations,EPL]
  alternative[ValueType, Integer+ Double]
  orGroup[Operations,Add+ Sub]
}
pred constraintsCK[] {
  provided[] ⇒ required[]
}
```

To determine whether an SPL is well-formed, we need to check if all products are well-formed. We do this relating the FM and the CK, verifying that all product configurations satisfy the implementation constraints. To check this in Alloy, we use assertions. Assertions are questions about the model that we wish to verify that are true up to a given scope. We verify assertions using the **check** command, that searches for counterexamples of an assertion. We must specify a scope, that is the maximum size of the top-level signatures that Alloy will use to search within. In our case, the top-level signature we have is (**Bool**). Although there are many features and assets, they are all subset signatures of **Bool**, so they can be either **True** or **False**. Therefore, the scope we use for checking our assertion is 2. The following fragment illustrates how we use Alloy to specify the assertion and its verification.

```
assert verifySPL {
  semanticsFM[] ⇒ constraintsCK[]
}
check verifySPL for 2
```

There are two possible results for a **check** command. If it can not find counterexamples, the assertion is said to be true for the scope defined. If it finds a counterexample, it is presented to the user, and we can analyze it to understand the problem. In Section II we discuss that EPL has ill-formed products. So, when checking the assertion **verifySPL** for EPL, we find a counterexample. Figure 5 illustrates a possible counterexample returned by the Alloy Analyzer. It is an assignment of features and assets to **True** and **False** that violates the assertion. If we examine the feature names (highlighted in bold), we notice that this counterexample is related to the product configuration

{**EPL, Value Type, Double, Operations, Add**}.

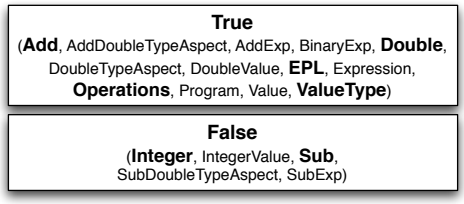


Fig. 5. Counterexample found when checking safe composition of the EPL.

So, from counterexamples, we are able to report the exact product configurations that are not safely composed for the SPL. Since in our context, the scope is well delimited — it does not need to be bigger than 2 — the Alloy Analyzer works as a theorem prover. The result of this check is a proof. If a counterexample is found, the assertion does not hold, meaning that the SPL is ill-formed. If it can not find counterexamples, we consider the SPL well-formed. We have built a tool in Java, using the Alloy API, that automates this verification process. Moreover, we have integrated it into Hephaestus, a tool suite used for managing SPL variabilities [23]. It automatically translates the FM and CK formats used in Hephaestus to Alloy specifications, and verifies safe composition of the SPL. If it is ill-formed, the tool reports the list of ill-formed products.

V. Evaluation

This section presents an evaluation of our approach using the code assets for seven releases of the MobileMedia SPL [15]. We investigate whether safe composition problems happen in MobileMedia releases. If they do, we are also interested whether our approach detects them in reasonable time. Next sections present an overview of MobileMedia and the results of applying the verification for each release. We also discuss issues related to this evaluation, and provide an outline

[3]Hereby omitted for brevity.

of how other SPLs can use our approach for safe composition verification.

A. MobileMedia Overview

MobileMedia is a SPL that manipulates photo, music, and video on mobile devices [15], which has its releases publicly available[4]. For each release, there are Java and AspectJ implementations, both using the Model-View-Controller (MVC) pattern. The Java versions handle variability through feature expressions associated to conditional compilation tags, in a fine-grained level. The AspectJ versions handle variability at a coarse-grained level, using features associated to classes and aspects. Using advices and inter-type declarations, aspects intercept different parts of the program, scattered throughout MVC roles. For example, implementation of the **Sorting** feature consists of modularizing in an aspect all code related to its functionality. We use the AspectJ versions of the first seven MobileMedia releases, due to the similarity of their build files with our present CK notion, which relates feature expressions to assets [4], [5].

Each release consists in the inclusion of new features and reorganization of others in the FM, besides the implementation. Figure 6 illustrates the FM for Release 7. Labels below some features correspond to the release that introduces this feature. For example, release 7 introduces **Capture Photo**.

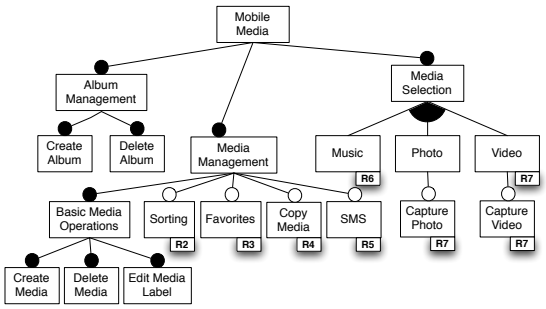


Fig. 6. FM for MobileMedia [15] release 7.

To generate a MobileMedia product, we use *.lst* build files, containing a list of paths for source files that the build task uses to compile. For example, consider part of `MobileMediaWithoutSorting.lst` in Figure 7 (a), and part of `MobileMediaWithSorting.lst` in Figure 7 (b). Both files are based on the product configurations from Release 2, with and without the **Sorting** feature. These *.lst* files contain annotations, starting with `#`, that relate a feature expression with the set of assets that realize it, as we see in the end of the `MobileMediaWithSorting.lst` file. We notice that, in order to include functionality for the **Sorting** feature in a product, we include the **SortingAspect** in the build task.

So, build files describe specific SPL products. Annotations that associate feature expressions to assets make these build files a sort of product-specific CK. For each release, we derive

[4]http://mobilemedia.cvs.sourceforge.net/viewvc/mobilemedia/

```
# MobileMedia
lancs/mobilemedia/core/ui/MainUIMidlet.java
lancs/mobilemedia/core/ui/controller/BaseController.java
lancs/mobilemedia/core/ui/screens/PhotoListScreen.java
...
lancs/mobilemedia/core/util/ImageUtil.java
```

(a) MobileMediaWithoutSorting.lst

```
# MobileMedia
lancs/mobilemedia/core/ui/MainUIMidlet.java
lancs/mobilemedia/core/ui/controller/BaseController.java
lancs/mobilemedia/core/ui/screens/PhotoListScreen.java
...
lancs/mobilemedia/core/util/ImageUtil.java

# Sorting
lancs/mobilemedia/optional/sorting/SortingAspect.aj
```

(b) MobileMediaWithSorting.lst

Fig. 7. Build files for products from Release 2 of the MobileMedia SPL [15].

the CK for the SPL based on the available build files. For releases 1 through 5, since there are not much variations, there is a single build file (*.lst*) for each release, with a complete mapping of feature expressions to assets for the entire SPL. So, they already are a CK representation for those releases. For releases 6 and 7, there are build files describing only a subset of product configurations. Therefore, we took special care when extracting the CK from these files, using *diff* tools to compare the common and variable parts between them. One thing that helped us to make sure that this task was not introducing errors is that the name of some aspects also references the feature expressions associated. For example, **PhotoAndMusicAndSMS** aspect implements a certain behaviour associated with the feature expression **Photo** $\wedge$ **Music** $\wedge$ **SMS**. Another thing worth mentioning is that for both sixth and seventh releases, there is a build file describing a product configuration with all features selected, which contains the majority of associations of feature expressions to assets, needed to extract the CK.

To enable the safe composition verification, we also need to have provided and required interfaces for CK items. In this evaluation, we considered required interfaces as syntactic dependencies between assets and provided interfaces as the asset names (classes and aspects). For this evaluation, we capture dependencies specific to the MobileMedia artifacts. We only focus on classes and aspects under the *lancs.mobilemedia* package, because these are the ones that the build files associate with feature expressions. Thus, we do not capture Java API and external libraries dependencies, for example. While capturing these would make the provided and required sets richer, the problems we found and discuss in this section would still happen. Nonetheless, since we are only using a subset of all actual dependencies, it is possible that there exist more errors than those we found. So, given that we know all classes and aspects names under the *lancs.mobilemedia* package, the provided and required interfaces are automatically extracted from the code using Soot[5]. While we need this information to

[5]Soot is a Java optimization framework that allows analyzing Java bytecode. http://www.sable.mcgill.ca/soot/

enable the use of our verification approach, it is important to remind that we only need to perform this step because MobileMedia does not have a CK with the interfaces already specified.

With this information, we are able to perform safe composition verification for MobileMedia releases. The process of generating an Alloy specification is *automated*, following the codification rules already presented. For each release, we check if, for all product configurations, the required interfaces are being provided. Next sections detail results found when analyzing the first seven releases of the MobileMedia SPL.

B. Releases 1-5

The first release is not considered a SPL, since it is a single product. Therefore, we just need to make sure that it is also a valid AspectJ program, which is the case. When we perform the verification for releases 2 to 5, the tool does not find any problems. These releases are smaller and simpler, ranging from 1 product, in release 1, to 16, in release 5. All releases deal only with photos. **Music** and **Video** are only introduced in later releases. So, changes do not have a great impact on existing assets, they mainly consist of adding new assets.

C. Release 6

The release 6 introduces the ability to handle music. Both the FM and CK were reorganized to support this change. New assets are added, and existing ones are modified. This release has 3 KLOC. The FM for this release allows 48 product configurations, which now can handle either photos and music, and both of them as well (see Figure 6). However, there are only six build files available for this release. We use them to extract the CK.

Verification for this release detects 8 ill-formed products. The six pre-existent build files do not describe these products. The MobileMedia developers have not generated or tested them. Analysis of the ill-formed products revealed that a single problem makes them ill-formed. Figure 8 illustrates that the **SMSAspect**, which is associated with the **SMS** feature, requires assets provided by the **Photo** feature. So, in the case of product configurations where we select **SMS** and **Music**, but not **Photo**, there are missing dependencies, which cause them to be ill-formed. The **ImageMediaAccessor** and **PhotoViewScreen** assets do not show up in other parts of the CK. Filenames below the CK in Figure 8 represent the existing build files where we find these association of features with assets.

Release 5 introduces the **SMS** feature, providing functionality for sending messages with photos. Since the **SMS** implementation only send photos, but not music, which was a functionality introduced in this release, there is an inconsistency of the implementation with the FM for this release, that allows product configurations to select **SMS** without **Photo**. There is no constraint stating that **SMS** requires **Photo**. Then, a possible solution to this problem would be to modify the implementation to conform to the FM, so **SMS** can work with both photos and music. Another solution would be to change

Feature Expression	Provided	Required
...	...	...
SMS	SMSAspect	..., **ImageMediaAccessor**, **PhotoViewScreen**, ...
Photo	ImageMediaAccessor, PhotoViewScreen, ...	MainUIMidlet, AbstractController, ...
...	...	...

MobileMedia01.lst, MobileMedia02a.lst, MobileMedia04.lst

Fig. 8. Part of the CK for Release 6 of MobileMedia [15], illustrating the SMS problem.

the FM, including the constraint stating that **SMS** requires **Photo**

D. Release 7

This release introduces another media type: **Video**. It also adds the ability to capture photos and videos. The MobileMedia SPL size grows to 4 KLOC. The number of product configurations increases to 272. However, similarly as in release 6, there are build files for only a subset of product configurations, describing 10 specific products. Thus, we extract the CK from the available build files. When performing the verification for this release, we find 116 ill-formed products. Again, the 10 pre-existent build files do not describe these products. Besides the SMS problem, which the previous subsection already discusses, there are two new problems.

Figure 9 illustrates the **Capture Photo** problem. This recently introduced feature provides the **CapturePhotoAspect**. This aspect depends on the **PhotoViewController** class, provided only when we select **Copy** or **SMS** features. Therefore, a solution for this problem is to change, in the CK, the feature expression that provides **PhotoViewController** to **Copy** $\vee$ **SMS** $\vee$ **CapturePhoto**.

Feature Expression	Provided	Required
...	...	...
Capture Photo	CapturePhotoAspect	..., **PhotoViewController**, ...
Copy $\vee$ **SMS**	PhotoViewController	MainUIMidlet, AbstractController, ...
...	...	...

MobileMediaA03.lst, MobileMediaABC03.lst

Fig. 9. Part of the CK for Release 7 of MobileMedia [15], illustrating the Capture Photo problem.

Another problem happens with the aspect **PhotoAndMusicAndVideo**, included when all media types are present in a product configuration. Figure 10 illustrates this problem presenting part of the CK for this release. This aspect depends on another aspect, **OptionalFeatureAspect**, associated to the selection of all optional features.

This is due to a precedence declaration, used to organize aspects application implementing MobileMedia variations. So, every time that some combination of aspects is present on a product, these precedence declarations organize their application. In some releases, there are actually aspects that only have precedence declaration in its body. This is the case of

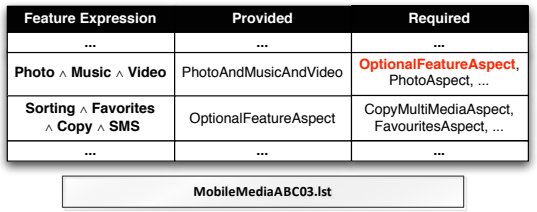


Fig. 10. Part of the CK for Release 7 of MobileMedia [15], illustrating the Photo ∧ Music ∧ Video problem.

OptionalFeatureAspect, for example. Listing 1 illustrates the code used in this aspect, associated with the feature expression **Sorting** ∧ **Favorites** ∧ **Copy** ∧ **SMS**. In other words, it is included in a product when we jointly select all optional features for this release.

Listing 1. Code for OptionalFeatureAspect.aj.

```
package lancs.mobilemedia.optional;
import lancs.mm...CopyMultiMediaAspect;
import lancs.mm...FavouritesAspect;
import lancs.mm...PersisteFavoritesAspect;
import lancs.mm...SortingAspect;
import lancs.mm...SMSAspect;
public aspect VideoAndOptionalFeatures {
  declare precedence : CopyMultiMediaAspect,
   CopyAndVideo, FavouritesAspect,
   SortingAspect, PersisteFavoritesAspect;
}
```

In this release, the only existing build file describing a product containing all media types also contains all optional features. This seems to be the reason for the inclusion of **OptionalFeatureAspect** in the precedence declaration of **PhotoAndMusicAndVideo**. So, when compiling this product, there are no problems at all. However, if we consider other SPL products with feature combinations not previously tested, problems might happen, as our approach detected. A possible solution aligned with the MobileMedia implementation is to refactor the code, thus removing this precedence declaration from this aspect. We would have to create variations of this aspect to consider all possible different combinations of **Photo** ∧ **Music** ∧ **Video** with the **Sorting**, **Favorites**, **Copy** and **SMS** features. After restructuring the code, we would also need to update the CK with these new assets.

E. Discussion

A possible alternative approach for the verification would be a brute-force one, as discussed in Section II. The average time for compiling a single product configuration of MobileMedia is around 50 seconds. For release 7, it would take almost 4 hours for generating and compiling all products, besides time for testing them. On the other hand, time for verifying release 7 and reporting the list of 116 ill-formed products takes less than 34 seconds, which is a reasonable time, since our main focus was not efficiency. This reinforces the benefits of using an automated approach for verifying safe composition [8], [10], [13], [12] instead of generating all SPL products. The larger the SPL, the bigger the benefits. Besides that, in order to compile all products, we would have to manually create a build file for each product. Moreover, the majority of this time is due to the steps needed to retrieve the list of ill-formed products. Actually, it only takes a second, in average, to only check if the SPL is well-formed, without reporting the ill-formed products. Time for extracting the CK in releases 6 and 7 was between 30 and 40 minutes, by careful analysis and comparison of the existing build files. Table I summarizes the results of the verification for each of the MobileMedia releases evaluated, with the CK already extracted.

TABLE I
SUMMARY FOR VERIFICATION OF MOBILEMEDIA RELEASES.

Release	# Products	ill-formed Products	Time (s)
2	2	0	0.751
3	4	0	0.947
4	8	0	1.092
5	16	0	1.217
6	48	8	4.123
7	272	116	33.813

It can be argued that the CK derivation process introduced problems. However, for all releases, the build files document the associations of feature expressions to assets, needed to extract the CK. Moreover, we found problems related to associations of feature expressions to assets documented in existing build files, as illustrated in Figures 8, 9 and 10.

These problems might have happened and went unnoticed due to the manner in which MobileMedia was implemented and tested. The implementation seems to be guided by a subset of product configurations, instead of envisioning the whole SPL scope. Therefore, developers made sure that the products described by the existing build files were correct, but did not check, even using brute-force, if all products were correct.

Also, since in this evaluation we are extracting our interfaces in a syntactic way, we are only capturing syntactic problems. This is a particular weakness of the way we extracted these interfaces, not of our verification approach. Had we used a more powerful mechanism for deducing the interfaces, or complemented them manually, our approach could find even more errors in MobileMedia, not only syntactic problems. For example, to ensure that the implementation of an interface is provided.

Also, it is worth mentioning that the interface information does not need to reside in the CK. So, we do not need to change existing models, specially if we can compute most of these interfaces in an automated way. For the sake of simplicity and clarity, the examples used throughout the text have the interfaces as part of the model. However, this approach is not dependent on such a modification of the CK model. It only depends on the existence of provided and required interfaces, that we can relate to feature expressions.

Finally, we can generalize the steps used for verifying safe composition of CK-based SPLs. Although we have only discussed our approach in the context of verifying releases of MobileMedia, we can use it in other SPLs, since we already automated a good part of it. For example, we could generalize

them with the following steps:

1) If not existent, the developer should define a CK associating feature expressions to assets used in the SPL. Also define what are provided and required interfaces in this context.
2) The developers should also define the algorithm or process for extracting provided and required interfaces from SPL assets. A good part of this extraction can be automatically performed [10], [13], [12], [8].
3) Translate the CK into Alloy. This is already automated by our tool.
4) Perform the verification following the process discussed in Section IV, which is also automated.
5) Finally, if there are ill-formed products, the developer must debug the CK using counterexamples and provided/required sets to detect what is causing the problem.

VI. Related Work

Thaker et al. presented techniques for verifying type safety properties of AHEAD [9] product lines using FMs and SAT solvers [10]. They extract properties from feature modules and verify that they hold for all SPL members. These properties are based on the AHEAD theory of program synthesis, and some of them do not reveal actual errors, but rather designs that *smell bad*. Delaware et al. formalized this work using an object-oriented kernel language extended with features, called Lightweight Feature Java (LFJ) [13], based on Lightweight Java. They prove soundness of the underlying type system. Their focus is on inferring type checking constraints from the language used for implementation, while we have not focused in extracting interfaces, but on verifying interfaces provided to the CK. Thus, the quality of our verification is dependent on such information. While we do not use a theorem prover to prove soundness of our formalization, as Delaware et al. do, our Alloy encoding provides sound and complete analysis, due to our scope being well-delimited.

Apel et al. proposed Feature Featherweight Java (FFJ) as a type system for feature oriented programming (FOP) [11]. They use this type system to check whether a given composition of features is safe, before compilation. They later presented a condensed version of FFJ, proving soundness and completeness, and also presenting an implementation of the language (FFJ$_{PL}$) [12]. Similar to our approach, they perform analysis using SAT solvers, checking if SPL implementation is well-typed. The type checker provides detailed error messages, while our approach only provides the list of products in which problems occur. Apel et al. also proposed the gDeep core calculus for uniform feature composition [24]. It aims to be a language-independent pluggable type system, that can be used with different kinds of artifacts. In this way, it is similar to our work, since our CK model is not language-specific as well, and we can extract interfaces from different kinds of artifacts. However, although we can interpret our analysis as a proof, we do not provide a full formalization of our CK model. Finally, Apel et al. also proposed a reference checking algorithm for feature-oriented SPLs [25]. Similarly as we do with our CK model, they extend feature structure trees (FSTs) with references. Two variations of the algorithm are presented: global and local. The global version, in a similar way as we do, generates a single propositional formula that contains all references. The local version creates a propositional formula for each reference, targeting efficiency, since it results in a numer of smaller formulas that can be cached and reused [12]. They evaluate two small product lines written in different languages (Java and C). As a future work, we intend to compare both approaches with respect to expressiveness.

Schaefer et al. proposed a compositional type system for Delta-oriented Programming (DOP) [26] SPLs in Java using a core calculus. DOP uses delta modules to implement features, that extend FOP with the ability to remove classes, methods and fields. Similarly as with our CK, a delta module can be associated with feature expressions. Constraints are inferred for each delta module, and differently from our work, delta modules are checked in isolation, instead of generating a single proposition. The type system is proven correct and complete with respect to the core calculus used. A difference from our work is that due to the compositional type-checking, if new modules are added, we only need to type-check the newly added modules, while using our approach we would have to check the entire SPL again.

Czarnecki and Pietroszek proposed a well-formedness verification approach for feature-based model templates [8]. A feature-based model template consists of an FM and an annotated model expressed in some general modeling language such as UML or a domain-specific modeling language. Annotations refer to features and resemble the use of conditional compilation directives. Given a product configuration, a template processor creates a template instance by evaluating presence conditions and removing elements when such conditions evaluate to false. The idea is to prevent ill-formed template instances. In a similar way, they check FMs against constraints to verify that no ill-formed template instances can be produced. They also retrieve ill-formed products from error messages returned by the SAT solver counterexamples. These annotations are equivalent to the feature expressions in our CK. A difference is that while our CK information is modularized, in model templates it is scattered throughout the SPL assets.

Also using annotations, Kaestner et al. proposed the Color Featherweight Java (CFJ) calculus [14]. This formalization was motivated by the Colored IDE tool (CIDE) [7], for refactoring legacy systems into SPLs. The mapping between features and code occurs through a disciplined form of conditional compilation. This calculus established type rules to ensure that only well-typed programs can be generated. They formally prove that given a well-typed CFJ SPL, all possible variants are well-typed FJ programs, i.e., generation preserves typing. Code annotated with two features has the same semantics of an *and* ($\wedge$) operator. However, other types of feature expressions are not supported, and the CK is scattered throughout the code. In the case of SPLs with multiple variability mechanisms, dealing with both coarse and fine-grained variabilities, we could combine our approaches to address both levels of

variability. This could help in contexts such as in the **Photo** ∧ **Music** ∧ **Video** problem, discussed in Section V-D.

Mendonca et al. show that FMs pose no significant difficulty for SAT solvers [27], justifying widespread use of SAT-based systems, such as the Alloy Analyzer. We confirmed their findings, performing analysis on FMs with the same encoding used in this work, up to 10,000 features, in a few seconds [22]. In this work, we add the CK to the encoding. However, the constraints on the CK formulae are similar to those in FMs. Therefore, they fit in the constraint class that Mendonca et al. show that is easy to analyze.

VII. Conclusions

In this work, we discuss safe composition of CK-based SPLs. We extend an existing CK notion [4], [5], taking into account required and provided interfaces expressing dependencies between assets. Using these interfaces, we defined an approach for safe composition verification of CK-based SPLs. We have also developed a tool that automatically translates FM and CK to Alloy, and checks whether an SPL is well-formed. If it is not, it reports the ill-formed products. Since we have a well-defined scope in this context, we use the Alloy Analyzer as a theorem prover. That is, the analysis is sound and complete.

We evaluate the approach using seven releases of the MobileMedia SPL [15]. For releases 6 and 7, the tool identifies safe composition problems related to non-conformity with the FM, bad CK specification, and implementation not envisioning the full SPL scope. It found errors that, in release 7, make 116 products invalid, out of the 272 possible. Analysis for this release took almost 34 seconds, which is less than the average time needed for compiling a single product.

As future work, we intend to evaluate our approach with other SPLs from different domains, exploring metrics for the evaluation. We also intend to investigate the use of our approach in more complex SPL settings, with multiple programming languages and with CK models where transformations go beyond asset selection. We also plan to integrate our approach with a tool that implements a general theory for SPL refinement [6] for checking refactorings [5]. In this case we would check whether changes made to an SPL preserve well-formedness. Finally, we intend to explore ways where we can optimize the verification.

Acknowledgment

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[6]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. We also acknowledge current financial support from CNPq, grants 477336/2009-4 and 304470/2010-4. The first author is supported by the grant MCT/CNPq/P-GAEST 70/2009.

[6]http://www.ines.org.br

References

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," tech. rep., Carnegie-Mellon University SEI, November 1990.
[3] K. Czarnecki and U. Eisenecker, *Generative programming: methods, tools, and applications*. Addison-Wesley, 2000.
[4] R. Bonifácio and P. Borba, "Modeling scenario variability as crosscutting mechanisms," in *AOSD '09*, pp. 125–136, 2009.
[5] P. Borba, "An introduction to software product line refactoring," in *GTTSE'09*, pp. 1–26, Springer-Verlag, 2011.
[6] P. Borba, L. Teixeira, and R. Gheyi, "A theory of software product line refinement," in *ICTAC'10*, pp. 15–43, Springer-Verlag, 2010.
[7] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE 2008*, pp. 311–320, 2008.
[8] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness ocl constraints," in *GPCE 2006*, pp. 211–220, 2006.
[9] D. Batory, "Feature-oriented programming and the AHEAD tool suite," in *ICSE '04*, pp. 702–703, 2004.
[10] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe composition of product lines," in *GPCE 2007*, pp. 95–104, 2007.
[11] S. Apel, C. Kästner, and C. Lengauer, "Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement," in *GPCE 2008*, pp. 101–112, 2008.
[12] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer, "Type safety for feature-oriented product lines," *Automated Software Engg.*, vol. 17, pp. 251–300, September 2010.
[13] B. Delaware, W. Cook, and D. Batory, "Fitting the pieces together: a machine-checked model of safe composition," in *ESEC/FSE 2009*, pp. 243–252, 2009.
[14] C. Kästner and S. Apel, "Type-checking software product lines - a formal approach," in *ASE '08*, pp. 258–267, 2008.
[15] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability," in *ICSE 2008*, pp. 261–270, 2008.
[16] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Mass.: The MIT Press, 2006.
[17] R. E. Lopez-Herrejon and D. Batory, "The expression problem as a product line and its implementation in AHEAD," Tech. Rep. CS-TR-04-52, The University of Texas at Austin, Jan. 3 2005.
[18] C. W. Krueger, "Easing the transition to software mass customization," in *PFE '01*, pp. 282–293, Springer-Verlag, 2002.
[19] D. Batory, D. Benavides, and A. Ruiz-Cortes, "Automated analysis of feature models: challenges ahead," *Commun. ACM*, vol. 49, no. 12, pp. 45–47, 2006.
[20] D. Batory, "Feature models, grammars, and propositional formulas," in *SPLC 2005*, vol. 3714 of *LNCS*, pp. 7–20, 2005.
[21] R. Gheyi, T. Massoni, and P. Borba, "A theory for feature models in Alloy," in *First Alloy Workshop*, pp. 71–80, 2006.
[22] R. Gheyi, T. Massoni, and P. Borba, "Automatically checking feature model refactorings," *Journal of Universal Computer Science*, vol. 17, no. 5, pp. 684–711, 2011.
[23] R. Bonifácio, L. Teixeira, and P. Borba, "Hephaestus: A Tool for Managing Product Line Variabilities," in *3rd Brazilian Symposium on Software Components, Architectures, and Reuse - Tools Session*, pp. 26–34, 2009.
[24] S. Apel and D. Hutchins, "A calculus for uniform feature composition," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 19:1–19:33, May 2008.
[25] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Language-independent reference checking in software product lines," in *FOSD '10*, pp. 65–71, ACM, 2010.
[26] I. Schaefer, L. Bettini, and F. Damiani, "Compositional type-checking for delta-oriented programming," in *AOSD '11*, pp. 43–56, ACM, 2011.
[27] M. Mendonca, A. Wasowski, and K. Czarnecki, "Sat-based analysis of feature models is easy," in *SPLC '09*, pp. 231–240, Carnegie Mellon University, 2009.