The Journal of Systems and Software 86 (2013) 1038–1053

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Safe composition of configuration knowledge-based software product lines

Leopoldo Teixeira[a,*], Paulo Borba[a], Rohit Gheyi[b]

[a] Informatics Center, Federal University of Pernambuco, Brazil
[b] Department of Computing Systems, Federal University of Campina Grande, Brazil

ARTICLE INFO

Article history:
Received 21 January 2012
Received in revised form 12 October 2012
Accepted 3 November 2012
Available online 12 December 2012

Keywords:
Software Product Lines
Configuration Knowledge
Safe composition

ABSTRACT

Mistakes made when implementing or specifying the models of a Software Product Line (SPL) can result in ill-formed products — the safe composition problem. Such problem can hinder productivity and it might be hard to detect, since SPLs can have thousands of products. In this article, we propose a language independent approach for verifying safe composition of SPLs with dedicated Configuration Knowledge models. We translate feature model and Configuration Knowledge into propositional logic and use the Alloy Analyzer to perform the verification. To provide evidence for the generality of our approach, we instantiate this approach in different compositional settings. We deal with different kinds of assets such as use case scenarios and Eclipse RCP components. We analyze both the code and the requirements for a larger scale SPL, finding problems that affect thousands of products in minutes. Moreover, our evaluation suggests that the analysis time grows linearly with respect to the number of products in the analyzed SPLs.

1. Introduction

A Software Product Line (SPL) is defined as a set of software systems generated from reusable assets, that share common characteristics but are sufficiently distinct from each other (Pohl et al., 2005). To accomplish this, some SPL approaches use Feature Models (FMs) (Kang et al., 1990), which describe domains through commonalities and variabilities; and Configuration Knowledge (CK) (Czarnecki and Eisenecker, 2000), which relates features to assets, driving product generation. We can make such knowledge explicit in a model (Bonifácio and Borba, 2009; Borba, 2009; Borba et al., 2012), or through annotations over SPL assets (Kästner et al., 2008; Czarnecki and Pietroszek, 2006). It can also be made implicit by the implementation mechanism, such as feature modules or containment hierarchies (Batory, 2004; Thaker et al., 2007; Apel et al., 2008, 2010).

In this context, *safe composition* is a property, or technique to check such property, that guarantees that all SPL products are safe accordingly to a particular property (Czarnecki and Pietroszek, 2006; Thaker et al., 2007; Delaware et al., 2009; Kästner and Apel, 2008). In this article, we use the term to denote checking the absence of unresolved dependencies. For example, it is undesirable that a class Account is absent in a product with class Bank that references it. We present a formal definition in Section 2. Some approaches avoid the need of synthesizing the entire SPL to verify safe composition (Thaker et al., 2007; Apel et al., 2008, 2010; Delaware et al., 2009; Czarnecki and Pietroszek, 2006). They solve it in the context of SPLs structured using a single language, through implicit CK models and annotations over assets.

Here we focus on a dedicated CK model, which modularizes the configuration information, relating features to assets (Bonifácio and Borba, 2009; Borba, 2009), and enables language independence. Assets might be classes, aspects, use cases, property files, and so on. CK evaluation against a selected feature configuration yields the assets needed to build a product. Mistakes made when specifying the assets or this model might result in safe composition problems. For example, incompatible entries in the CK lead to incompatible assets in the resulting product, such as two different implementations of the same class.

To verify safe composition for SPLs with such CK models, which we call CK-based SPLs, we propose an automated analysis that performs the verification in a language-independent way, using interfaces expressing dependencies between assets (Teixeira et al., 2011). These interfaces can be calculated using standard or tailored analysis. Given that such interfaces are available, we translate them to the Alloy formal specification language (Jackson, 2006). To actually perform the verification we use the Alloy Analyzer (Jackson et al., 2000) tool.

In particular, we extend our previous work (Teixeira et al., 2011) in a number of ways. First, to give further evidence of the generality of our approach, we instantiate our approach in other compositional settings, going beyond Java and AspectJ. We now deal with different kinds of assets such as use case scenarios

* Corresponding author.
E-mail addresses: lmt@cin.ufpe.br (L. Teixeira), phmb@cin.ufpe.br (P. Borba), rohit@dsc.ufcg.edu.br (R. Gheyi).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.11.006

and Eclipse RCP components (McAffer and Lemieux, 2005), which require implementing new interface extractors and involve new kinds of dependencies that occur between different languages, such as XML and Java, for example. We also go beyond Mobile Media (Figueiredo et al., 2008), an SPL which handles different media types in mobile devices, using a non academic SPL. This helps us to show that safe composition problems happen in larger scale industrial case studies, and that our approach is able to detect them. We do that analyzing both the code and the requirements for Release 6 of TaRGeT (Ferreira et al., 2010), an SPL of automatic test generation tools, with 31 implemented features and approximately 32KLOC. In the code analysis, we detected that a single problem, due to a renaming refactoring, causes thousands of products to be ill-formed. This confirms our previous findings using different MobileMedia releases, where few problems have an impact over a good portion of products in the SPL. In TaRGeT, requirements are structured using the Modeling Scenario Variability as Crosscutting Mechanisms (MSVCM) (Bonifácio and Borba, 2009) approach, which enables generating product-specific use case scenarios. Our analysis found a number of problems that caused the majority of the product-specific scenarios to be ill-formed. Finally, to handle the scalability issues that come with larger case studies, we have changed our strategy for reporting ill-formed products. After we detect a safe composition problem by analyzing a model of the entire SPL, we generate and analyze the models of all products to identify the ill-formed ones. This way, we change our analysis from family-based to family-product-based (Thuem et al., 2012). This change enabled us to analyze thousands of products in minutes, whereas we needed hours using our previous approach. Our evaluation now suggests that the analysis time grows linearly with respect to the number of products in the analyzed SPLs. We also observe that the analysis time depends on the kinds of assets used for implementing the SPL. By providing a new implementation of the approach of the reporting ill-formed products phase of our approach, this work might be also useful to others interested in analyzing larger scale SPLs.

This text is organized as follows. In Section 2, we show how the safe composition problem can happen with different kinds of assets, such as code and use case scenarios. Section 3 presents our approach for verifying CK-based SPLs using interfaces. Following that, in Section 4 we discuss the evaluation of our approach using MobileMedia and TaRGeT code assets and TaRGeT use case scenarios. We discuss related work in Section 5 and conclude with Section 6.

2. Motivating examples

In this section, we show how the safe composition problem might occur in both code and requirements assets. First, we introduce a formal definition for safe composition. As discussed, safe composition is a property to verify that all SPL products are safe according to a particular property. In this article, for a product p, we use $safe(p)$ to denote that p is absent of unresolved dependencies. For example, in an object-oriented context this means that the product does not include references to a class X which is absent in the set of classes for the product. We could provide other meanings such as type-safety, for example.

Definition 1 *(Safe Composition).* For software product line PL, we say that PL satisfies the safe composition property when

$$\forall\, product \in PL \cdot safe(product)$$

□

To illustrate how safe composition problems can happen in practice, consider MobileMedia, an SPL that manipulates media on mobile devices (Figueiredo et al., 2008).[1] It is implemented with classes and aspects. For example, implementation of the **Sorting** feature consists of modularizing in an aspect the code related to its functionality. Each release adds new features and their corresponding implementation. Fig. 1 illustrates how the FM evolved until Release 7, using labels below features to show the release that introduces a feature. For example, release 7 introduces **Capture Photo**.

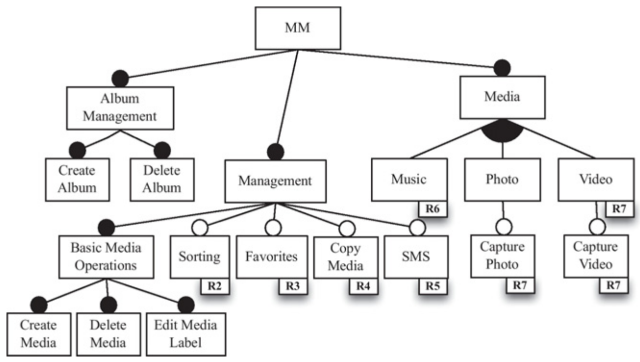


Fig. 1. FM for MobileMedia (Figueiredo et al., 2008) release 7.

We denote mandatory features, such as **Management**, using filled circles. The **SMS** feature is optional, as denoted by the blank circle, so it might not be selected in some products. Child features of **Media** form an OR group, denoted by a filled arc, which means that we must select at least one. The FM denotes the set of possible product configurations, that is, valid feature selections for an SPL. A product configuration from this FM is as follows: {**Music**, **Photo**, **Sorting**, **SMS**}.[2]

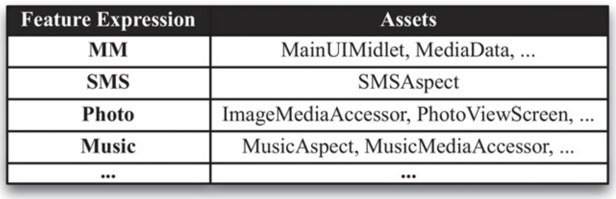
To enable automatic product generation, we use a CK (Czarnecki and Eisenecker, 2000) to relate features to assets. In this article, we use a dedicated model (Bonifácio and Borba, 2009; Borba, 2009; Zschaler et al., 2009). If not properly specified, a CK might cause problems such as unresolved dependencies in the final product (Thaker et al., 2007; Apel et al., 2008; Delaware et al., 2009; Kästner and Apel, 2008). Such problems can happen because of incompatible mappings in the CK, but it could also be the case that, due to negligence, the developer forgets to add an entry to the CK, for example forgetting to relate the X class with the X feature. Therefore, when we generate a product containing the X feature, it will likely be ill-formed, since it was expected that the X class would be present, and some other classes would depend on it, resulting in a unresolved dependency. We show an instance of this problem in the following subsections, where we discuss how the safe composition problem might happen when using CK models to relate features to both code and requirements.

2.1. Code assets

Focusing on code first, Fig. 2 presents part of the MobileMedia CK. In this case, assets are classes and aspects. For instance, we associate the **SMS** feature with the **SMSAspect** in the second row. This aspect modularizes code related to the SMS functionality. Feature expressions act as enabling conditions, and we can use standard propositional logic operators to associate assets with the presence of two features, for example. Evaluating the CK against a product configuration yields a set of assets, used to build the corresponding product.

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

[2] We omit mandatory features for brevity.



Feature Expression	Assets
MM	MainUIMidlet, MediaData, ...
SMS	SMSAspect
Photo	ImageMediaAccessor, PhotoViewScreen, ...
Music	MusicAspect, MusicMediaAccessor, ...
...	...

Fig. 2. Part of the Configuration Knowledge for the code of the MobileMedia.

For example, evaluating the CK from Fig. 2 for the product configuration {**SMS, Music**} yields the following set of assets: {**MainUIMidlet, MediaData, SMSAspect, MusicMediaAccessor, MusicAspect,** ...}. However, **SMSAspect** references both **ImageMediaAccessor** and **PhotoViewScreen** classes, which are only associated with the **Photo** feature. The FM in Fig. 1 allows product configurations where we select the **SMS** feature, but do not select the **Photo** feature. In these cases the corresponding products would have unresolved dependencies, which are undesirable, according to Definition 1. A possible solution to this problem would be to change the FM, including the constraint stating that **SMS** requires **Photo**. This way, no products would have unresolved dependencies. Errors of this kind can happen if we do not have a way to check safe composition for the SPL without synthesizing all products.

2.2. Requirements

A number of approaches for managing variability in requirements documents have been proposed (Eriksson et al., 2005; Czarnecki and Antkiewicz, 2005; Bonifácio and Borba, 2009; Alférez et al., 2009). Here, we focus on the Modeling Scenario Variability as Crosscutting Mechanisms (MSVCM) (Bonifácio and Borba, 2009) approach. MSVCM specifies use case scenarios in a tabular notation, with columns for user actions and system responses. It specifies variant behavior in aspectual use cases, parameters and non mandatory scenarios. In this context CK evaluation yields product-specific use case scenarios, useful, for example, to generate functional tests. For instance, still using MobileMedia as an example, Fig. 3 specifies a common usage behavior of users reproducing content. It consists of a sequence of steps that describe the functional behavior of the intended system.

We can associate scenarios to features. Thus, a particular use case scenario might exist in some products and not in others. We can use the from and to step for defining alternative scenarios. MSVCM also proposes other constructs. For example, in step SC01.1 from Fig. 3, we reference the *mediacontent* parameter. This parameter allows instantiating the scenario for different configurations of the **Media** feature. We could then have instantiated scenarios where *mediacontent* is replaced by ***Video***, ***Audio***, for example. We relate parameters to features in the CK, detailed in Fig. 5. Use case scenario parameterization allows fine grained configuration of scenarios that share the same behavior and differ in relation to the values of a specific concept.

To specify other kinds of variations on scenarios, MSVCM uses aspectual use cases. This makes it possible to express variations in the behavior (represented as a sequence of steps) of an existing scenario. For example, see the @Favorites annotation in step SC01.3. It basically indicates a point that is supposed to be affected by an aspectual use case. In this case, such use case specifies a variation associated to the optional feature Favorites, which enables users to set media as favorite, and also to view the list of favorite media. Fig. 4 presents the aspectual use case that changes the behavior in particular points of the specification. It quantifies over the defined annotation, so, when evaluated, we compose it wherever the annotation happens. In this example, we include the ADV01.1 step after all points where the @Favorites annotation appears. When generating the product-specific use case scenarios, annotations are removed.

While we use the same FM from Fig. 1, the CK for requirements is slightly different from the one in Fig. 2, since we have to compose use case scenarios to generate a product specification. Instead of associating features with assets, we have features associated to three types of transformations: (i) select scenario (includes a given scenario in the final product); (ii) evaluate advice (composes an advice through join points); and (iii) bind parameter (replaces parameterized text by feature data). Fig. 5 shows the CK for our example. Since we associate the scenario SC01 with the root feature,

SC01
Scenario: Reproduce media.
Description: This scenario allows a mobile phone user to start the reproduction of content.
From Step: Start
To Step: End
Flow of events:

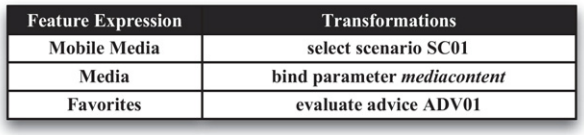
Step	User Action	System Response
SC01.1	User selects an existing folder from options *mediacontent*.	The device populates the list of available content in an album.
SC01.2	User views the content of the album.	-
SC01.3	User selects one file from the list of *mediacontent* files.	Media is displayed on the screen. @Favorites

Fig. 3. MSVCM specification of the common usage scenario for EPL products.

ADV01
Advice: Set as Favorite.
Description: Set media content as favorite.
Type: after
Pointcut: @Favorites
Flow of events:

Step	User Action	System Response
ADV01.1	User sets media as favorite.	System includes media in the favorites list.

Fig. 4. MSVCM specification of the advice for Favorites.



Feature Expression	Transformations
Mobile Media	select scenario SC01
Media	bind parameter *mediacontent*
Favorites	evaluate advice ADV01

Fig. 5. CK used for managing requirements variability in MobileMedia.

it is always included in the resulting specification. The *mediacontent* parameter is bound to the **Media** feature. When we evaluate the CK, it replaces the *mediacontent* parameter with the features selected under **Media**. Finally, we associate the **Favorites** feature with the ADV01 advice. That is, we only compose the advice into the specification when we select the **Favorites** feature.

Evaluating the CK in this context yields product-specific use case scenarios. For example, evaluating the CK from Fig. 5 for the product configuration {**Photo, Music, Favorites**} yields the result we show in Fig. 6. Notice that, in the resulting SC01 scenario, through CK evaluation, the ADV01.1 step is woven into the scenario, since we selected the **Favorites** feature and thus evaluated the ADV01 advice. Also, the *mediacontent* parameter is bound to the children of the **Media** feature, **Photo** and **Music** in this example.

Use case scenarios can be composed using references to steps, parameter names, and pointcuts. As with code, we can have unresolved dependencies if we make a mistake when specifying the CK or the scenarios. For example, recall that in step SC01.1 from Fig. 3 we reference the *mediacontent* parameter. Suppose that in the CK from Fig. 5, due to the feature name we mistakenly refer to *media*, instead of *mediacontent*, when defining the bind parameter in the second row. Then, the parameter would be unbound and the product-specific use case scenario generated would be incorrect, since it would not contain any information about the media content available. By Definition 1, the SPL is unsafe. We show further MSVCM examples in Section 4.2.2.

It is undesirable to generate ill-formed products, but it might happen due to mistakes when designing and implementing the SPL. To ensure productivity and avoid unexpected maintenance costs, it is important to verify safe composition without synthesizing the entire SPL (Thaker et al., 2007; Delaware et al., 2009). Generating all SPL products for verifying is often impractical, since there are SPLs that can generate thousands of products or take too long to build products (Batory et al., 2006). The next section describes our approach to verify safe composition in CK-based SPLs, in a language independent way. We illustrate how we could use it to detect the problem in Section 2.1, but we can also instantiate the approach to detect problems in requirements, as we discuss in Section 4.2.2.

3. Safe composition of CK-based Software Product Lines

In this section we introduce our approach for verifying safe composition of CK-based SPLs. To enable the verification, we use dependencies between assets in the form of provided and required interfaces. Consequently, for each CK item we also have provided and required interfaces. Fig. 7 exemplifies this, illustrating how we can associate feature expressions from the CK in Fig. 2 to provided and required interfaces. In this example, since we deal with code assets, interfaces are asset names, representing dependencies between classes and aspects. For requirements, we use as interfaces scenario names, steps, parameter names, pointcuts and annotations. Although we focus on the code example in this section, the same concepts work for requirements and other kinds of assets, given that we have provided and required interfaces. With such interfaces, we can verify safe composition of an SPL without building all products. Intuitively, we only need to verify, for all product configurations, that all required interfaces are provided, therefore, no unresolved dependencies happen in any product. This is compliant with Definition 1.

In what follows we discuss the issue of extracting interfaces from assets in Section 3.1. Then, in Section 3.2 we use the motivating example to discuss the intuition behind the verification process. Finally, in Section 3.3 we show how we formalize the safe composition verification using Alloy, and how we can retrieve ill-formed product configurations when needed.

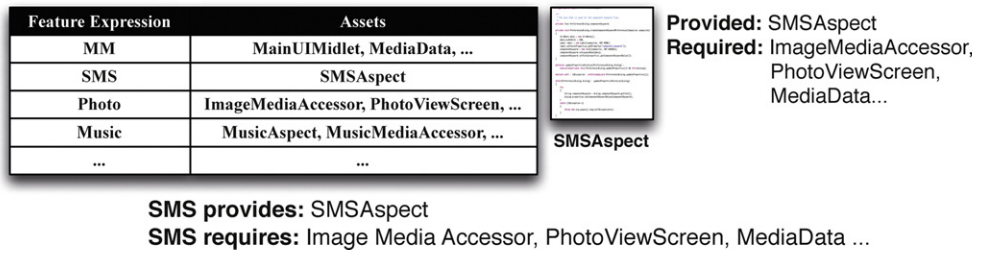
3.1. On extracting interfaces

This work proposes a language independent way of computing safe composition, give that provided and required interfaces are

SC01
Scenario: Reproduce media.
Description: This scenario allows a mobile phone user to start the reproduction of content.
Flow of events:

Step	User Action	System Response
SC01.1	User selects an existing folder from options Photo, Music.	The device populates the list of available content in an album.
SC01.2	User views the content of the album.	-
SC01.3	User selects one file from the list of Photo, Music files.	Media is displayed on the screen.
ADV01.1	User sets media as favorite.	System includes media in the favorites list.

Fig. 6. Resulting scenario after evaluating the CK.



Feature Expression	Assets
MM	MainUIMidlet, MediaData, ...
SMS	SMSAspect
Photo	ImageMediaAccessor, PhotoViewScreen, ...
Music	MusicAspect, MusicMediaAccessor, ...
...	...

SMSAspect
Provided: SMSAspect
Required: ImageMediaAccessor, PhotoViewScreen, MediaData...
SMS provides: SMSAspect
SMS requires: Image Media Accessor, PhotoViewScreen, MediaData ...

Fig. 7. Associating feature expressions with required and provided interfaces.

Feature Expression	Assets
MS Word	DocToXmlConverter.exe
...	...

Fig. 8. Part of the CK for TaRGeT PL, illustrating the selection of the Word to XML converter.

available, whether automatically derived or manually informed. To perform the verification, we only check the interfaces. Thus, we do not propose a new way of extracting interfaces, since the derivation process is an orthogonal concern, and it is not the focus of this work. The provided and required interfaces can often be automatically derived from assets (Thaker et al., 2007; Delaware et al., 2009; Kästner and Apel, 2008; Apel et al., 2010). The derivation process also has limitations. For instance, TaRGeT is an SPL of model-based test generation tools that generate functional tests from use case specifications written in a structured language (Ferreira et al., 2010). It can use different kinds of input files to generate tests from use case scenarios with the notation described in Section 2.2. A possible input type is Microsoft Word. Therefore, to process the use cases and generate tests, the tool must convert the document to XML.

Fig. 8 shows that when we select the MS Word feature, we must include the converter in the final product. Listing 1 shows the code responsible for loading and running the executable. If we have a product that attempts to load this file without it being provided, it will result in an error at runtime, not possible to detect when compiling the product.

We found other similar examples in the TaRGeT SPL (Ferreira et al., 2010). Currently, there are no widely available tools that can perform such analysis, and building one that does so might depend on the customer needs, resources and a cost–benefit analysis. Therefore, our approach assumes that sometimes it is useful to complement the interfaces manually, since we can implement an SPL using different kinds of assets, instead of a single language. It might not be possible to extract all desirable interfaces automatically.

Listing 1. Class that executes the Word to XML converter.

```
public class WordDocumentProcessing {
    . . .
    public static final String EXE = "DocToXmlConverter.exe";
    . . .
    try {
      p = Runtime.getRuntime().exec(DLL_FOLDER +EXE);
    }
    catch (Exception e) {. . .}
    . . .
}
```

Finally, these interfaces also depend on the variability implementation mechanism and language used in the SPL. The CK model we use enables us to deal with multiple languages for implementing an SPL. In some cases, there are not interface extraction techniques available. For example, to analyze TaRGeT requirements, we had to implement our own analysis, to extract the provided and required interfaces. So, our approach can be used together with different interface extraction techniques. We can use more precise mechanisms such as type systems, as well as analysis that do not capture too much information, therefore providing less precision. We could even use tailored analysis that could capture advanced dependencies, such as the one we show in Listing 1. Therefore, depending on which interface extraction technique we use together with our approach, we might have the same precision as an SPL-aware type system (Delaware et al., 2009; Schaefer et al., 2011). To perform the verification, we only require that such interfaces are available.

$$
\boxed{
\begin{array}{c}
\textbf{constraintsCK: Provided}_{ck} \Rightarrow \textbf{Required}_{ck}\\
\textbf{Provided}_{ck}\textbf{: } \bigwedge \textbf{Provided}_{item}\\
\textbf{Required}_{ck}\textbf{: } \bigwedge \textbf{Required}_{item}\\
\textbf{Provided}_{item}\textbf{: (FeatExp} \Rightarrow \bigwedge \textbf{Provided)}\\
\textbf{Required}_{item}\textbf{: (FeatExp} \Rightarrow \bigwedge \textbf{Required)}
\end{array}
}
$$

Fig. 9. Propositional logic codification rules for CK.

This way, we can verify SPLs implemented using different kinds of assets, ranging from code to requirements, as we detail in Section 4.

3.2. *Intuition for the verification*

While the FM gives us the domain constraints, represented by the set of product configurations, the CK specifies some constraints and can be used to derive the provided and required interfaces of the assets. Fig. 7 presents how we associate feature expressions to interfaces. As discussed, the CK associates features, like **SMS**, to assets, such as **SMSAspect**, or transformations over assets, such as **select scenario SC01**. Each of these has provided and required interfaces. Therefore, for each CK item, we have the associated provided and required interfaces.

Given that we have such interfaces, we can translate them to a logical proposition. For each CK item, we generate propositions based on the associated required and provided interfaces. In this case, each interface name is an atom in the proposition. Fig. 9 illustrates the codification rules. For example, for the CK item related to the **SMS** feature, the proposition representing the provided interface is given as in the following:

$$SMS \Rightarrow SMSAspect.$$

If we select the **SMS** feature in a product configuration, the feature expression *SMS* evaluates to true, and *SMSAspect* is provided. We translate required interfaces likewise, based on the conjunction of the required interfaces. For the same CK item, the proposition representing the required interfaces is

$$SMS \Rightarrow ImageMediaAccessor \wedge PhotoViewScreen \wedge MediaData \ldots$$

We use the conjunction ($\bigwedge$) of the provided propositions for each CK item to express the provided interfaces for the entire 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 SPL products, required interfaces should be provided. If this is the case, safe composition of the SPL is guaranteed, since for all products there would be no unresolved dependencies.

To check the entire SPL, we use the FM, since it represents the set of product configurations. Rules for translating a FM into propositions have been discussed previously (Batory, 2005), so we omit the details here. Every feature relationship (root, optional, mandatory, alternative, or) has a specific translation to propositional logic, representing its semantics. So, we could represent the release 6 of the FM from Fig. 1 with the proposition that follows:

$$
\begin{array}{l}
semanticsFM:\\
MM \wedge (MM \Leftrightarrow Management) \wedge (MM \Leftrightarrow Media)\\
\wedge (Management \Rightarrow Sorting) \wedge (Management \Rightarrow SMS)\\
\wedge (Management \Rightarrow Copy) \wedge (Management \Rightarrow Favorites)\\
\wedge (Media \Leftrightarrow (Music \vee Photo)).
\end{array}
$$

To verify safe composition, we need to check if all products are absent of unresolved dependencies (see Definition 1). To do so,

Table 1
Meaning of Alloy operators as used in this work.

Operator	Meaning
and	$\wedge$
or	$\vee$
one	a signature has one element
+	$\cup$
in	$\in$
extends	$\subseteq$

we relate the propositions *semanticsFM* and *constraintsCK*. Domain constraints (FM) must satisfy implementation constraints (CK), likewise previous works (Czarnecki and Pietroszek, 2006; Thaker et al., 2007; Batory, 2005). The proposition we then need to check is the following:

$$semanticsFM \Rightarrow constraintsCK.$$

If this proposition evaluates to true, we have that all products respect the provided and required interfaces, that is, for all products there are no unresolved dependencies. If not, we have that at least one product has unresolved dependencies.

3.3. Formalization

Alloy is a formal object-oriented modeling language, based on first-order logic, that gives a mathematical notation for specifying objects and their relationships (Jackson, 2006). Alloy specifications are similar to Object Constraint Language (OCL) and Unified Modeling Language (UML) class diagrams, but Alloy has a simpler syntax, type system and semantics, being designed for automatic analysis. Moreover, Alloy is a fully declarative language. We use Alloy because it has a formal semantics. Besides, due to its tool support, which can perform automatic analysis over specifications, we can use it with 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 (Gheyi et al., 2006). Table 1 summarizes the meaning of Alloy operators used throughout the text.

An Alloy specification contains a number of signature paragraphs (**sig**). We can think of Alloy in terms of set theory. So, a signature paragraph introduces a new set of objects. The following fragment illustrates signature declarations for the boolean idiom we use and declarations for features and interfaces. We define an abstract signature **Bool**, and two signatures **True** and **False** extending it. The **extends** keyword means that these are disjoint subsets of **Bool**. The **one** keyword denotes multiplicity, so there is always exactly one instance of a signature. Therefore, every **Bool** is either **True** or **False**.

```
abstract sig Bool {}
one sig True, False extends Bool {}
```

We define features, such as SMS, and interfaces, such as SMSAspect, using the **in** keyword. By using it together with the **one** keyword, we declare that features and interfaces are singleton elements of **Bool**, and can assume **True** or **False** values. We do it in this way since depending on the product configuration, features and interfaces can be included (**True**) or not (**False**).

```
one sig MM, Management, Media, Photo, ···SMS in Bool {}
one sig SMSAspect, ImageMediaAccessor, PhotoViewScreen ···in Bool {}
```

In Alloy, predicates (**pred**) are named formulae. If all constraints listed in the body are satisfied, the predicate evaluates to true, otherwise to false. We use predicates to represent the FM and CK as propositions, encoded using the rules discussed in the previous section. The following fragment specifies the semantics of the FM from Release 6 of MobileMedia. The predicates for specifying FM relationships, such as root and optional, have been previously specified, since we reuse a theory for encoding FMs in Alloy (Gheyi et al., 2006). For example, we declare the *or* relationship using two arguments: the parent feature, and the set of features, declared using the +union set operator.

```
pred semanticsFM[] {
  root[MM] and
  mandatory[Media,MM] and mandatory[Management,MM] and
  orGroup[Media, Photo+Music] and
  optional[SMS, Management] and optional[Sorting, Management] and
  optional[Copy, Management] and optional[Favorites, Management]
}
```

We specify in the constraintsCK predicate the implementation constraints. The provided and required predicates follow the rules in Fig. 9. That is, associating feature expressions with the conjunction of the provided or required interfaces for CK items. While the structure of the constraintsCK function is always the same, that is, *provided* ⇒ *required*, the predicates for provided and required vary according to the SPL used. We detail part of the provided predicate for the release 6 of MobileMedia in what follows, and omit the required for brevity, since we use the same rationale for constructing it — conjunction of the interfaces. We see that the first row of the provided[] predicate states that, when MM is selected in a product configuration, interfaces such as MainUIMidlet and MediaController are provided, that is, set to a **True** value.

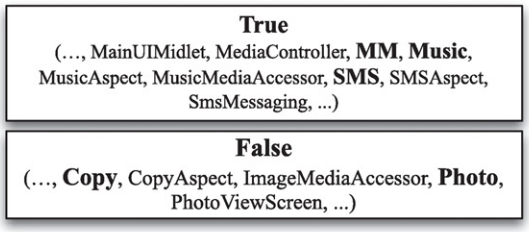
```
pred provided[] {
  selected[MM] ⇒ provide[MainUIMidlet+MediaController+··] and
  selected[SMS] ⇒ provide[SMSAspect+SmsMessaging+··] and
  selected[Photo] ⇒ provide[ImageMediaAccessor+··] and
  selected[Music] ⇒ provide[MusicAspect+MusicMediaAccessor+··] and
  ...
}
pred required[] {...}
pred constraintsCK[] {
  provided[] ⇒ required[]
}
```

To determine safe composition for an SPL, we need to check if all products are well-formed. We do this relating the FM and the CK, as the previous section discusses. To check this in Alloy, we use assertions (**assert**). Assertions are claims that something must be true due to the rest of the model. We verify (or falsify) assertions using the **check** command, that searches for counterexamples of an assertion. A counterexample, in our context, is an assignment of features and interfaces to **True** and **False** that violates the assertion. We must specify a scope, the maximum size of each signature in the model that will be considered. To actually check the assertion, we use the Alloy Analyzer tool (Jackson et al., 2000). The analysis is sound and complete up to the given scope. The following fragment illustrates how we use Alloy to specify the assertion and its verification, providing an implementation for Definition 1.

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

There are two possible results for a **check** command. If it cannot find counterexamples, the assertion is said to be true for the scope defined. When the tool finds a counterexample, this means that there is at least one ill-formed product configuration in the SPL.

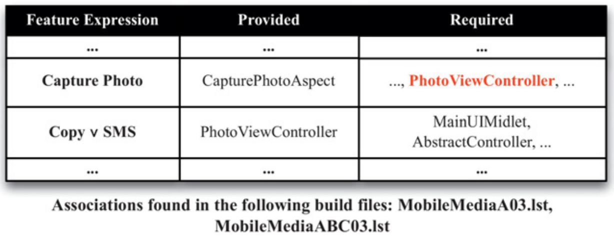
In the first part of Section 2 we show that the Release 6 of MobileMedia has products with unresolved dependencies. So, when checking the **verifySPL** assertion for this SPL, we find a counterexample. Since this is an assignment of features and interfaces to **True** and **False** that violates the assertion, we can examine it to retrieve the ill-formed product configuration. Fig. 10 illustrates a possible counterexample returned by the Alloy Analyzer, with the feature names highlighted in bold. We have two boxes, representing the two possible values that our features and interfaces can assume, **True** or **False**. Features in the **True** box are selected, while features in the **False** box are not. So, this counterexample is related



True
(…, MainUIMidlet, MediaController, **MM**, **Music**, MusicAspect, MusicMediaAccessor, **SMS**, SMSAspect, SmsMessaging, …)

False
(…, **Copy**, CopyAspect, ImageMediaAccessor, **Photo**, PhotoViewScreen, …)

Fig. 10. Possible counterexample returned by the Alloy Analyzer (Jackson et al., 2000) when checking the verifySPL assertion for Release 6 of MobileMedia.



Feature Expression	Provided	Required
...	...	...
Capture Photo	CapturePhotoAspect	..., PhotoViewController, ...
Copy v SMS	PhotoViewController	MainUIMidlet, AbstractController, ...
...	...	...

Associations found in the following build files: MobileMediaA03.lst, MobileMediaABC03.lst

Fig. 11. Part of the association of feature expressions to interfaces for Release 7 of MobileMedia (Figueiredo et al., 2008), illustrating the Capture Photo problem. Below the figure we show in which build files this association appears.

to the product configuration {**SMS**, **Music**}. The other names in the counterexample are interface names. The ones included in the True box are provided, while the ones in the **False** box are not — see that the ImageMediaAccessor and PhotoViewScreen classes show up in the **False** box. These classes are the source of this problem, since they are referenced by the SMSAspect, related to the **SMS** feature. In Section 4.3, we further discuss our alternative approach for reporting ill-formed product configurations.

In our context, the top-level signature we have is **Bool**. Although there are a number of features and interfaces, they are all subset signatures of **Bool**, so they can be either **True** or **False**. Therefore, the scope we use for checking our assertion does not need to be bigger than 2, in any scenario, since the specification is based on propositional logic, which is decidable. Thus, the Alloy Analyzer works as a theorem prover in this context, the analysis is sound and complete. 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 cannot find counterexamples, we consider the SPL well-formed.

We have built a command-line tool in Java, using the Alloy API, which automates this verification process (Teixeira et al., 2012). Moreover, we have integrated this tool with Hephaestus, a tool suite used for managing SPL variabilities (Bonifácio et al., 2009). It is important to highlight that the tool 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 product configurations. We further discuss specifics about the implementation in Section 4.3.

4. Evaluation

This section presents the evaluation of our approach using different SPLs. We intend to evaluate whether the safe composition problem happens in different SPLs, implemented with a variety of techniques, considering multiple kinds of artifacts, such as code and use case scenarios, and if our approach is general enough to find such problems. Moreover, we also investigate if our tool scales to deal with larger scale case studies that allow generating thousands of products. For all SPLs evaluated, we collect the time for verifying safe composition, the time for reporting all ill-formed product configurations, and the number of ill-formed product configurations.

First, we briefly discuss the analysis of the code for seven releases of the MobileMedia SPL (Figueiredo et al., 2008) in Section 4.1. In Section 4.2.1 we detail the analysis of the code for release 6 of the TaRGeT SPL, and in Section 4.2.2 we detail the analysis for the use case scenarios from the same release. In Section 4.3 we discuss the performance of our analysis and our new approach for reporting ill-formed product configurations. Finally, in Section 4.4 we discuss other issues related to this evaluation.

4.1. MobileMedia

We have evaluated seven releases of the MobileMedia SPL, which we first discuss in Section 2. Fig. 1 illustrates how the FM evolved until Release 7. We extracted the CK from the existing build files that were used to build products, which associated features to classes and aspects. To extract dependencies between assets, we have used Soot[3] to analyze the code and retrieve the provided and required interfaces. We express required interfaces as syntactic dependencies between assets and provided interfaces as the asset names. For this evaluation, we capture dependencies specific to the MobileMedia artifacts, focusing on classes and aspects under the *lancs.mobilemedia* package. Although we need this information, it is important to remind that our approach is orthogonal to the interface extraction part, as we discuss in Section 3.1.

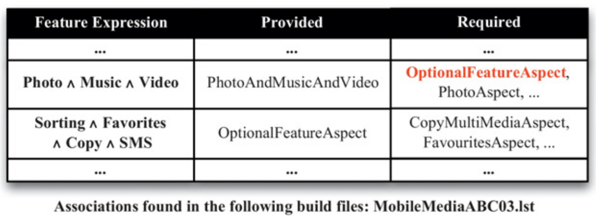
The first release is not considered an SPL, since it is a single product. Therefore, we just need to make sure that it is also a valid AspectJ program, which is the case. When we perform the verification for releases 2–5, the tool does not find any problems. These releases are smaller and simpler, ranging 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. 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. Verification for this release detects 8 ill-formed product configurations, due to the dependency between **SMS** and **Photo**, discussed in Section 2.1.

4.1.1. Release 7

This release introduces another media type: **Video**. It also adds the ability to capture photos and videos. The SPL size grows to 4 KLOC. The number of product configurations increases to 272. When performing the verification for this release, we find 116 ill-formed product configurations. Besides the problem caused by the dependency between **SMS** and **Photo**, there are two new problems. Fig. 11 illustrates the problem related to the **Capture Photo** feature, using a tabular view of the association of feature expressions to interfaces. This feature provides the **CapturePhotoAspect**, which depends on the **PhotoViewController** class, provided only when we select **Copy** or **SMS** features. Therefore, a solution for this problem is to change the feature expression to **Copy** $\vee$ **SMS** $\vee$ **CapturePhoto**.

Another problem happens with the aspect **PhotoAndMusicAndVideo**, included when all media types are present in a product configuration. Fig. 12 illustrates this problem. This aspect depends on **OptionalFeatureAspect**. This is due to a precedence declaration, used to organize the order in which aspects should apply. There are

[3] http://www.sable.mcgill.ca/soot/.



Feature Expression	Provided	Required
...	...	...
Photo ∧ Music ∧ Video	PhotoAndMusicAndVideo	OptionalFeatureAspect, PhotoAspect, ...
Sorting ∧ Favorites ∧ Copy ∧ SMS	OptionalFeatureAspect	CopyMultiMediaAspect, FavouritesAspect, ...
...	...	...

Associations found in the following build files: MobileMediaABC03.lst

Fig. 12. Part of the association of feature expressions to interfaces for Release 7 of MobileMedia (Figueiredo et al., 2008), illustrating the Photo ∧ Music ∧ Video problem. Below the figure we show in which build files this association appears.

aspects that only have precedence declaration in its body. This is the case of **OptionalFeatureAspect**, for example. Listing 2 shows the code for this aspect, associated with the expression **Sorting** ∧ **Favorites** ∧ **Copy** ∧ **SMS**, thus, included in a product when we jointly select all optional features.

Listing 2. Code for OptionalFeatureAspect.aj.

```
package lancs.mobilemedia.optional;
import lancs.mm. . .CopyMultiMediaAspect;
. . .
public aspect OptionalFeatureAspect {
  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.

Without such a tool, we could only find such problems if we manually tried to generate a single SPL product that is ill-formed. Therefore, these problems could go unnoticed until later in the SPL development. The tool aids on finding problems faster, allowing also to develop solutions faster as well.

4.2. TaRGeT

TaRGeT is an SPL of model-based test generation tools that generate functional tests from use case specifications written in a structured language (Ferreira et al., 2010). Among other variations captured by a FM with 42 features, different products support specifications and tests in different formats like MS Word, HTML and XML. In addition, other major features include: a Controlled Natural Language, to avoid spelling and grammatical errors; Test Case Selection, to filter test suites due to time or budget restrictions; Consistency Management, to maintain the consistency of different test suites generated over time.

TaRGeT has a number of releases, but only in its 4th release it became an SPL. We focus on release 6 since it has both the code and requirements structured as an SPL. Also, the FM contains all kinds of feature relationships, such as optional, or groups, and alternative groups, while in earlier releases, there was reduced variability. Fig. 13 shows the TaRGeT FM for release 6, which is used to describe variability for both the code and requirements. It has 31 implemented features. The FM allows generating more than 60 thousand products. Although some SPLs allow a large number of products to be generated, in practice, companies focus on a reduced number of products. In the case of TaRGeT, the main focus was to support a reduced number of customer configurations. Nonetheless, to evaluate whether our approach scales to analyze the SPL as a whole, we take into account all of TaRGeT product configurations in this study. Moreover, in other contexts, for example, as in the Linux kernel (Lotufo et al., 2010), there is the need to embrace all possible products. In this cases, there is the need for maintaining the high SPL configurability, thus it is important to evaluate whether our approach scales, so we can apply it in such cases. Besides the implementation, TaRGeT also has its use case scenarios written using MSVCM (Bonifácio and Borba, 2009), in the same way as the ones we describe in Section 2.2. They describe the functional behavior of the features. The two TaRGeT case studies, although from the same release, are only related by the FM. In one case study we focus only in the requirements and in the other we focus in the code.

4.2.1. Code assets

As an SPL of Eclipse based desktop applications, TaRGeT uses Eclipse RCP plug-in mechanism (McAffer and Lemieux, 2005) to implement most variations. Therefore, instead of implementing features using only aspects, as MobileMedia does, it uses the extension point mechanism (similar to subtype polymorphism) of Eclipse to implement variability. Fig. 14 shows that the common responsibilities of the application are distributed in four distinct basic plug-ins:

- **Core plug-in:** responsible for system start-up and setting up the workspace and perspective of the RCP application.
- **Common plug-in:** implements basic entities to represent use cases documents and test case suites. Besides that, it contains parsers for different input formats and provides support to new implementations for new input formats.
- **Project Manager plug-in:** contains operations and exceptions to handling projects, test case generation algorithm, basic GUI components to be extended in the implantation and support for implementing variability to make TaRGeT compatible with different formats of input use case documents.
- **Test Case Generation plug-in:** generates test case suites in different formats and provides support to extend TaRGeT with new implementations for different output formats.

TaRGeT also uses other variability implementation mechanisms, including AspectJ aspects to implement some feature interaction. Some features — including **Language**, **Environment** and **Branding** — are implemented using a combination of property files, and auxiliary files, such as images. Finally, some conditional compilation is used in property files. However, since there are not many files using this technique, we have eliminated the tags by generating all variants of these files. The total code size is approximately 32KLOC.

Release 6 already has the FM and CK structured using Hephaestus (Bonifácio et al., 2009). Therefore, unlike the MobileMedia case, we did not have to derive the CK, since it already existed. It consists of feature expressions associated to assets, which can be classes, aspects, property files, images, and so on. CK evaluation generates the build files needed for creating an executable product.

To obtain the provided and required interfaces, in a similar way to the MobileMedia evaluation, we automatically calculated the syntactic dependencies between classes and aspects used to implement the features. However, we also refined interfaces manually

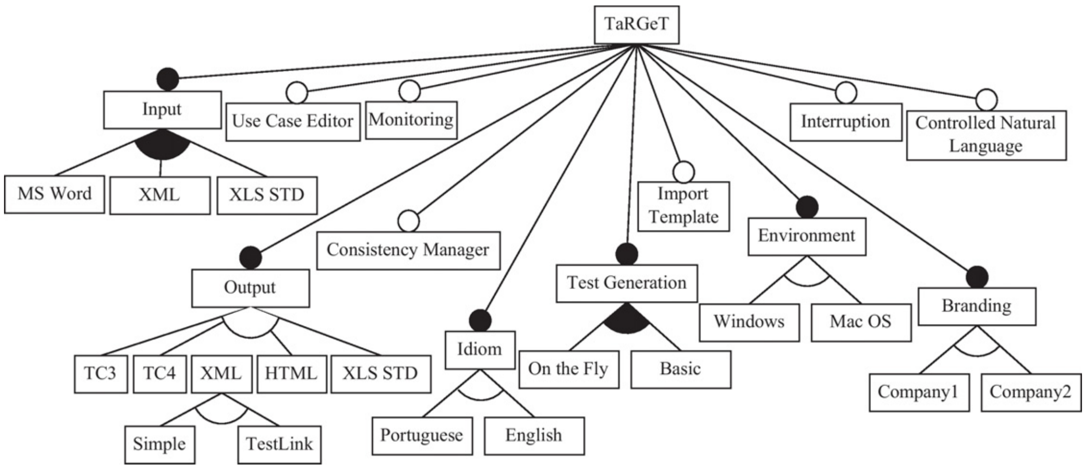


Fig. 13. TaRGeT FM for release 6. Higher resolution image at the online appendix (Teixeira et al., 2012).

to capture indirect dependencies, given that we had no special purpose tool for inferring this kind of dependence, such as the ones we mention in Section 3.1. With this information, we are able to verify safe composition. As we discuss, the Alloy specification is *automatically* generated, once we have the interfaces established.

Although this SPL is larger than MobileMedia, we found a single safe composition problem when analyzing the code. The problem happens with the **ExceptionMonitoring** aspect, included in the product when we select the **Monitoring** and **MS Word** features. Listing 3 shows that this aspect references the **PhoneDocument** class, when defining a pointcut. Since TaRGeT was initially developed in a joint project with a company for generating test cases for mobile phones, the use case documents were called phone documents. This class represents a use case document. It was later renamed as **UseCaseDocument**, since test cases generated by TaRGeT can be used outside the mobile context.

Listing 3. ExceptionMonitoring aspect.

```
public aspect ExceptionMonitoring {
    ...
    pointcut createWordDocumentObj(List<String>doc, boolean e):
    execution(
      public List<PhoneDocument>
        WordProcessing.createFromWord(List<String>, boolean))
      && args(doc,e);
    ...
}
```

TaRGeT developers explained that the **Monitoring** feature was not being used by any particular client of the tool. Therefore, the problem went unnoticed by them, since no products containing both **Monitoring** and **MS Word** features were being generated. However, although a simple problem, it affected more than 18 thousand products of the SPL. The solution to this problem consists on renaming **PhoneDocument** to **UseCaseDocument**. This also reinforces the need for tools to help developers refactor SPLs (Borba,

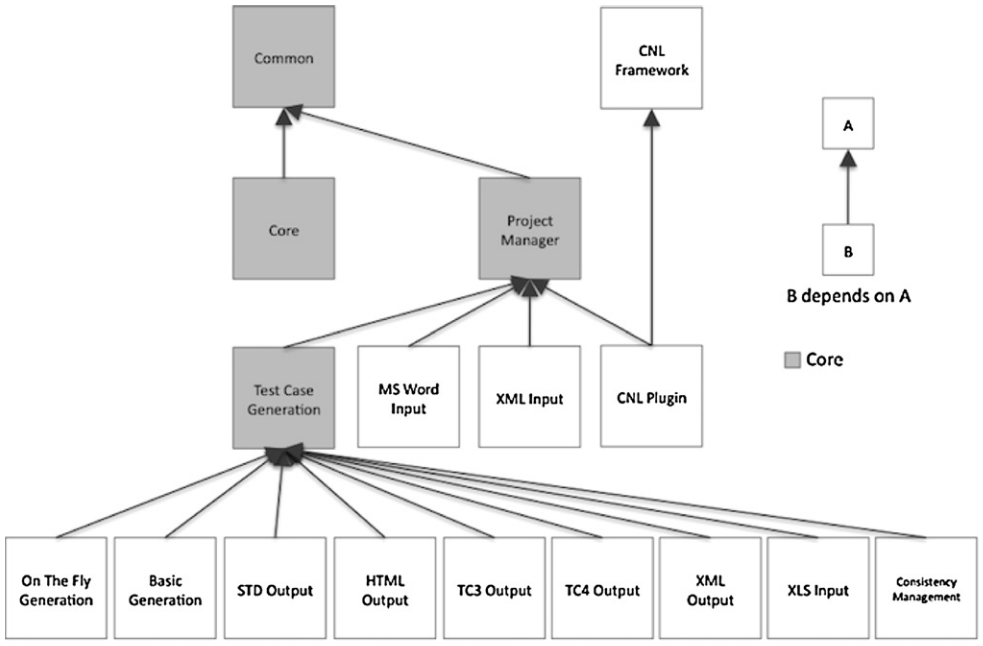


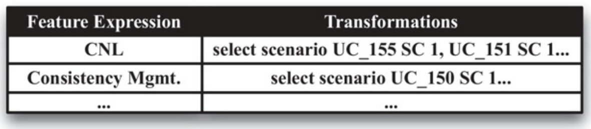
Fig. 14. TaRGeT architecture.

UC_155 SC 1
Scenario: Visualizing unknown terms synonyms.
Description: Shows unknown terms synonyms.
From Step: UC_150#1M
To Step: End
Flow of events:

Step	User Action	System Response
1M	Choose one unknown term from the use case document in the Term combo.	The selected term and its synonyms are displayed in the 'Add New Lexical Term' dialog box.

Fig. 15. MSVCM Scenario for the Controlled Natural Language feature, describing the steps for visualizing unknown terms synonyms in TaRGeT.

2009) without making such mistakes. After performing an SPL refactoring, it is important to verify if safe composition is preserved. It is also important to notice that we only analyzed a single and stable release from TaRGeT. In practice, the other safe composition problems that happened prior to this release, were found and corrected between the commits using brute force, by individually testing some of the SPL products.



Feature Expression	Transformations
CNL	select scenario UC_155 SC 1, UC_151 SC 1...
Consistency Mgmt.	select scenario UC_150 SC 1...
...	...

Fig. 16. Part of the MSVCM CK for TaRGeT, to illustrate the typographical error.

4.2.2. Requirements

Safe composition problems can also happen in requirements documents, as we mention in Section 2. We can make mistakes when referring to parameters, scenarios, steps, and annotations. In release 6, TaRGeT's requirements have also been structured with the aim of exploring the inherent reuse associated with SPLs. In particular, using the MSCVM approach (Bonifácio and Borba, 2009). The FM used is the same as in Fig. 13, and the CK uses the transformations associated with use case scenarios that we discuss in Section 2.2: select scenario; evaluate advice; and bind parameter.

For each of these transformations, we need the provided and required interfaces, to enable the verification. In this case, interfaces are scenario names, steps, parameter names, and annotations. For example, Fig. 15 presents a use case scenario, therefore we use as provided interfaces the scenario name (UC_155 SC 1), the scenario steps, in this case a single one (UC_155#1M). Had we used parameters or annotations, as in the motivating example, we would include them in the provided interfaces as well. Required interfaces in this case, consist of the from and to steps. When we use the bind parameter transformation, it requires that the parameter must exist to be bound. Finally, in the evaluate advice, we use the pointcut definition as the required interface. We built an interface extractor which automatically extract the interfaces from the use case model and associate them with the feature expressions from the CK. Thus, since we have the same structure of feature expressions associated to interfaces, we use the same encoding described in Section 3 to verify the SPL.

After running the analysis, unlike in the code version, a number of problems were found, that resulted in most of the products being defined as ill-formed. In contrast to the code analysis, we believe that this number was high because this is the first release where MSVCM was used, while in the code version, the majority of the code has been used and tested throughout formal and informal releases. Besides, TaRGeT developers did not have tool support to help analyze the resulting MSVCM product-specific use case scenarios, therefore, it was hard to detect such problems, since such documents could contain over a hundred pages. In what follows, we discuss some of the problems we found.

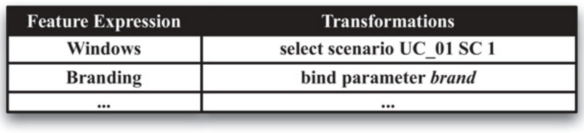
References issues due to typographical errors happened more than once. For example, Fig. 15 shows the use case scenario 1 of UC_155, which belongs to the **Controlled Natural Language** feature (CNL), as Fig. 16 shows. We notice that in the from step field, it references the step 1M from the use case scenario 1 from UC_150, which belongs to the **Consistency Management** feature. It should in fact reference the step 1M from UC_151, which also belongs to the **Controlled Natural Language** feature. Therefore, in this example, there are safe composition problems in all products that contain the **Controlled Natural Language** feature without the **Consistency Management** feature, which we detect by this unsatisfied dependence. Other similar typographical mistakes happened throughout the specification, which caused many products to be considered ill-formed.

We also found another type of problem, when we try to bind a parameter that does not exist in the resulting specification, in a similar way to what we illustrated in the motivating example, in Section 2.2. For example, see in Fig. 17 the scenario that describes

UC_01 SC 1
Scenario: Opening TaRGeT.
Description: Start up the TaRGeT with .NET and Java runtime environment installed.
From Step: Start
To Step: End
Flow of events:

Step	User Action	System Response
1M	Start the TaRGeT tool.	The {brand} splash screen is displayed. Verify the requirements document to check the splash screen.
2M	Wait some seconds.	The TaRGeT is started. No TaRGeT project is opened. A background image is displayed (see requirements document for more details)

Fig. 17. MSVCM Scenario describing the steps for opening TaRGeT in the Windows environment.



Feature Expression	Transformations
Windows	select scenario UC_01 SC 1
Branding	bind parameter *brand*
...	...

Fig. 18. Part of the MSVCM CK for TaRGeT, to illustrate the bind parameter problem.

opening TaRGeT in the Windows environment, with the.NET and Java runtime environment installed. Fig. 18 shows part of the MSVCM CK, illustrating that this scenario is correctly associated with the **Windows** feature, child of the **Environment** feature. We notice the *brand* parameter in the first step (1M). In Fig. 18, we also see that this parameter is bound to the **Branding** feature. Therefore, when generating the resulting scenarios, *brand* should contain the company name selected in the product configuration. However, there are no scenarios describing opening TaRGeT in the Mac OS X environment, which was introduced in this release. Also, there is no other scenario that uses the *brand* parameter. Therefore, in all products where we select the **Mac OS X** feature, an alternative feature to **Windows**, we have an unresolved reference. We try to bind a parameter that does not exist. In this case, this single problem causes half of the SPL to be ill-formed, since **Environment** is a mandatory feature directly connected to the root, with two mutually exclusive children. We could fix it by creating a scenario to describe opening TaRGeT in the Mac OS X environment.

Another problem we found more than once is related to optional from and to step fields. For example, Fig. 19 illustrates the use case scenario 6 from UC_110. Fig. 20 show that it belongs to the **Testlink XML Output** feature. We see in Fig. 19 that this scenario references the step 2M from the use case scenario 1 of UC_110. This scenario belongs to the **On the Fly** feature, as we can see in Fig. 20. However, the FM allows us to have products in which we select **Testlink XML Output** and do not select **On the Fly**. Therefore, we reference a step that is not being provided. What caused this problem is that only products with both **Testlink XML Output** and **On the Fly** were generated. Actually, to avoid this problem, we should have to modify the feature expression in the CK to **Testlink XML Output** $\wedge$ **On the Fly**, since this scenario only makes sense when we have both features selected.

Finally, one other problem that we found is also related to the from step field, when referencing steps introduced by advices. For example, Fig. 21 shows the use case scenario 3 from UC_110, which belongs to the **Import Template** feature. It references the step 1B from the same UC_110. However, this step does not exist. It was extracted to an advice. Therefore, the from step should be updated to reference the step in the advice. This is another example where an SPL refactoring tool would be useful, this time in a different context.

4.3. Performance

As we mention in Section 3.3, we built a command-line tool for verifying safe composition of CK-based SPLs. With this tool, it is possible to verify whether an SPL is well-formed, and if it is ill-formed, it can report the list of ill-formed product configurations. We have executed the analyses on a 64-bit machine with a Intel Core i5 CPU running at a 2.4 GHz frequency with 8 GB of memory, 256KB of L2 cache per core, and 3MB of L3 cache on the Mac OS X 10.7.4 operating system. We used the Alloy Analyzer API version 4.1.10.

Table 2 summarizes the time for analyzing whether each of the MobileMedia and TaRGeT releases are well-formed, and the time for reporting all ill-formed product configurations, if there are safe composition problems. We list, for each SPL, the total number of products and the number of ill-formed product configurations reported by the tool. There are two columns reporting the time. The **Verify** column lists the time spent to verify only whether the SPL is well-formed or not. For each SPL, we ran the analysis 5 times to obtain an average time, which we show in Table 2. In the **Report** column, we list the average time, in seconds (s) or minutes (m), spent by our tool to report the complete list of ill-formed product configurations. This step is only necessary when the SPL is ill-formed.

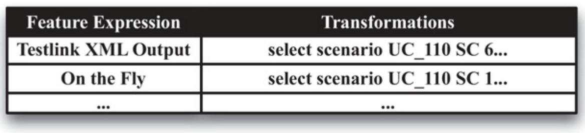
We separate the verify and report times because one might only want to verify whether the SPL is safe. When our tool detects that the SPL is ill-formed, the counterexample already indicates an ill-formed product configuration. Therefore we could find the problem source using this information, fix it and run the analysis again. For example, if we used such practice with TaRGeT r6 code, we would fix the 18,432 ill-formed product configurations at once, since they were caused by a single problem. Thus, although knowing the exact product configurations that are ill-formed can be useful, in this case it would actually be redundant, since all of them expose the same single problem. However, it is important to evaluate if our approach scales to deal with thousands of products.

Previously (Teixeira et al., 2011), when our tool detected that there was a safe composition problem in the SPL, we used the Alloy Analyzer counterexample to retrieve an ill-formed product

UC_110 SC 6
Scenario: Generating Test Suites through the On The Fly Generation.
Description: Generates test suite without generating the requirements specification document.
From Step: UC_110 SC 1#2M
To Step: End
Flow of events:

Step	User Action	System Response
1E	Go to Test Cases tab in the On The Fly Generation Editor view and click on the Save Test Suite button.	The system asks the user if it wants to generate the requirements specification file.
2E	The user selects the No button.	The Test Cases Generation Summary pop up is displayed informing the number of generated test cases and the total time required to complete the process.
3E	Choose an invalid .xls template and press Open.	A message is displayed informing that the .xls file is invalid and that the template information will not be imported.

Fig. 19. MSVCM Scenario describing the steps for opening TaRGeT in the Windows environment.



Feature Expression	Transformations
Testlink XML Output	select scenario UC_110 SC 6...
On the Fly	select scenario UC_110 SC 1...
...	...

Fig. 20. Part of the MSVCM CK for TaRGeT, to illustrate the optional from step field problem.

UC_110 Scenario 3

Description: Generating tests without a template

From Step: UC_110#1B

To Step: End

Flow of events:...

Fig. 21. Scenario 3 from UC_110, belonging to the **Import Template** feature.

configuration. Then we would remove it from the FM semantics and verify it again. For example, we would find an ill-formed product configuration, say *product1[]*, and update our assertion. So, instead of *semanticsFM[]* $\Rightarrow$ *constraintsCK[]* we would use *semanticsFM[]* $\wedge\neg$ *product1[]* $\Rightarrow$ *constraintsCK[]*. The updated assertion guarantees that the product configuration already identified as ill-formed is not taken into account anymore. After updating this predicate, we would check the assertion again, iterating through this process of updating assertions until there would be no counterexamples returned, that is, we found all ill-formed product configurations. However, this becomes inefficient as the SPL grows, since for each ill-formed product configuration found, the assertion also grows, making the verification slower. The analysis for TaRGeT code took over 12 h to report all ill-formed product configurations, while for its requirements it took over 24 h.

In this article, we improve the results from our previous work (Teixeira et al., 2011), by changing how we report the list of ill-formed product configurations, making our analysis family-product-based instead of family-based (Thuem et al., 2012). In our family-product approach for reporting ill-formed product configurations, we first verify the entire SPL (family-based) to detect whether there is a safe composition problem. If a problem exists, we verify all product configurations individually against the CK. That is, for each product configuration from the FM, we check the assertion *product*$_n$*[]* $\Rightarrow$ *constraintsCK[]*. While this might seem inefficient, we can run the analysis much faster, since the product predicates are actually smaller than the FM predicate and do not grow with the number of ill-formed product configurations. It is important to notice that we do not actually build the products, but only generate

Table 2
Summary of our analyses. **# Products**: the total number of products for the SPL. **# ill-formed**: the total number of ill-formed product configurations found. **Verify**: average time, in seconds (s), to verify whether the SPL is well-formed or not. **Report**: average time, in seconds (s) or minutes (m), to report all ill-formed product configurations.

	# Products	# ill-formed	Verify	Report
Real SPLs				
MobileMedia r2	2	0	0.3 s	–
MobileMedia r3	4	0	0.4 s	–
MobileMedia r4	8	0	0.5 s	–
MobileMedia r5	16	0	0.5 s	–
MobileMedia r6	48	8	0.7 s	2.8 s
MobileMedia r7	272	116	0.9 s	7.4 s
TaRGeT r6 Code	64,512	18,432	1.2 s	~13 m
TaRGeT r6 Reqs	64,512	64,512	1.2 s	~20 m
Artificial SPLs				
TaRGeT r7 Code	129,024	36,864	1.4 s	~26 m
TaRGeT r8 Code	258,048	165,888	1.6 s	~56 m

its abstractions (propositional formulae) using Alloy. With the new implementation, instead of hours to analyze TaRGeT, we can report all ill-formed product configurations in minutes, as we show in Table 2.

Fig. 22 shows the time to report ill-formed product configurations for the SPLs we analyzed. We see that although the number of ill-formed product configurations grows thousands of times between release 6 of MobileMedia and the requirements from release 6 of TaRGeT, the report time does not follow the same pattern. For example, from releases 6 and 7 of the MobileMedia SPL, the number of ill-formed product configuration grows over 14 times, but the time for reporting all ill-formed product configurations grows less than 3 times. Comparing the different TaRGeT releases, the number of ill-formed product configurations grows 3.5 times when we look at the code version versus the requirements version. In its turn, the report time grows approximately 1.5 times.

To better understand how our approach handles even larger situations, we simulated adding new optional features to the code version of TaRGeT. The last rows on Table 2 summarize the results of the verification for each of the extra feature we added. First, we added a new optional feature directly connected to the root, doubling the number of products. The added feature did not introduce any safe composition problem. Therefore, we also doubled the number of ill-formed product configurations, to 36,864. The time for reporting ill-formed product configurations on this SPL roughly doubles, taking now approximately 26 min (see Fig. 22). Then, we added another optional feature connected to the root, but now introducing a safe composition problem, simulating the same issue discussed in Section 4.2.1. The number of ill-formed product configurations now rises to 165,888, which is basically half of the products (since we added a problematic optional feature) plus the other 36,864 carried from the previous artificial release. Time for reporting all ill-formed product configurations now is 56 min (see Fig. 22). Although the number of ill-formed product configurations rises 4.5 times, the time roughly doubles again, as we doubled the total number of products in the SPL. Even though we now take almost 1 h to report the ill-formed product configurations, this would still be feasible to have as a nightly task in a software company.

We see that the numbers do not seem to suggest that there is a correlation between the report time and the number of ill-formed product configurations in an SPL, but rather that there is a relationship between the total number of products and the analysis time. It is to be expected that that the evaluation time doubles as the number of products in the SPL doubles. This is an exponential increase in complexity with respect to the number of products that are described by an SPL, but our evaluation shows that the approach works for SPLs of reasonable size. Even though we only simulated adding new features, we did it by using the same pattern as found in other existing features from TaRGeT. Also, we cannot conclude that the analysis time for any SPL with the same number of products as TaRGeT would be the same. We can notice that by comparing the time for analyzing TaRGeT code and requirements. Even though the total number of products is the same, the analysis time differs in a few minutes. This is due to the different kinds of assets used to implement the SPL, which result in different structures of the generated CK formulae, as well as to the number of ill-formed product configurations, which is higher in the requirements version. Therefore, even though we were able to analyze more than 250,000 products in less than an hour, it could be the case where a smaller SPL might take long to analyze, due to complex dependencies between the assets. However, from our findings, we believe that the analysis time grows roughly linearly with the total number of products in the SPL.

Finally, using product configurations for reporting ill-formed product configurations might also be interesting when considering

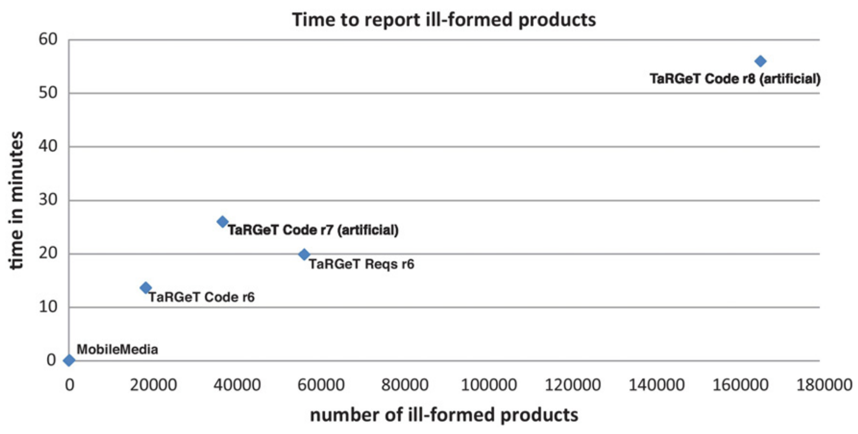


Fig. 22. Time to report ill-formed product configurations.

SPL evolution. After refactoring a feature, we could then verify only the products in which the feature is selected. The same reasoning could be applied to adding a new feature. Exploring this information could be an optimization that could reduce the analysis time in the context of larger scale SPLs.

4.4. Discussion

A possible alternative approach for the verification would be to generate all products in a brute-force manner. However, in MobileMedia, for example, the average time for compiling a single product configuration is around 50 s. For release 7, it would take almost 4 h for generating and compiling all products. On the other hand, the tool spends less than 6 s for verifying release 7 and reporting the list of 116 ill-formed product configurations. This reinforces the benefits of using an automated approach for verifying safe composition (Czarnecki and Pietroszek, 2006; Thaker et al., 2007; Delaware et al., 2009; Apel et al., 2010) instead of generating all SPL products. Even though the tool spends minutes to retrieve the list of ill-formed product configurations from TaRGeT, this still holds. The average time for generating a TaRGeT executable is around 30 s. We then would need more than 500 h to generate all products. In the case of requirements, we would need to generate over 60 thousand specifications to analyze.

For MobileMedia, it can be argued that the CK derivation process introduced problems. However, for releases 6 and 7, there is a build file with all features selected, containing almost all 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 Figs. 11 and 12.

Still on MobileMedia, some problems might have happened and gone unnoticed due to the manner in which the SPL was implemented and tested. As we discuss in Section 4.2, 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 verify, even using brute-force, if all products were correct. However, our evaluation shows that even a well-structured SPL such as TaRGeT also unveiled a problem that affected thousands of products. The safe composition analysis we perform could then be used to detect problems and prioritize fixes for products. It is also important to notice that while a single problem in release 6 of MobileMedia affects few products, a single problem in the code from release 6 of TaRGeT affects thousands.

Our approach adopts a closed-world perspective in the sense that safe-composition verification requires the FM and provided and required interfaces for the assets referenced in the CK. On the other hand, we do not necessarily rely on the assets implementations. Depending on specific asset implementation languages and associated extractors of required and provided interfaces, changes to the implementation of a given asset might not require safe-composition verification if the asset interfaces are preserved. The three extractors we use in the case studies could actually have extra functionality for managing asset interfaces in a modular way. Other forms such as guaranteeing that adding new assets associated to new features preserves safe composition are not language independent (for example, due to conflicting assets, like two aspects (Kiczales et al., 2001) used as variability mechanisms (Gacek and Anastasopoules, 2001), that introduce methods with the same signature in the same class) and therefore cannot be supported, in general, by our approach. That is why we classify our approach as family-based, instead of feature-based (Thuem et al., 2012).

Similarly to previous work that encode the constraints using propositional logic (Czarnecki and Pietroszek, 2006; Thaker et al., 2007; Batory, 2005), we limit our solution in both the domain and implementation constraints. In the domain side, we cannot use cardinality-based feature models (Czarnecki et al., 2005), for example, since the use of range intervals and attributes enable the FM semantics to generate infinite products. In the implementation side, we cannot express some kinds of constraints, such as the version number of a given component must be higher than 1.2 (Tucker et al., 2007), or the memory consumption must lie below 256 kilobytes (White et al., 2007). To handle these scenarios, one should use constraint solver techniques (Benavides et al., 2005; White et al., 2007). In practice, this extra power might not be used in some cases, as we have experienced in the TaRGeT case study, discussed in Section 4.

Finally, it could be argued that the manual effort involved to use such an approach can outweigh the benefits. Usually, if there is a systematic way of generating products from feature selections, there is already some kind of mapping between features and assets, which constitutes a CK. As we discuss, it might be the case where it is implicit or different from the CK model used in this article. For example, the Linux kernel uses a mixture of preprocessor directives and special languages (Kconfig and KBuild) for handling variability and the mapping between features and code (Lotufo et al., 2010). Based on such available information, to use our approach, we would need provided and required interfaces. Such interfaces are usually relevant elements of the languages used for implementing the SPL. In the Java and AspectJ context, we used classes and aspect names. In the requirements context, interfaces were scenario and step names, parameters and annotations. A good part of this extraction can be automatically performed (Thaker et al., 2007; Delaware et al., 2009; Apel et al., 2010; Czarnecki and Pietroszek, 2006).

However, it might be the case where there is no extraction technique available for a language, as it was the case with MSVCM requirements. Then, some effort might be needed to define and implement an extractor. Such an extractor can be also implemented with different precision levels, depending on the needs of developers. The decision about implementing extractors should depend on an effort analysis, business reasons, and the potential benefits that the safe composition analysis can bring. Given that interfaces are available, our tool can *automatically* translate the FM and CK constraints to Alloy and perform the verification, reporting the ill-formed product configurations, when there are safe composition problems.

5. Related work

Thaker et al. present techniques for verifying type safety properties of AHEAD (Batory, 2004) product lines using FMs and SAT solvers (Thaker et al., 2007). They extract properties from feature modules and verify that they hold for all SPL members. Delaware et al. formalizes this work using Lightweight Feature Java (LFJ), an extension of Lightweight Java with features (Delaware et al., 2009). They prove soundness of the underlying type system. Their focus is on inferring type checking constraints from the language used for implementing assets, while we have not focused in extracting interfaces from specific asset languages, but on verifying, in a language independent way, possibly considering manually provided semantic dependencies, interfaces provided to the CK. Thus, the quality of our verification is dependent on such information. Since our main focus is not on inferring type constraints for a specific asset language, we do not prove soundness of our formalization, as Delaware et al. does. However, our Alloy encoding provides sound and complete analysis, due to our scope being well-delimited. We could also integrate both approaches, by adapting the constraints generated to use as interfaces in our encoding.

Apel et al. propose Feature Featherweight Java (FFJ) as a type system for feature oriented programming (FOP) (Apel et al., 2008). They use this type system to check whether a given composition of features is safe, before compilation. They presented a condensed version of FFJ in Apel et al. (2010), together with a soundness and completeness proof of the language. 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. However, our approach works in a language independent way, and we have evaluated it in the context of an industrial SPL, showing that it scales to deal with thousands of products.

Apel et al. also propose the gDeep core calculus for uniform feature composition (Apel and Hutchins, 2008). It aims to be a language-independent pluggable type system, which 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. However, although we can interpret our analysis as a proof, we do not provide a full formalization of our CK model. Apel et al. also present a reference checking algorithm for feature-oriented SPLs (Apel et al., 2010). 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 number of smaller formulae that can be cached and reused (Apel et al., 2010). 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. Similarly to this article, both works focus on a language independent solution. However, we evaluate and show that our approach works in larger case studies, consisting of SPLs implemented using multiple languages.

Schaefer et al. propose Delta-oriented Programming (DOP) (Schaefer et al., 2010) as another way for implementing SPLs. Implementation uses delta modules 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, separating when such implementation should be applied and in which order, increasing reusability. They also present a compositional type system (Schaefer et al., 2011) for delta-oriented SPLs in Java using a core calculus for DOP. Constraints are inferred for each delta module. The type system is proven correct and complete with respect to the core calculus used. Similarly to what our work does for reporting ill-formed product configurations, it verifies safe composition considering the delta module constraints and an abstraction of each program variant. Our work, as mentioned, works in a language independent way. Therefore, as future work, we intend to investigate whether we could use it for verifying SPLs implemented using DOP.

Czarnecki and Pietroszek present a well-formedness verification approach for feature-based model templates (Czarnecki and Pietroszek, 2006), which consists of an FM and an annotated model expressed in some general modeling language. Annotations refer to features and resemble the use of conditional compilation directives. In a similar way as our work, they check FMs against constraints to verify that no ill-formed template instances can be produced. They also retrieve ill-formed product configurations from error messages returned by the SAT solver counterexamples. These annotations are equivalent to the feature expressions in our CK. A difference, besides their work being specific to model templates, is that while our CK information is modularized, in model templates it is scattered throughout the SPL assets. However, we could integrate their constraints with our approach, since all variability is associated to feature expressions.

Also using annotations, Kaestner et al. propose the Color Featherweight Java (CFJ) calculus (Kästner and Apel, 2008), motivated by the Colored Integrated Development Environment tool (CIDE) (Kästner et al., 2008). The mapping between features and code occurs through a disciplined form of conditional compilation. The calculus establishes type rules to ensure that only well-typed programs can be generated. They prove that — given a well-typed CFJ SPL — all possible variants are well-typed FJ programs. Code annotated with two features has the same semantics of an *and* ($\wedge$) operator. However, other types of feature expressions are not supported. Kaestner et al. further extend this study, formalizing an SPL-aware type system (Kästner et al., 2011). They discuss a solution for the alternative feature typing problem that happened with CFJ and implement checks for full Java in CIDE, conducting case studies to evaluate their approach. Kenner et al. developed TypeChef (Kenner et al., 2010), a type checker that aims at identifying errors in SPLs implemented with the C preprocessor. Comparing to our work, the main difference between all of these annotation-based works is that their CK is scattered throughout the code. In the case of SPLs dealing with both coarse and fine-grained variabilities, we could combine our approaches to address both variability levels. This could help in contexts such as in the **Photo** $\wedge$ **Music** $\wedge$ **Video** problem, discussed in Section 4.1.1.

Alferez et al. propose an approach to check consistency between features and use case scenarios that realize them (Alférez et al., 2011). In their work, instead of the textual use case scenarios we use, they work with use case and activity diagrams. They customize scenarios with the Variability Modeling Language for Requirements (VML4RE) (Alférez et al., 2009), a language that allows associating feature expressions with actions that result in model transformations, similar to the MSVCM CK. They check constraints generated from the diagrams and their associations with features against the feature model formula, likewise we do in this work. Unlike MSVCM, the order of the variants influences in the composition process, therefore they plan to research algorithms that take into account the precedence order between variants and their application. Mendonca et al. show that FMs pose no significant difficulty for SAT solvers (Mendonca et al., 2009), justifying widespread use of SAT-based systems, such as the Alloy Analyzer. In this work, we add the CK to the encoding. However, the general CK formula structure is similar to that of FMs, which fit in the constraint class that Mendonca et al. show that is easy to analyze. Therefore, our results also confirm their findings.

6. Conclusions

In this article we propose an approach to verify safe composition of CK-based SPLs. We use dependencies between assets in the form of required and provided interfaces to enable the verification (Teixeira et al., 2011). We also present tool support that checks safe composition of an SPL. If the SPL is ill-formed, the tool reports the ill-formed product configurations. We give further evidence of the generality of our approach, instantiating it in other compositional contexts, beyond Java and AspectJ, using different kinds of assets, such as use case scenarios. Besides evaluating seven releases of the MobileMedia SPL (Figueiredo et al., 2008), we investigate safe composition problems in both the code and requirements of the TaRGeT SPL (Ferreira et al., 2010). TaRGeT is implemented using Eclipse RCP plug-ins technology. When analyzing the code, we found a problem related to an incomplete refactoring that affects more than 18 thousand products. Besides the code, TaRGeT also has its use case scenarios structured using the Modeling Scenario Variability as Crosscutting Mechanisms (MSVCM) (Bonifácio and Borba, 2009) approach. When evaluating the requirements, we found problems similar to the ones found in code, such as unresolved dependencies and bad CK specification, which causes the majority of the product-specific scenarios to be considered ill-formed. Finally, we propose an alternative implementation for reporting ill-formed product configurations, which enables our approach to analyze SPLs consisting of thousands of products. Our evaluation numbers suggest that the analysis time depends on the kinds of assets used for implementing the SPL, due to the different number of interfaces that might result. We also observe that time for performing the analysis grows linearly with the number of products in the SPL.

In our earlier work, we presented a general theory for SPL refinement (Borba et al., 2012). In it, we define an SPL as a tuple formed by FM, CK, and assets, in which all products that can be generated are well-formed. In this work, we propose and implement a language independent way to check well-formedness. This is important as an initial step towards tool support for SPL refactoring. We plan to integrate our approach with a tool (Alves et al., 2008) that implements the general theory for SPL refinement (Borba et al., 2012) for checking refactorings (Borba, 2009). In this case we would check whether changes made to an SPL preserve well-formedness. This could be useful to avoid errors such as the one detailed for TaRGeT code, where the class name was changed, whereas the aspect that referenced the class remained unchanged. As future work, we intend to investigate other representations in a case study involving cardinality-based FMs and CK. Moreover, we also intend to evaluate how our solution scales compared to constraint solver techniques, when the extra power from cardinality-based settings is not used. We also intend to evaluate our approach in the context of SPLs implemented with conditional compilation. Finally, we intend to explore ways where we can optimize the reporting ill-formed product configurations phase, and also providing better guidance to the source of the safe composition problem.

Acknowledgements

We thank the anonymous referees for the useful suggestions and comments to improve this work. We would also like to thank colleagues of the Software Productivity Group (http://www.cin.ufpe.br/spg/) for helping to significantly improve this work. Márcio Ribeiro provided an idea that helped us to optimize the reporting ill-formed product configurations phase. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), (http://www.ines.org.br) funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. We also acknowledge support from CNPq grants 304470/2010-4, 480160/2011-2, and 484860/2011-9. The first author is supported by the grant MCT/CNPq/PGAEST 70/2009.

References

Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., Amaral, V., 2009. Multi-view composition language for software product line requirements. In: SLE'09, pp. 103–122.

Alférez, M., Lopez-Herrejon, R.E., Moreira, A., Amaral, V., Egyed, A., 2011. Supporting consistency checking between features and software product line use scenarios. In: ICSR'11, pp. 20–35.

Alves, V., Calheiros, F., Nepomuceno, V., Menezes, A., Soares, S., Borba, P., 2008. Flip: managing software product line extraction and reaction with aspects. In: SPLC'08, p. 354.

Apel, S., Hutchins, D., 2008. A calculus for uniform feature composition. ACM Transactions on Programming Languages and Systems 32 (5), 19:1–19:33.

Apel, S., Kästner, C., Lengauer, C., 2008. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement. In: GPCE'08, pp. 101–112.

Apel, S., Scholz, W., Lengauer, C., Kästner, C., 2010. Language-independent reference checking in software product lines. In: FOSD'10, pp. 65–71.

Apel, S., Kästner, C., Grösslinger, A., Lengauer, C., 2010. Type safety for feature-oriented product lines. Automated Software Engineering 17, 251–300.

Batory, D., Benavides, D., Ruiz-Cortes, A., 2006. Automated analysis of feature models: challenges ahead. Communications of the ACM 49 (12), 45–47.

Batory, D., 2004. Feature-oriented programming and the AHEAD tool suite. In: ICSE'04, pp. 702–703.

Batory, D., 2005. Feature models grammars, and propositional formulas. In: SPLC'05, pp. 7–20.

Benavides, D., Martín-Arroyo, P.T., Cortés, A.R., 2005. Automated reasoning on feature models. In: CAiSE'05, pp. 491–503.

Bonifácio, R., Borba, P., 2009. Modeling scenario variability as crosscutting mechanisms. In: AOSD'09, pp. 125–136.

Bonifácio, R., Teixeira, L., Borba, P., 2009. Hephaestus: A Tool for Managing Product L ine Variabilities, pp. 26–34.

Borba, P., Teixeira, L., Gheyi, R., 2012. A theory of software product line refinement. Theoretical Computer Science 455, 2–30.

Borba, P., 2009. An introduction to software product line refactoring. In: GTTSE'09, pp. 1–26.

Czarnecki, K., Antkiewicz, M., 2005. Mapping features to models: a template approach based on superimposed variants. In: GPCE'05, pp. 422–437.

Czarnecki, K., Eisenecker, U., 2000. Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional.

Czarnecki, K., Pietroszek, K., 2006. Verifying feature-based model templates against well-formedness OCL constraints. In: GPCE'06, pp. 211–220.

Czarnecki, K., Helsen, S., Eisenecker, U.W., 2005. Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice 10 (1), 7–29.

Delaware, B., Cook, W., Batory, D., 2009. Fitting the pieces together: a machine-checked model of safe composition. In: ESEC/FSE'09, pp. 243–252.

Eriksson, M., Borstler, J., Borg, K., 2005. The PLUSS approach – domain modeling with features, use cases and use case realizations. In: Proceedings of the 9th International Conference on Software Product L ines, pp. 33–44.

Ferreira, F., Neves, L., Silva, M., Borba, P., 2010. TaRGeT: a model based product line testing tool. In: CBSoft'10 – Tools Session, pp. 1–4.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F., Dantas, F., 2008. Evolving software product lines with aspects: an empirical study on design stability. In: ICSE'08, pp. 261–270.

Gacek, C., Anastasopoules, M., 2001. Implementing product line variabilities. In: SSR, pp. 109–117.

Gheyi, R., Massoni, T., Borba, P., 2006. A theory for feature models in Alloy. In: 1st Alloy Workshop, pp. 71–80.

Jackson, D., Schechter, I., Shlyakhter, I., 2000. Alcoa: the alloy constraint analyzer. In: ICSE, pp. 730–733.

Jackson, D., 2006. Software Abstractions: Logic Language and Analysis. The MIT Press, Cambridge, MA.
Kästner, C., Apel, S., 2008. Type-checking Software Product L ines – a formal approach. In: ASE'08, pp. 258–267.
Kästner, C., Apel, S., Kuhlemann, M., 2008. Granularity in software product lines. In: ICSE'08, pp. 311–320.
Kästner, C., Apel, S., Thüm, T., Saake, G., 2011. Type checking annotation-based product lines. ACM Transactions on Software Engineering and Methodology (TOSEM) 21 (3), 14:1-14:39.
Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S., 1990, November. Feature-oriented domain analysis (FODA) feasibility study. Tech. rep. Carnegie-Mellon University Software Engineering Institute.
Kenner, A., Kästner, C., Haase, S., Leich, T., 2010. TypeChef: toward type checking #ifdef variability in C. In: FOSD'10, pp. 25–32.
Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., 2001. Getting started with AspectJ. Communications of the ACM 44 (10), 59–65.
Lotufo, R., She, S., Berger, T., Czarnecki, K., Wasowski, A., 2010. Evolution of the linux kernel variability model. In: SPLC'10, pp. 136–150.
McAffer, J., Lemieux, J.-M., 2005. Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications. Addison-Wesley Professional.
Mendonca, M., Wasowski, A., Czarnecki, K., 2009. Sat-based analysis of feature models is easy. In: SPLC'09, pp. 231–240.
Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag.
Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N., 2010. Delta-oriented programming of software product lines. In: SPLC'10, pp. 77–91.
Schaefer, I., Bettini, L., Damiani, F., 2011. Compositional type-checking for delta-oriented programming. In: AOSD'11, pp. 43–56.
Teixeira, L., Borba, P., Gheyi, R., 2011. Safe composition of Configuration Knowledge-based software product lines. In: SBES'11, pp. 263–272.
Teixeira, L., Borba, P., Gheyi, R., 2012. Online appendix. http://www.cin.ufpe.br/ lmt/jss2012/
Thaker, S., Batory, D., Kitchin, D., Cook, W., 2007. Safe Composition of Product L ines. In: GPCE'07, pp. 95–104.
Thuem, T., Apel, S., Kaestner, C., Kuhlemann, M., Schaefer, I., Saake, G., 2012, April. Analysis strategies for software product lines. Tech. Rep. FIN-004-2012, University of Magdeburg, Germany.
Tucker, C., Shuffelton, D., Jhala, R., Lerner, S., 2007. OPIUM: optimal package install/uninstall manager. In: ICSE'07, pp. 178–188.
White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A., 2007. Automating product-line variant selection for mobile devices. In: SPLC'07, pp. 129–140.
Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U., 2009. VML* – a family of languages for variability management in software product lines. In: SLE'09, pp. 82–102.

Leopoldo Teixeira is a PhD student in the Informatics Center at Federal University of Pernambuco. His research interests include software product line development, evolution, and verification. He holds a MSc in Computer Science from the Federal University of Pernambuco, and is a member of the ACM. More on Teixeira can be found at http://www.cin.ufpe.br/ lmt.

Paulo Borba is Professor of Software Development at the Informatics Center of the Federal University of Pernambuco, Brazil, where he leads the Software Productivity Group. His main research interests are in the following topics and their integration: software modularity, software product lines, and refactoring.

Rohit Gheyi is an assistant professor in the Department of Computer Science at Federal University of Campina Grande. His research interests include refactorings, formal methods and software product lines. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco, and is a member of the ACM. More on Gheyi can be found at http://www.dsc.ufcg.edu.br/ rohit.