Journal of Universal Computer Science, vol. 20, no. 5 (2014), 640-665
submitted: 28/7/13, accepted: 15/2/14, appeared: 1/5/14

Consistency Checking in Early Software Product Line Specifications - The VCC Approach

Mauricio Alférez
(CITI and Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal &
INRIA, DiverSE Team, Rennes, France
mauricio.alferez@{campus.fct.unl.pt, inria.fr})

Roberto E. Lopez-Herrejón
(Institute for Systems Engineering and Automation
Johannes Kepler University, Linz, Austria
roberto.lopez@jku.at)

Ana Moreira, Vasco Amaral
(CITI and Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal
{amm,vasco.amaral}@fct.unl.pt)

Alexander Egyed
(Institute for Systems Engineering and Automation
Johannes Kepler University, Linz, Austria
alexander.egyed@jku.at)

Abstract: Software Product Line Engineering (SPLE) is a successful paradigm to produce a family of products for a specific domain. A challenge in SPLE is to check that different models used in early SPL specification do not contain inconsistent information that may be propagated and generate inconsistent products that do not conform to its requirements. This challenge is difficult to address due to the high number of possible combinations of product features and model fragments specifying those features. Variability Consistency Checking (VCC) offers automatic means to address that challenge. VCC relates information inferred from the relationships between features and from base models related to those features. Validating if all the products in an SPL satisfy user-defined consistency constraints is based on searching for a satisfying assignment of each formula generated by VCC. We validated VCC and its supporting tool on two case studies from different application domains, the results were encouraging as we did not observed significant performance penalties.

Key Words: Model-Driven Development, Variability Modeling, Verification, Model-Based Software Product Lines, Requirements Engineering, Architecture Design, Feature Modeling Analysis, Variability-Intensive Systems, Highly Configurable Systems.
Category: D.2.1, D.2.2, D.2.4, D.2.10, D.2.11, D.2.13, F.3.1, F.4.1, I.6.4, I.6.5

1 Introduction

A Software Product Line (SPL) is "a set of software–intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [Clements and Northrop, 2002]. An SPL is described in terms of *features*, defined as increments in system's functionality. Every concrete system in an SPL is seen as a *product variant* (or product) that is identified by a unique combination of features. Traditionally, a *feature model* is used to represent features and their interdependencies. Each product is then assembled from a set of common features and a selection of optional features from the feature model, what is called *feature model configuration* [Czarnecki and Eisenecker, 2000].

The meaning of the features contained in a feature model are usually described using other complimentary models, supporting the developers' focus on different processes of the SPL, such as requirements, architecture, and detailed design. However, the need for these multiple models to coexist demand for consistency among them.

In general, a system is consistent when there is logical coherence (i.e., no contradictions) between its parts. Algorithms that verify the satisfaction of consistency conditions in a system can measure consistency. These conditions can be as diverse as architectural and design guidelines, programming conventions and well-formedness rules. In the context of this paper, consistency checking refers to the verification of consistency constraints (or conditions) between a feature model and other early base models (i.e., requirements and high-level architecture models) used to describe those features.

In an inconsistent SPL, valid base models may be invalid in terms of the feature model and invalid base models may be valid in terms of the feature model. According to [Apel et al., 2010b], the latter case is especially difficult to address because it is usually detected only after generating specific product variants based on a selection of features. Due to the possible very large number of different valid features combinations and their relationships with model fragments, consistency checking for every product variant is feasible only for small SPLs [Apel et al., 2010b].

The goal of this paper is to present the Variability Consistency Checker approach (VCC) that addresses consistency checking during domain engineering for early SPL specifications, not for only a single, small product at a time. Checking if all the products in an SPL satisfy consistency constraints is based on searching for a satisfying assignment for each propositional formula generated by VCC.

Each propositional formula varies according to the type of consistency constraint and the relationships between: (1) *feature expressions* (i.e., combinations of features and logical operators) obtained from the feature model, and (2) dependencies and incompatibilities between fragments of the base models related to feature expressions. The VCC tool translates propositional formulas that are evaluated by satisfiability (SAT) solvers[1]. For consistency constraints that are not satisfied by the SPL, the VCC tool presents to the developer the features and fragments in the base models involved in the violation of the constraint. Such output is useful to take informed decisions to improve the feature model and its associated base models.

Our aim of checking consistency for an entire SPL instead of only for a single product is in line with a new generation of approaches [Kästner and Apel, 2008, Apel et al., 2010a, Delaware et al., 2009, Thaker et al., 2007, Czarnecki and Pietroszek, 2006]. The goal of those approaches is to guarantee that every valid feature selection produces a type-correct program (see Section 6–Discussion and Related Work). VCC, however, focuses on proactively checking consistency on base models. Therefore, it helps developers to complete the models through repeated cycles (iterations) and in small portions at a time (incrementally). Also, VCC considers that relationships among base models fragments can suggest constraints that should be reflected in the feature model. That characteristic is especially useful during reactive SPL development where different people create base models before the feature model.

We recognize that not all technical possible configurations of base models should be reflected in the feature model. Nevertheless, given that the knowledge expressed by domain experts in requirements and high-level architectural models is not too technical (in comparison with what is required for code), both feature model and base models used in early SPLE should be consistent (or at least, to have a tolerable small number of potential inconsistencies that can only be solved later in the development process).

The results of applying VCC to two case studies from different application domains are encouraging, given that no significant performance penalties were observed when automatically verifying a set of user-defined consistency rules.

The remaining of this paper is structured as follows. Section 2 motivates consistency checking with an example brought from part of a home automation SPL called Smart Home. Section 3 presents the VCC approach followed by a description of its tool support in Section 4. Section 5 shows the validation of VCC based on the full Smart Home and an SPL for photos manipulation in mobile devices called Mobile Photo. Section 6 discusses and relates our work with other works. Finally, Section 7 presents our concluding remarks and future work.

[1] The SAT problem is to check whether a formula is satisfiable. For example, the formula "a And Not b" is satisfiable because one can find the values a = True and b = False, which make (a And Not b) = True. In contrast, "a And Not a" is unsatisfiable. See details in http://www.satisfiability.org

2 Motivational Example

This section presents an example that illustrates inconsistency based on four kinds of models used during early SPL development: (1) feature model, (2) base models[2], (3) composition specification model, which determines how each selection of features triggers transformations of the base models, and (4) mapping model between features and base models.

The example used in this section to illustrate VCC is taken from one of the two case studies used in the European AMPLE project (`http://www.ample-project.net`). The two case studies briefly discussed in the validation section (Section 5) are Smart Home and Mobile Photo. For the Smart Home, we have chosen use cases and activity diagrams to model the requirements of the system. For the Mobile Photo, we started from a top-down analysis of architectural components. Hence, in this paper we focus on those three types of base models. However, other models (e.g. class diagrams) could be used as base models in VCC.

2.1 Creating a Feature Model and Base Models

Figure 1 (a) shows part of a feature model for one of our case studies, the Smart Home SPL [Morganho and Gomes, 2008]. Smart Home has four optional features, Automated Windows, Automated Heating, Remote Heating Ctrl and Internet to control the heater and other devices remotely. Also, it has a set of common features, such as Manual Windows, Manual Heating and In-home Screen that will be included in all the target products produced from the Smart Home SPL.

Each feature model configuration corresponds to a product variant and is defined by selecting optional features in the feature model. Figure 1 (b) shows Product-1, which has all the features, and Figure 1 (c) shows Product-2, which has all features except Automated Windows. Domain constraints such as the Requires relationship from Remote Heating Ctrl to Internet, can be added incrementally when creating base models (discussed below).

Use cases and activity diagrams provide a description of what products in the domain should do. Feature models determine which functionality can be selected when engineering new products from the SPL. Therefore, product requirements specifications consist of customized use cases and activity diagrams. The customization is guided by a composition specification discussed in Section 2.2.

For space reasons we do not show separately all the available model fragments before composition; instead, we show the resulting models for one sample

[2] Section 5 shows constraints for use case, activity and components diagrams that can be found online at http://people.rennes.inria.fr/Edward-Mauricio.Alferez_Salinas/JUCS/data.htm

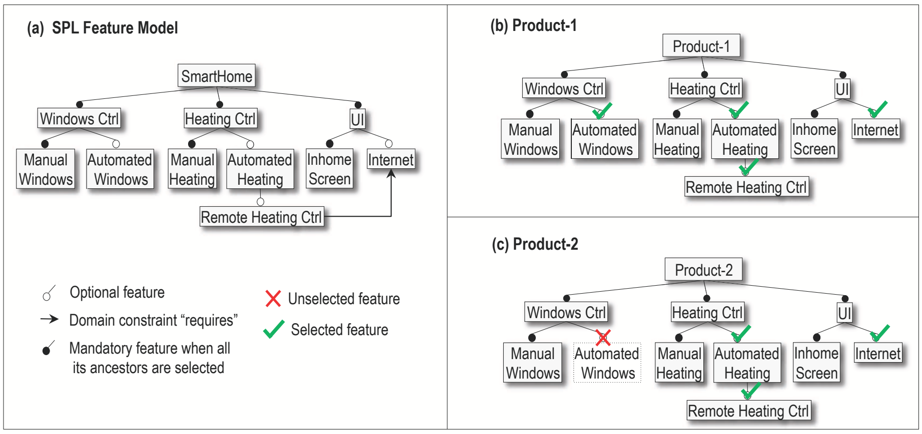


Figure 1: (a) Simplified sample of the Smart Home feature model; (b) Sample configuration that includes all features; (c) Sample configuration that excludes Automated Windows.

product. Figure 2 (a) (left- and right-hand sides) shows two of the models that are part of the base models for Product-1. The left-hand side shows an activity diagram that depicts the possible scenarios for the use case Ctrl Temp Remotely that is described in the use case diagram on the right-hand side of Figure 2 (a). The activity diagram also contains activities of use cases Open And Close Win Auto, Adjust Heater Value and Notify By Internet. Within this activity diagram it is possible to select several scenarios that correspond to different paths. Two of these scenarios are: Scenario (S1) includes reaching the in-home temperature and save energy by means of closing some windows, and Scenario (S2) uses the heater to reach the desired in-home temperature.

Figure 2 (a) (right-hand side) shows part of the use case model composed for Product-1. The Includes relationship supports the reuse of functionality in a use case diagram and is used to express that the behavior of the inclusion use case is common to other use cases. Note that Includes relationships between use cases may constrain the relationship between the features related to them. For example, the Includes relationship between the base use case Ctrl Temp Remotely that includes the use case Open And Close Win Auto may imply that feature Remote Heating Ctrl requires the feature Automated Windows.

Customization of base models such as activity diagrams and scenarios depends on the features chosen and on their relationships with the use case diagram. For example, given that in Product-2 the feature Automated Windows was not selected, the Win Actuator actor in the use case diagram as well as the activity partition (or swim-lane) related to Windows Actuator will not appear in any diagram. Therefore, scenarios such as Scenario S1, are not realizable due to lack of window actuators. This and other constraints will be discussed in Section 3.

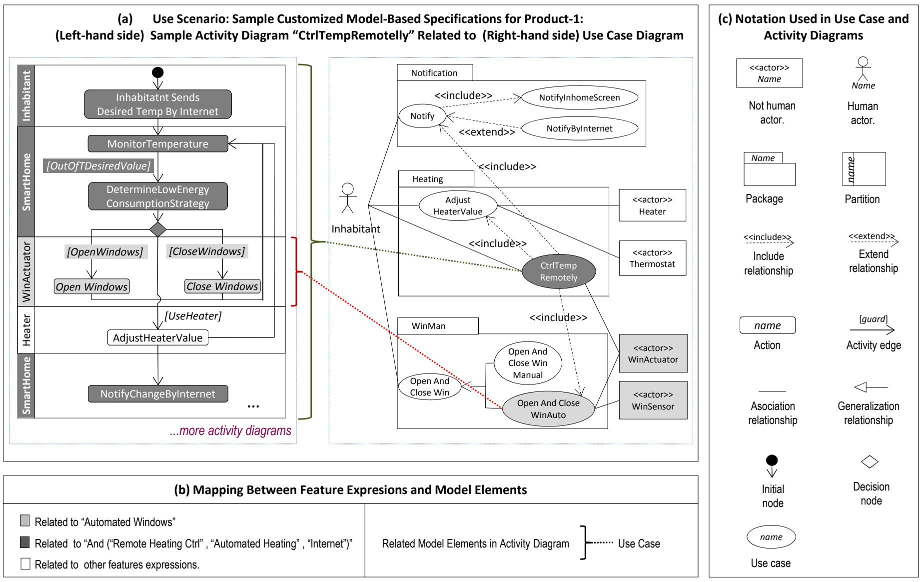


Figure 2: (a) Sample customized base models for Product-1 (i.e., the resulting models after composition); (b) Mapping between feature expressions and model fragments; (c) Notation used in use case and activity diagrams in (a).

2.2 Creating Composition and Mapping Models

A way to produce specific product models is through a composition. We show composition specifications of base models to illustrate the results of inconsistencies when deriving product variants and also as an artifact that is useful to map features to base models. We use VML4RE [Zschaler et al., 2009, Alférez et al., 2009] as an example of a composition specification language because it allows referencing model elements easily and is very expressive.

Figure 3 illustrates part of the composition specification for the Smart Home. In VML4RE, actions, that wrap a set of model transformations, are related to combinations of features called *feature expression* and written as a logic expression. Feature expressions can be Atomic, representing single features (Line 1 of Figure 3) or Compound, containing logic operators such as And, Not and Or (Line 7).

Feature expressions evaluation works as follows: if in a feature model configuration, a feature is selected to be part of a product, that feature evaluates to True and if the feature is not selected it evaluates to False. Thus, a feature expression can be evaluated to True or False taking into account the Boolean value of each feature in the feature expression. If a feature expression evaluates to True, its corresponding actions will be processed and applied to another model (or base model). Otherwise, if the feature expression evaluates to False, the next

```
1.   Variant { name : "A-W" for : "Automated Windows"
2.     + UseCase : "OpenAndCloseWinAuto" in Package : "WinMan"
3.     + Partition : "WinActuator" in ActDiagram :"CtrlTempAuto"
4.     ...
5.   }
6.   Variant { name : "R-H"
7.      for : And ("Remote Heating Ctrl","Automated Heating","Internet")
8.       + UseCase : "CtrlTempRemotely" in Package : "Heating"
9.       + UseCase : "CalcEnergyConsumption" in Package : "Heating"
10.      Includes from UseCase : "CtrlTempRemotely"   to UseCase(s) :
11.            "NotifyByInternet" and "OpenAndCloseWinAuto" and "AdjustHeaterValue"
12.      ...
13.  }
    ...more variants
```

Figure 3: Composition specification of variants blocks A-W and R-H

feature expressions will be read and evaluated until the end of the composition specification.

In our example, if Automated Heating, Remote Heating Ctrl and Internet features are selected in a product configuration, the feature expression associated to the variant block named R-H (i.e., the compound feature expression: And ("Remote Heating Ctrl","Automated Heating","Internet")) will be evaluated to True. The consequence is that the actions inside the R-H variant block (Figure 3, Lines 6-13) will be processed and applied to a base model. For example, the Ctrl Temp Remotely use case will be inserted into the package Heating and then it will be related to other use cases using Includes and Extends relationships. For simplicity, we omitted some of the actions, such as the insertion of actors Win Sensor and Win Actuator, and some activity partitions, such as Heater.

Figure 2 (a) shows model fragments, such as actors and use cases, related to the feature expressions shown in Figure 3. Figure 2 keeps composition models and mapping model separated, as they result from different processes. Nonetheless, it is possible to parse the composition specification to generate the mapping between feature expressions and base models. Therefore, for example, we relate the feature expression of A-W, Automated Windows, to Open And Close Win Auto. To facilitate the visualization of such relationships, we assign different gray tones to the model fragments according to the feature expressions they are related to (see mapping in Figure 2 (b)). Also, note that specific model fragments could be related to more than one variant block; this may be considered as a M-to-N (with M, N >= 1) mapping between variant blocks (and its feature expressions) and model fragments (not illustrated in Figure 2).

2.3 Consistency Checking Motivation

Consistency checking aims at ensuring that requirements and constraints related to a feature model and base models are consistent between them. We analyze two requirements in the Smart Home example. Requirement-1, which is inferred from the feature model and the feature expressions in the mapping model, and Requirement-2 which is inferred from the base models:

– Requirement-1: Only one, none or both R-H and A-W can be included in a product. This information is inferred from the feature model and mapping model because all the features in the feature expression of the R-H variant block are optional and not exclusive between them (i.e., Remote Heating Ctrl, Automated Heating and Internet are optional features), and the only feature in the feature expression of the variant block A-W is also optional (i.e., the Automated Windows feature is optional).

– Requirement-2: If the use case Ctrl Temp Remotely is provided in a product, then the use case Open And Close Win Auto and its related steps in the activity diagram must be provided too in order to support all the intended use scenarios. This is implicit on the Includes relationship from the use case Ctrl Temp Remotely to Open And Close Win Auto in the use case diagram in Figure 2 (a) (right-hand side), and also because of the control flow between Determine Low Energy Consumption Strategy and Open Windows / Close Windows in Figure 2 (a) (left-hand side).

In the particular case of the Smart Home use scenarios, an inconsistency can be detected: there is at least one product that cannot satisfy Requirement-1 and Requirement-2. Let's analyze how both requirements are satisfied (or not) in each product.

While Requirement-1 is satisfied by all the product configurations, therefore satisfied by Product-1 and Product-2 (since they have the same feature model and mapping models), Requirement-2 is satisfied by Product-1 only as its use cases and activities supported all the required use scenarios for this product. For example, given that the base use case Ctrl Temp Remotely was provided in Product-1, the use cases related to it through an Includes relationship, for example Open And Close Win Auto, are also present in the model. The Includes relationship supports the reuse of functionality in use case diagrams in which one use case (the base use case) requires the functionality of another use case (the inclusion use case). Therefore, all possible use scenarios related to Ctrl Temp Remotely are supported only when its inclusion use cases are included.

Requirement-2 is not satisfied by Product-2 because its feature configuration (shown in Figure 1 (c)) does not include the Automated Windows feature. Therefore, the feature expression of variant block A-W (Figure 3, Line 1) evaluates to False and the actions inside its variant block are not processed, for example, the inclusion of the use case Open And Close Win Auto. The result is that the functionality provided by Open And Close Win Auto will not be present in the requirements of Product-2 and therefore it will not be taken into account in later stages of its development process, thus given no support for the scenarios related to Ctrl Temp Remotely.

One solution to solve the inconsistency for our example is to guarantee the presence of the feature Automated Windows when Automatic Heating or Remote Heating Ctrl are selected, in every possible feature model configurations. This can be achieved by creating feature model constraints such as "Requires" (i.e., the selection of one feature requires the selection of another feature) from Remote Heating Ctrl to Automated Windows or to establish that Automated Windows is a mandatory feature. However, as the number of possible feature combinations may grow exponentially with the number of features of an SPL, it is unfeasible to manually check the consistency of all products, one by one. The following section discusses VCC, our solution for consistency checking during domain engineering.

3 The VCC Approach

This section explains the automation of VCC, through the functions Vcc and Main shown in Algorithm 1. The creation of the models to be analyzed (Lines 1-4) was exemplified in Section 2. Main is composed of a loop (Lines 16-24) where new iterations take place if the report generated by function Vcc (Line 15) shows the existence of inconsistencies (Line 16), or the developer decides to modify the input models after reported inconsistencies (Lines 18-19).

In VCC, Feature Model Constraints (FC) and Base Models Constraints (BC) must be consistent among them. FC (Line 8) is obtained from the feature model (addressed in Section 3.1) and BC (Line 9) is obtained from the relationships between model elements in the base models (addressed in Section 3.2).

We employ propositional logic to express and relate FC and BC. Our goal is to guarantee that FC meets BC (i.e., $BC \rightarrow FC$) and that BC meets FC (i.e., $FC \rightarrow BC$). To check consistency (Line 10), Equation 1 should not be satisfiable for each and every consistency rule instance that we want to check. If it is satisfied, it means that feature model constraints do not meet base models constraints (i.e., $\neg(BC \rightarrow FC)$ is satisfiable), or that the base models constraints do not meet feature model constraints (i.e., $\neg(FC \rightarrow \mathrm{BC}))$ is satisfiable).

Algorithm 1. Overview of the VCC Approach

```
1  input: B = Base Models
2         F = Feature Model
3         C = Composition Model
4         M = Mapping Model
5  output: R = Report
6
7  Report  Vcc (B, F, C, M){
8    FC ← Obtain FC (F)
9    BC ← Obtain BC (B, C, M)
10   R ← Check (FC , BC)
11   return R
12 }
13
14 Main (B, F, C, M) {
15   R ← Vcc (B, F, C, M)
16   while (not R.splConsistent ()){
17     R.showInconsistencies ()
18     if (developerAnswer equals
19        "I want more iterations")
20        /* developers modifies one or
21           more of the input models */
22        R ← Vcc (B ', F ', C ', M ')
23     else exit
24   }
25   print ("SPL Consistent")
26 }
```

$$\neg(BC \rightarrow FC) \bigvee \neg(FC \rightarrow BC) \quad (1)$$

Section 3.3 explains how VCC and a SAT solver can identify the inconsistent features and the model fragments (Line 13). This information helps the user making informed decisions to modify the initial models (in Line 20, the symbol ' indicates a modified version of the model).

Section 2 showed that at least one product (i.e., Product-2) from the Smart Home SPL is inconsistent. In that case, constraints that can be inferred from use scenarios, such as the "Includes" between use cases and related control flows, and actions in activity diagrams, imply the application of feature model constraints (i.e., $BC \rightarrow FC$). In our example, the particular consistency condition that guarantees that $\neg(BC \rightarrow FC)$ is not satisfiable, is defined by the rule:

- Required Inclusion Use Case: at least one variant block defining an inclusion use case must be selected in every feature configuration containing the variant block that introduces a base use case linked to the inclusion use case.

The following subsections discuss the VCC approach in details. In particular, they explain how to derive FC (Section 3.1) and BC (Section 3.2), as well as how the Check function uses FC and BC to generate a report (Section 3.3).

3.1 Obtaining Constraints between Features (FC)

To express FC it is necessary to translate the feature model to some type of logic. That idea was first proposed by [Mannion, 2002] and [Batory, 2005]. The results of their work are summarized in Table 1, where a mapping between feature model elements and propositional logic is presented. The next three steps summarize the process:

1. Each feature maps to a variable of the propositional formula;

2. Each relationship (e.g., requires, optional) is mapped onto one or more formulas;

3. The resulting formula is the conjunction of all the resulting formulas of Step 2, plus an additional constraint assigning True to the variable that represents the root feature (e.g., $SmartHome \leftrightarrow True$).

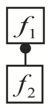
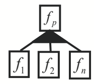
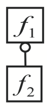
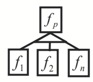
Mandatory  $f_1 \leftrightarrow f_2$	Requires  $f_a \rightarrow f_b$	Or  $f_p \leftrightarrow (f_1 \vee f_2 \vee \ldots \vee f_n)$
Optional  $f_2 \rightarrow f_1$	Excludes  $\neg(f_a \wedge f_b)$	XOr  $(f_1 \leftrightarrow (\neg f_2 \wedge \ldots \wedge \neg f_n \wedge f_p)) \wedge$ $(f_2 \leftrightarrow (\neg f_1 \wedge \ldots \wedge \neg f_n \wedge f_p)) \wedge$ $(f_n \leftrightarrow (\neg f_1 \wedge \neg f_2 \wedge \ldots \wedge \neg f_{n-1} \wedge f_p))$

Table 1: Mapping from feature model to propositional logic (based on [Benavides et al., 2010])

Equation 2 shows an example for the Heating Ctrl branch, the most complex branch in Figure 1 (a), and relates directly with the example "Required Inclusion Use Case" rule. The translation obtained in the first line of Equation 2 means that all products unconditionally must contain the root feature Smart Home. Line 2 means that Heating Ctrl must be included in all the products, since it is a mandatory feature. Line 3 means that Manual Heating is included in all the products that include their parent feature (i.e., Heating Ctrl), in contrast to Automated Heating and Remote Heating Ctrl (Lines 3-4) that may, or may not, be included when their respective parents Heating Ctrl and Automated Heating are included in a product. Line 5 means that Remote Heating Ctrl requires the Internet feature.

$$\begin{array}{l}
1.\ (SmartHome \leftrightarrow True) \wedge \\
2.\ (SmartHome \leftrightarrow HeatingCtrl) \wedge \\
3.\ (HeatingCtrl \leftrightarrow ManualHeating) \wedge \\
\quad (AutomatedHeating \rightarrow HeatingCtrl) \wedge \\
4.\ (RemoteHeatingCtrl \rightarrow AutomatedHeating) \wedge \\
5.\ (RemoteHeatingCtrl \rightarrow Internet)
\end{array} \tag{2}$$

3.2 Obtaining Base Models Constraints (BC)

To derive BC it is necessary to analyze the relationships among model elements in base models. This section explains how to obtain such relationships and how to express them in terms of feature expressions (propositional logic).

3.2.1 Obtaining Relationships between Model Elements

There are two non-mutually exclusive ways to obtain relationships between model elements:

1. Based on the actions of the composition specification. All the actions that suggest the pre-existence of model elements are good candidates to be analyzed. This is the case presented in Figure 4 (b), where the use case Ctrl Temp Remotely includes the behavior represented by Open And Close Win Auto. Therefore, one may create a requirer-required relationship among model elements. The requirer is Ctrl Temp Remotely and the required model element is Open And Close Win Auto. (More types of relationships are shown in the Validation, Section 5.)

   Figure 4 shows an example of the relationships between model elements and their metaclasses based on a line of a composition specification written using VML4RE (Figure 4 (c)). ConnectByIncludes, in Figure 4 (a), is a metaclass in VML4RE that represents the action (Figure 4 (c)) that links a use case to one or more use cases using the Include relationship, as is shown in Figure 4 (b). ConnectByIncludes is one of the actions that specializes the action Connect, and UseCase is a type of ModelElement referenced by ConnectByIncludes.

2. Based on the base models. This is probably the best way to obtain relationships when there is explicit information about the elements provided, required or incompatible. This is the case for UML component diagrams where each component usually declares explicitly a list of provided and required interfaces. It is straightforward to parse a base model and to create a list of relationships.

3.2.2 Expressing BC

Base models' constraints (BC) act as consistency rules describing the relationships that must hold among different model elements. There are two basic types of BC that we classify according to the type of feature model constraint that they relate with: (1) constraints that imply an Excludes relationship between features, and (2) constraints that imply a Requires relationship between features.

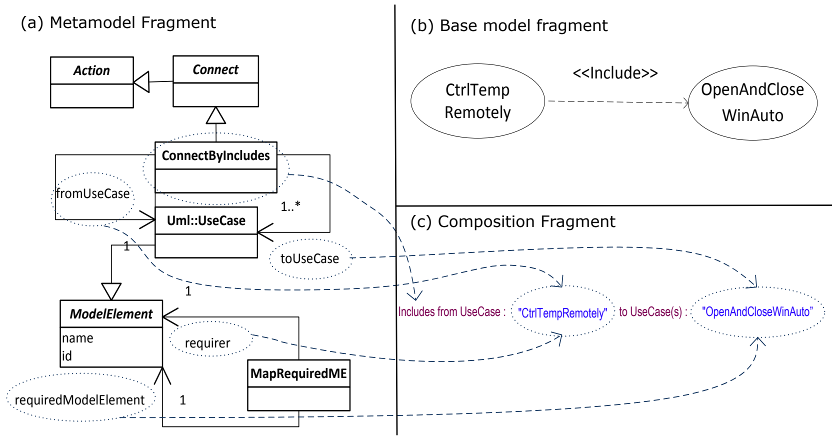


Figure 4: (a) Set of metaclasses that support the dependencies for the ConnectByIncludes action; (b) Exemplar model fragment related to an Includes relationship between use cases; (c) Composition model

Given that composition of models is not determined by individual feature selections, the BC constraints should be studied based on variant blocks and their associated feature expressions. This section shows the equations that express each type of BC. In the definition of the equations, the function *FE* (Feature Expression) receives as input a variant block name and returns its corresponding feature expression. Thus, we can express BC in terms of feature expressions and not in terms of variants blocks names. Therefore, both FC and BC use features as name of variables.

1. Excludes Relationship: Let *Var* be a variant block that defines a model element e. Another variant block $VarExc_i$ conflicts with *Var* if $VarExc_i$ defines a model element c which cannot be present in the same base model of a product where element e is. Due to the incompatibility between the elements e and c, if variant block *Var* is selected (i.e., its feature expression evaluates to true) then variant block $VarExc_i$ should not be included in the same product configuration. This is denoted in Equation 3, where n represents the total number of conflicting variants blocks:

$$BC_{Exc} \equiv FE\,(Var) \rightarrow \neg \bigvee_{i=1}^{n} FE\,(VarExc_i) \tag{3}$$

2. Requires Relationship: Let *Var* be a variant block that refers to a model element e defined by another variant. To be consistent, the model-based specifications of a product that includes *Var* must also include at least one other variant block $VarReq_i$ (required variant) where element e is defined. This is denoted in Equation 4, where n represents the number of required variants:

$$BC_{Req} \equiv FE\,(Var) \rightarrow \bigvee_{i=1}^{n} FE\,(VarReq_i) \tag{4}$$

Equation 4 evaluates to False when any action in variant *Var* requires an element or set of *required elements* that are not composed, for example. This happens because none of the corresponding variants blocks $VarReq_i$ that introduce the *required elements* was selected in the product configuration. Also, Equation 3 evaluates to False when variant *Var* defines an element or set of elements that are introduced in the base models that also contain conflicting elements defined by other variant(s) $VarExc_i$.

The rule Required Inclusion Use Case, mentioned at the beginning of this section, is an example of BC_{Req}. An instance of this constraint is found in our motivational example related to the use scenario of Ctrl Temp Remotely. For example, given that variant Var = R-H is selected (i.e., a product with Remote Heating Cntrl, Automated Heating and Internet features) and it is related to the base use case Ctrl Temp Remotely, we want to guarantee that there is at least one variant (e.g., $VarReq_i$ = A-W) related to the inclusion use case Open And Close Win Auto (i.e., model element e = use case Open And Close Win Auto) and that its feature expression evaluates to True in all possible feature model configurations. In this way, we guarantee the presence of the functionality required by Ctrl Temp Remotely, such as to include a Windows Actuator that regulates the temperature opening and closing windows. Hence, with that information, it is possible to create the base models' constraint $BC_{Req} \equiv$ R-H →A-W. We can get a constraint instance replacing the variants by their corresponding feature expressions using the function *FE* to obtain Equation 5:

$$\begin{gathered} BC_{Req} \equiv FE(R-H) \rightarrow FE(A-W) \equiv \\ (\,RemoteHeatingCtrl \wedge AutomatedHeating \wedge \\ Internet\,) \rightarrow (\,AutomatedWindows\,) \end{gathered} \tag{5}$$

3.3 Check Consistency

The implementation of the Check consistency function shown in Algorithm 1 (Line 10) depends on the kind of constraint created in the previous section. If we replace BC_{Req} of Equation 4 in Equation 1 and perform some logic manipulation, VCC obtains the expression in Equation 6 (for a step by step process see the Appendix section). The simplification steps shown in the Appendix are not considered as part of VCC. In fact, the formulas used by VCC are the ones that we show at the end of equations 6 and 7. We provided those simplification steps just to increase the confidence about the way in which these equations were built.

$$
\begin{gathered}
FC\, and\, BC_{Req}\, replaced\, in\, Equation1: \\
\neg\left(BC_{Req} \rightarrow FC\right) \bigvee \neg\left(FC \rightarrow BC_{Req}\right) \\
\equiv \left(\left(\neg FE\left(Var\right) \bigvee_{i=1}^{n} FE\left(VarReq_i\right)\right) \wedge \neg FC\right) \\
\bigvee \left(FC \wedge FE\left(Var\right) \wedge \bigwedge_{i=1}^{n} \neg FE\left(VarReq_i\right)\right)
\end{gathered} \tag{6}
$$

Similarly to Equation 6, if we replace BC_{Exc} of Equation 3 in Equation 1, and perform some logic manipulation, we obtain the expression in Equation 7.

$$
\begin{gathered}
FC\, and\, BC_{Exc}\, replaced\, in\, Equation1: \\
\neg\left(BC_{Exc} \rightarrow FC\right) \bigvee \neg\left(FC \rightarrow RC_{Exc}\right) \\
\equiv \left(\left(\neg FE\left(Var\right) \vee \bigwedge_{i=1}^{n} \neg FE\left(VarExc_i\right)\right) \wedge \neg FC\right) \\
\bigvee \left(FC \wedge FE\left(Var\right) \wedge \bigvee_{i=1}^{n} FE\left(VarExc_i\right)\right)
\end{gathered} \tag{7}
$$

The input for satisfiability checking are expressions written according to Equations 6 and 7. VCC has to check satisfiability for all those expressions to determine if an SPL is consistent. However, VCC does that process by parts, it starts by expressions of type BC_{Req} and then by expressions of type BC_{Exc}. Each expression when instantiated with concrete values is called *consistency rule instance*. The concrete values used for instantiating each expression are:

- The specific domain constraints *(FC)* of the SPL produced according to Table 1. For our example, we show part of the FC in Equation 2;
- The feature expressions related to the variants Var and either the set of its required variants $VarReq_i$ or the set of its conflicting variants $VarExc_i$. In our example, "Required Inclusion Use Case" is a rule of type BC_{Req}, thus (as shown previously by Equation 5), Var is instantiated by *R-H* and $VarReq_1$ is instantiated by *A-W*. Then, *FE*(*Var*) returns *RemoteHeatingCtrl* $\wedge$ $AutomatedHeating$ and *FE*(*A-W*) returns $AutomatedWindows$, which are the values used in the instantiation.

Each part of the disjunction in Equation 6 and Equation 7 is evaluated using different calls to the SAT solver to identify the part of the disjunction that evaluates to True (i.e., it is inconsistent). Therefore, we consider that the feature model and its base models are consistent only when all the parts of the disjunction evaluate to False.

The possible results generated by a satisfability (SAT) checker for each consistency rule instance can be True (satisfiable) or False (not satisfiable). If False is obtained for all the consistency rule instances, we know that the SPL is consistent because there are no inconsistencies between the relationships and dependencies (e.g., excludes, optional, mandatory, requires) between features depicted in the SPL feature model, and their base models. If True is obtained, our tool, based on the mapping between feature expressions and model elements in the base models, shows a list of the feature expressions and the model elements related to each inconsistency (Algorithm 1, Line 17).

Taking the example of the Smart Home feature model depicted in Figure 1, the result of the SAT solver for our particular instance of the rule "Required Inclusion Use Case" (of type BC_{Req}) is that it is satisfiable (i.e., it evaluates to True). This means that there is an inconsistency between the features and use scenarios. An example of the type of message generated by our tool is:

"...Inconsistent [use case diagram] [Ctrl Temp Remotely] and feature(s) in feature expression(s) of variant block(s) [A-W], [R-H]. The Action: [Includes from UseCase: "Ctrl Temp Remotely" to Use Case(s) "Open And Close Win Auto"] implies a [Requires] relationship between variant block [R-H] and [required] variant block(s) [A-W] that is not consistent with the feature model...".

Based on this information, developers may consider to modify the models and start another iteration of the process until obtaining a consistent SPL (Algorithm 1, Line 25).

4 Tool Support

Tools for consistency checking can be highly effective for detecting errors in variability modeling. Such tools can find errors people miss, but they can also alleviate developers from the tedious and error-prone task of checking feature models and their base model for consistency. A VCC tool prototype supporting the approach described in Sections 2 and 3 is available online[2]. This tool offers the functionalities described next.

Create or Modify Base Models. We used the Papyrus (`www.eclipse.org/papyrus`) open-source editor to create use case, activity, and component diagrams.

Create or Modify Feature Model. We use the SPLOT editor and parser (`www.splot-research.org`) that allows to share and edit our models collaboratively via Web and write arbitrary cross-tree constraints between features, and generate an initial conjunctive normal form (CNF) formula that represents the feature model. CNF is the default format read by SAT solvers.

Create or Modify Composition Model. We use VML4RE and a lightweight version of composition model editor for architecture, called lightVC (Lightweight Variability Composer). Those editors were created using EMFText (`http://www.emftext.org/`) which provided the software infrastructure to derive a concrete syntax and plug-ins based on the metamodel of each language.

Obtain Mapping between Feature Expressions and Base Models. The VCC tool parses the composition model to find the feature expression of each variant block and the model elements that each one introduces into the use cases, activity and component diagrams.

Obtain Feature Model Constraints (FC). The VCC tool translates the clauses generated by SPLOT to an appropriate CNF form readable by SAT solvers that expresses the mapping shown in Table 1.

Obtain Base Models Constraints (BC). For requirements models, such as use case and activity diagrams, the VCC tool obtains an initial list of required and provided elements based on the composition and base models. For architectural models, the VCC tool parses the component diagram to obtain the set of provided and required interfaces of each component.

Derive Base Models Constrains. Given the dependencies between model elements and their relationships with variants, the VCC tool deduces the base models constrains as described in Section 3.2. For VML4RE and lightVC it is necessary to translate to CNF the feature expressions written as propositional formulas in prefix notation.

Check Consistency. The VCC tool generates the formulas that will be passed to the SAT solver, according to the patterns shown in Section 3.3 and based on the constraints obtained from the feature model and base models. The VCC tool employs PicoSat (`http://fmv.jku.at/picosat/`) as SAT solver to determine the satisfiability of each formula.

5 Validation

The goal of our validation is to determine if VCC can *automatically check consistency between features and different kinds of base models used in early SPL development in an efficient way.* We considered that VCC is efficient if it can show results for all the consistency rule instances (of the set of rules that we defined in this section), in the order of seconds instead of minutes, hours or days.

5.1 Setup

The variables that we consider are mostly quantitative, for example, number of rule instances checked, rules, features and product variants. We validated VCC with two case studies of different sizes which are available online[2]. The first case study was the complete Smart Home SPL from which we used a small part as example in the first sections of this paper. The second case study is Mobile Photo [Young and Young, 2005], an SPL for applications that manipulates photos on mobile devices, such as mobile phones. These case studies were selected because:

1. They have sufficient complexity to validate VCC. In particular, Mobile Phone has 16 product variants and the second is Smart Home which has one Billon product variants according to SPLOT feature model analyzer (`http:`

#	Rule Description	Relationship	#	Rule Description	Relationship
1	Required inclusion *Use Case* when *base Use Case* included	Inclusion	7	Required *required Interfaces* when *Component* is included	Usage / Realization
2	Required *Package* when any of its *Use Cases* are included	Containment	8	Excluded base *Use Case* when inclusion *Use Case* is excluded	Inclusion
3	Req. *Package* when any of its contained *Packages* is included	Containment	9	Excluded *Use Case* when its *Package* is excluded	Containment
4	Req. *Use Case* when any of its children *Use Cases* are included	Generalization	10	Excluded provided *Interfaces* when its *Component* is excluded	Usage / Realization
5	Req. *Actor* when at least one of its children *Actors* are included	Generalization	11	Excluded *Opaque* Actions when its *Activity* is excluded	Containment
6	Req. *Activity* if at least one of its *Opaque Actions* are included	Containment			

Table 2: Summary of the rules implemented as base model constraints

`//www.splot-research.org`). These numbers of variants show that manual verification in the case of Mobile Phone can be very time consuming or almost impossible to finish in the case of Smart Home.

2. They come from different application domains while still can be understood by readers in general given their everyday life utility.

3. They use different kinds of base models: use scenarios for the Smart Home and component models for the Mobile Photo.

4. The chosen case studies were selected by the community of a successfully evaluated European AMPLE (`http://www.ample-project.net`) project involving two well-known major players in the industry (SAP and Siemens). It was considered by both evaluation and collaboration (involving different reputed research institutions) experts to have industry strength and complexity enough to be used as a target SPL application, therefore we considered it to be a good benchmark to validate our technique.

For the experiment, we defined feature models for each case study, base models for each one, and the mapping between model elements and variants using VML4RE and lightVC. Next, we ran the VCC tool and analyzed the results. Table 2 summarizes the rules we used in this study. We used 7 rules of type Requires and 4 of type Excludes, considering 5 kinds of relationships between model elements.

– Rules that consider Includes relationships: an include relationship, in which one use case includes the functionality of another use case, supports the reuse of functionality in use case diagrams. In VCC this kind of rule suggest including in a product the *inclusion* model element if the *base* model element is included.

- Rules that consider Containment relationships: the contained model element is the one that requires a container. In UML a common container is the package, but others exist, such as Activity that contains model elements represented in activity diagrams. In VCC this kind of rule suggest including in a product the container model element if the contained model element is included.

- Rules that consider the Usage and Realization relationships: usage relationship is a type of dependency relationship in which one model element (the client) requires another model element (the supplier) for full implementation or operation. In this experiment we consider Usage and Interface Realization relationships in component diagrams employing components as clients, and interfaces as suppliers. In VCC this kind of rule suggests including in a product the supplier model element if the client model element is included.

- Rules that consider the Generalization relationships: Generalization relationships are used in class, component, deployment, and use case diagrams to indicate that the child receives all the attributes, operations, and relationships that are defined in the parent. In our experiment we considered generalization relationships between Actors and between Use Cases. In VCC this kind of rule suggests including in a product the parent model element if the child model element is included.

Each consistency rule instance requires separated calls to the SAT solver. If VCC shows that it is satisfiable, it means that there is an inconsistency. VCC provides information about the concrete variant, feature expression, action in the variant block, and model elements related to the rule instance.

5.2 Results

Table 2 summarizes our evaluation for both cases studies. Smart Home has 60 features and comprises significant aspects of modern home automation domain such as security, HVAC (Heating, Ventilating, and Air Conditioning), illumination control, fire control and multiple user interfaces. Mobile Photo has 14 features and less variety and number of model elements than Smart Home. It has interconnected components, some acting as controllers for albums and photos, and others implementing specific operations (e.g., such as view, create, delete, edit labels, and manage data access). According to the SPLOT feature models analyzer, the Smart Home feature model allows the generation of one billion

Case Studies	Smart Home	Mobile Photo
Features	60	14
Main model elements	36 Use Cases 12 Activities 14 Packages	12 Comp. 15 Interfaces
Variant blocks	28	7
Rules	10	2
Rules instances checked	71	10
Product variants (NPV)	One billion	16
Time for checking	830	694
Avg. Rules utilization	7,1	5
Number of inconsistencies (NI)	9	3

Table 3: Evaluation results

product configurations and sixteen are possible for Mobile Photo. The number of possible combinations of features, variants and model elements makes this task time consuming and error prone if performed manually. The VCC approach and tool produces the results in milliseconds when run on an Intel Core-Duo i5 at 2.4 GHz. Given that feature models and constraints are mapped to clauses in VCC, the performance and scalability of our approach are proportional to the efficiency of the SAT solvers which are able to handle large number of clauses in industrial applications (`http://www.satisfiability.org/`).

In our experiments, we found 81 rule instances that covered the 12 rules created with an utilization average of 7.1 times each type of rule for the Smart Home and 5 times for the Mobile Photo. These numbers shows that the proposed rules were adequate to the types of models employed.

We found 9 inconsistencies in the case of the Smart Home and 3 for the Mobile Media. We do not speculate or try to predict the correlation between the number of inconsistencies (NI) and the number of product variants (NPV) as each inconsistency varies according to its effects, and the number and type of product variants affected.

5.3 Threats to Validity

We identified different factors that could be seen as threats to the validity of our work: (1) Our previous experience with the case studies, (2) The type and size of the case studies, and (3) the type of models that we employed. We explain the way in which we addressed these threats.

Previous experience. We avoided using any consistency rules specific to the domain of each case study to avoid being biased to favor the creation of particular patterns in the models based on our previous knowledge of the domain. Also, we employed a general-purpose modeling language such as UML to avoid complex explanations about domain-specific consistency rules and modeling languages well understood only by domain experts.

Type and size of case studies. It was important for us to show that VCC scales well and therefore, we found case studies of a relevant size and variability. Both case studies have variability rich feature models as they include all types of variability described in Table 1 and allowed to produce billions of products as happens in the Smart Home. The numbers that characterize the size of each case study are summarized in Table 1 to show that manual consistency checking is not an option; an approach and tool support such VCC were very necessary.

Type of models. We chose feature diagrams to model variability because (1) the case studies provided no other alternative models, and (2) this technique is widely used to specify variability, not only by researchers but also by industry (e.g. Gears, pure::variants, Linux Kconfig and eCos). Although this choice could raise a question about languages that use different variability models, we are focusing on "features" and so "feature" models emerged naturally to specify features and their relationships. Additionally, alternative specification diagrams for variability are not representative of any group of techniques.

Also, VCC does not require or depend on a specific composition model such as VML. VML models were used simply as examples from which we mined some information required to check consistency (e.g., the mapping between features expressions and model elements). Taking into account that people interested in VCC may be interested in other mapping techniques than VML, we described and implemented a way to create a mapping in Section 3.2.1 without VML. This way consists on a table-like structure with two columns "feature expression" and "model elements".

6 Discussion and Related Work

An issue in the development of SPLs is the lack of efficient approaches for consistency checking among models. In model-driven development this becomes an important issue as software is built by means of a chain of transformations. This can start from assets such as requirements specification models, to code-based assets that typically depend on a particular implementation technology. In this setting, the quality of the final code of target products depends mostly on (1) the transformations, (2) the source models of each transformation and (3) the information added after each transformation. Therefore, to create constraints not only helps stakeholders to understand the intended products, but also to obtain good quality source models that aim to derive good quality code.

Assumptions and Limitations of VCC. There are some assumptions in VCC that may limit its application for all kinds of SPLs: (1) VCC works for model-based SPLs, therefore, it supposes that there are feature models mapped to base models, (2) VCC assumes that each model was created by people that really understand the meaning of their models and how to design them, otherwise, the information inferred from each model when compared with each other would produce consistent wrong models whose errors may be propagated and produce wrong product variants, and (3) VCC assumes the use of traditional but recurring representation of feature models, thus, new kinds of feature models or other models to model variability must be translated to propositional formulas before being integrated to VCC.

Importance of Using SAT Solvers in Early Verification. We explore the use of SAT solvers and propositional logic to support consistency checking between a feature model and its corresponding base models. Usually, SAT solvers and propositional formulas are used much later in the development to represent dependencies between software assets. The result of this work showed that they can be used much earlier and therefore some inconsistencies do not have to be left until later to be detected. The use of these methods is almost transparent for SPL developers as SAT solvers and the processing of propositional logic are implemented by libraries used internally by VCC.

Feature Modules Safe Composition. Our work is related to previous work on type systems for SPL [Apel et al., 2010a, Delaware et al., 2009, Kästner and Apel, 2008, Thaker et al., 2007, Apel et al., 2010b]. Most of these works are based on feature-oriented programming where a feature is implemented by a code unit called feature module. When composed to a base program, a feature module introduces new structures, such as classes and methods, and refines existing ones. A program that is decomposed into features is called therefore, a feature-oriented program.

Thaker *et al.* address consistency checking as a Safe Composition problem for feature-oriented programs [Thaker et al., 2007]. Safe Composition guarantees that all programs in a SPL are type safe, i.e., absent of references to undefined elements such as classes, methods, and variables. They show how safe composition properties can be verified for AHEAD SPL [Batory, 2004].

VCC addresses more than safe composition, which can be seen as a subproblem in consistency checking related to type safety. Also, VCC addresses early models and therefore, does not depend on any particular feature-oriented programming tooling (e.g., AHEAD), and does not require to write repeatedly common code or model fragments that will then be superimposed to generate a final model.

Well-Typed Java Programs. In a similar way than Thaker *et al.*, Kastner and Apel provide a formal approach for type-checking of SPL systems [Kästner and Apel, 2008]. They employ a calculus named Color Featherweight Java (CFJ) to develop an SPL that produces Featherweight Java (FJ) programs. They prove that given a well-typed CFJ SPL, all possible program variants that can be generated are well-typed FJ programs, i.e., generation preserves typing. Although this formalization covers only a small subset of Java, it provides theoretical insights about typing during the products generation mechanism.

The benefit of VCC with respect to the approach of Kastner and Apel (and also Thaker *et al.*) is that it does not use feature modules but Variant modules (each one identified by a name and a feature expression). Therefore in VCC there is an M-to-N (where M, N $>= 1$) relationship between features and related model fragments. The implication of using variant modules instead of feature modules is that "individual features or small sets of related features typically change more or less independently from other features" [Griss, 2000], and thus a group of changes to code or models (such as the Actions in the Variant blocks of VML4RE) can be related to threads drawn from a few related features. Particularly, in domain-specific application such as e-commerce systems, the ability to rapidly and consistently evolve sets of related features is key [Griss, 2000].

Feature-Oriented Model Templates Verification. Not much time before the work of Thaker *et al.*, [Czarnecki and Pietroszek, 2006] presented a verification procedure to ensure that no ill-structured template instances (i.e., concrete models of products) will be generated from a correct product configuration. They suggested the development of a type system that checks the entire assets of the feature-oriented SPL, instead of all individual feature-oriented programs, in a similar way to the approaches mentioned previously.

VCC goes further than previous approaches as it is interested in detecting patterns in the models that restrict or reveal incomplete parts in feature models. VCC does not assume that the feature model contains all domain constraints since its creation as it usually happens during early modeling in incremental SPLE. In fact, it benefits from the base models to suggest domain constraints in the feature model.

To the best of our knowledge, the approach of Czarnecki and Pietroszek based their product derivation strategy on removing parts of a large monolithic model that contains all the possible models fragments. VCC, in contrast, can be also applied when there are separated model fragments. Also, VCC does not rely on model templates annotated with feature expressions (which reduces readability of large models), but on separated composition and mapping models.

VCC Compared to Our Previous Works. Previous work of [Lopez-Herrejon and Egyed, 2010] also addressed consistency with an extension for multi-view modeling (MVM). Each view is represented by a different type of model, from which the authors focus on artifacts closer to software implementation, such as class, sequence and state diagrams. Lopez-Herrejón and Egyed observed that their approach for feature-oriented software development (FOSD) composes elements such as methods or classes (coarser granularity) but not their nested elements (finer granularity).

VCC takes into account more than one type of model. However, VCC is applied to early SPLE, proposes the use of complex feature expressions, a more accurate and complete main equation, composition technique different than FOSD, and a way to automatically create a mapping of complex feature expressions to model fragments. Also, VCC considers more types of model transformations (Actions), such as connections and insertions of model elements and use them to obtain dependencies between model elements and mappings between variant blocks (each variant block related to a feature expression) and model elements.

Another key difference between VCC and other type systems, such as the approach of Lopez-Herrejón and Egyed, is the way they support modularization of the model or code fragments. In FOSD the modularization is mainly based on feature modules, however, VCC can support a dominant modularization different to features modules. For example, developers may employ a use case diagram to represent a use case model for the entire system, instead of modeling each part of the use case diagram fragmented by feature modules. In any case, feature modules can be verified in VCC when each variant block is related to a feature expression that contains only one feature and related to a set of model fragments that are related exclusively to that feature as much as possible. We do not speculate about which modularization technique for the base models is better. We just argue that more freedom to the developers should be provided to decide how their models should be modularized and composed.

Other Related Works. The approaches presented here are the ones closer to our work. Nevertheless, there is a plethora of approaches related to software verification, consistency checking, safe composition, software testing and quality assurance that can be considered as related work but that cannot be examined in details. However, we want to mention the most related works in the scope of this J.UCS series of special issues on Software Components, Architectures and Reuse: Gheyi et al. proposes automatic checking of SPL refactorings during SPL evolution [Gheyi et al., 2011], Fernandes *et al.* introduces a mechanism to verify the consistency of context rules and product configurations [Fernandes et al., 2011], and Sabouri that verifies actor families using properties expressed in Linear Temporal Logic [Sabouri and Khosravi, 2013].

7 Conclusions and Future Work

This paper defines constraints and presents tool support for consistency checking between features models and base models in early SPL development. VCC has been applied to requirements analysis and architectural models but it may be applied for other kind of models and code. The feasibility of VCC and its tool was evaluated with two case studies where the results showed that performance is not an issue.

As part of our future work in consistency checking, we will address the co-evolution of the models, that is, to understand how the changes on each model (i.e., feature model, base models and composition model) influence its consistency with respect to the others. Also, we will perform a controlled experiment to measure the time required by a group of developers to create new rules or modify existing ones. Finally, we will establish consistency rules and constraints for base models that were not explicitly addressed in this paper, such as code modules or quality models. Here, we are addressing part of the problem in general.

References

[Alférez et al., 2009] Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., and Amaral, V. (2009). Multi-view composition language for software product line requirements. In *SLE*, pages 103–122.

[Apel et al., 2010a] Apel, S., Kästner, C., Größlinger, A., and Lengauer, C. (2010a). Type safety for feature-oriented product lines. *Autom. Softw. Eng.*, 17(3):251–300.

[Apel et al., 2010b] Apel, S., Scholz, W., Lengauer, C., and Kästner, C. (2010b). Language-independent reference checking in software product lines. In *FOSD*, pages 65–71.

[Batory, 2004] Batory, D. S. (2004). Feature-oriented programming and the ahead tool suite. In *ICSE*, pages 702–703.

[Batory, 2005] Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20.

[Benavides et al., 2010] Benavides, D., Segura, S., and Cortés, A. R. (2010). Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636.

[Clements and Northrop, 2002] Clements, P. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, USA.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[Czarnecki and Pietroszek, 2006] Czarnecki, K. and Pietroszek, K. (2006). Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE*, pages 211–220.

[Delaware et al., 2009] Delaware, B., Cook, W. R., and Batory, D. S. (2009). Fitting the pieces together: a machine-checked model of safe composition. In *ESEC/SIGSOFT FSE*, pages 243–252.

[Fernandes et al., 2011] Fernandes, P., Werner, C., and Teixeira, E. (2011). An approach for feature modeling of context-aware software product line. *J. UCS*, 17(5):807–829.

[Gheyi et al., 2011] Gheyi, R., Massoni, T., and Borba, P. (2011). Automatically checking feature model refactorings. *J. UCS*, 17(5):684–711.

[Griss, 2000] Griss, M. L. (2000). Implementing product-line features by composing aspects. In *SPLC*, pages 271–289.

[Kästner and Apel, 2008] Kästner, C. and Apel, S. (2008). Type-checking software product lines - a formal approach. In *ASE*, pages 258–267.

[Lopez-Herrejon and Egyed, 2010] Lopez-Herrejon, R. E. and Egyed, A. (2010). Detecting inconsistencies in multi-view models with variability. In *ECMFA*, pages 217–232.

[Mannion, 2002] Mannion, M. (2002). Using first-order logic for product line model validation. In *SPLC*, pages 176–187.

[Morganho and Gomes, 2008] Morganho, H. and Gomes, e. a. (2008). Requirement specifications for industrial case studies. Deliverable D5.2, Ample Project.

[Sabouri and Khosravi, 2013] Sabouri, H. and Khosravi, R. (2013). Modeling and verification of reconfigurable actor families. *J. UCS*, 19(2):207–232.

[Thaker et al., 2007] Thaker, S., Batory, D. S., Kitchin, D., and Cook, W. R. (2007). Safe composition of product lines. In *GPCE*, pages 95–104.

[Young and Young, 2005] Young, T. J. and Young, T. J. (2005). Using aspectj to build a software product line for mobile devices. msc dissertation. In *University of British Columbia, Department of Computer Science*, pages 1–6.

[Zschaler et al., 2009] Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., and Kulesza, U. (2009). Vml* - a family of languages for variability management in software product lines. In *SLE*, pages 82–102.

Appendix

This section includes the complete derivation of the requires and excludes rules. We use the following logical equivalences to translate expressions: (A) $\neg(x \to y) \equiv x \wedge \neg y$, (B) $x \to y \equiv \neg x \vee y$, and (C) $\neg(z \vee w) \equiv \neg z \wedge \neg w$. At the right side of each line appears the letter that identifies each logical equivalence.

Requires

The requires constraint in Equation 6 can be obtained from a disjunction of Equation 8 and Equation 9:

$$\begin{aligned}
&\neg\left(BC_{Req} \to FC\right) \\
&\equiv \neg\left(\left(FE\,(Var) \to \textstyle\bigvee_{i=1}^{n} FE\left(VarReq_i\right)\right) \to FC\right) \\
&\equiv \left(FE\,(Var) \to \textstyle\bigvee_{i=1}^{n} FE\left(VarReq_i\right)\right) \wedge \neg FC \quad (applying\ A) \\
&\equiv \left(\neg FE\,(Var) \textstyle\bigvee_{i=1}^{n} FE\left(VarReq_i\right)\right) \wedge \neg FC \quad (applying\ B)
\end{aligned} \tag{8}$$

$$\begin{aligned}
&\neg\left(FC \to BC_{Req}\right) \\
&\equiv \neg\left(FC \to \left(FE\,(Var) \to \textstyle\bigvee_{i=1}^{n} FE\left(VarReq_i\right)\right)\right) \\
&\equiv FC \wedge \neg\left(FE\,(Var) \to \textstyle\bigvee_{i=1}^{n} FE\left(VarReq_i\right)\right) \quad (applying\ A) \\
&\equiv FC \wedge FE\,(Var) \wedge \neg \textstyle\bigvee_{i=1}^{n} FE\left(VarReq_i\right) \quad (applying\ A) \\
&\equiv FC \wedge FE\,(Var) \wedge \textstyle\bigwedge_{i=1}^{n} \neg FE\left(VarReq_i\right) \quad (applying\ C)
\end{aligned} \tag{9}$$

Excludes

The Excludes constraint in Equation 7 can be obtained from a disjunction of Equation 10 and Equation 11:

$$\begin{aligned}
&\neg\left(BC_{Exc} \to FC\right) \\
&\equiv \neg\left(\left(FE\,(Var) \to \neg \textstyle\bigvee_{i=1}^{n} FE\left(VarExc_i\right)\right) \to FC\right) \\
&\equiv \neg\left(\left(\neg FE\,(Var) \vee \neg \textstyle\bigvee_{i=1}^{n} FE\left(VarExc_i\right)\right) \to FC\right) \quad (applying\ B) \\
&\equiv \left(\neg FE\,(Var) \vee \neg \textstyle\bigvee_{i=1}^{n} FE\left(VarExc_i\right)\right) \wedge \neg FC \quad (applying\ A) \\
&\equiv \left(\neg FE\,(Var) \vee \textstyle\bigwedge_{i=1}^{n} \neg FE\left(VarExc_i\right)\right) \wedge \neg FC \quad (applying\ C)
\end{aligned} \tag{10}$$

$$\begin{aligned}
&\neg\left(FC \to BC_{Exc}\right) \\
&\neg\left(FC \to \left(FE\,(Var) \to \neg \textstyle\bigvee_{i=1}^{n} FE\left(VarExc_i\right)\right)\right) \\
&\neg\left(FC \to \left(\neg FE\,(Var) \vee \neg \textstyle\bigvee_{i=1}^{n} FE\left(VarExc_i\right)\right)\right) \quad (applying\ B) \\
&FC \wedge \neg\left(\neg FE\,(Var) \vee \neg \textstyle\bigvee_{i=1}^{n} FE\left(VarExc_i\right)\right) \quad (applying\ A) \\
&FC \wedge FE\,(Var) \wedge \textstyle\bigvee_{i=1}^{n} FE\left(VarExc_i\right) \quad (applying\ C)
\end{aligned} \tag{11}$$