Supporting Consistency Checking between Features and Software Product Line Use Scenarios

Mauricio Alférez[1], Roberto E. Lopez-Herrejon[2], Ana Moreira[1], Vasco Amaral[1], Alexander Egyed[2]

[1]CITI/Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Caparica, Portugal
[2]Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
{mauricio.alferez,amm,vasco.amaral}@di.fct.unl.pt
{roberto.lopez,alexander.egyed}@jku.at

Abstract. A key aspect for effective variability modeling of Software Product Lines *(SPL)* is to harmonize the need to achieve separation of concerns with the need to satisfy consistency of requirements and constraints. Techniques for variability modeling such as feature models used together with use scenarios help to achieve separation of stakeholders' concerns but ensuring their joint consistency is largely unsupported. Therefore, inconsistent assumptions about system's expected use scenarios and the way in which they vary according to the presence or absence of features reduce the models usefulness and possibly renders invalid SPL systems. In this paper we propose an approach to check consistency —the verification of semantic relationships among the models— between features and use scenarios that realize them. The novelty of this approach is that it is specially tailored for the SPL domain and considers complex composition situations where the customization of use scenarios for specific products depends on the presence or absence of sets of features. We illustrate our approach and supporting tools using *variant* constructs that specify how the inclusion of sets of *variable features* (that refer to uncommon requirements between products of a SPL) adapt use scenarios related to other features.

1 Introduction

A *Software Product Line* (SPL) can be defined as "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"[7]. In SPLs, requirements are organized by *features* that are useful to express product functionalities concisely [19]. There are *common* features between all the products in the product line (sometimes called *mandatory* features), and there are *variable* features that allow distinguishing between products in a product line. In SPL development the *problem space* focuses on variability modeling and describes the different features available in an SPL and their interdependencies. A common representation to model variability are the *feature models,* where features are realized with correspondent artifacts, for example use scenarios diagrams [8].

To produce particular products from a SPL, feature realizations have to be composed according to a specific selection of features from a feature model usually called *product configuration* (also referred to *feature model configuration*). This process requires a mapping between features from a feature model, and artifacts such as use scenarios that realize them. A *use scenario* is a widely used technique that describes, step by step, how an actor is intending to use a system [14]. A number of different approaches have been proposed to create mappings among features and models [13,8,20]. However, ensuring consistency between feature models and recurring requirements specifications techniques such as use scenario modeling has not been thoroughly researched. In this context, by consistency checking we mean the verification of semantic relationships among features and use scenarios. Inconsistent assumptions about system's expected use scenarios and their variations according to the selection of different features, reduce the models usefulness and possibly renders invalid systems. Therefore, it is essential in SPL to determine whether the variability model and its use scenarios defined in the domain requirements specification enable the derivation of any product requirements specification that contains inconsistent requirements.

When a model-based approach is used to represent use scenarios (e.g., in form of use cases or activity diagrams), consistency goes beyond syntactical or semantic errors of each kind of model in isolation. For example, an actor that is not associated with any use case, a dangling node, a loop without exit conditions in activity diagrams or specific set of features that are both simultaneously (and incorrectly) declared as excluding and depending. It means that we aim at taking into account constraints that are not merely expressed in terms of only one language's metamodel which is generally well supported by UML editors in the case of use cases and activity diagrams (e.g., using OCL or hard-coded restrictions particular of each editor) or feature model editors (e.g., using *domain constraints* expressing features interdependencies, and hard-coded restrictions that constrain the construction of the models to conform to their metamodel). In our work, much of consistency checking difficulty lies on maintaining consistency among several, interrelated models. This can become a time-consuming and error prone task given that the number of ways to compose feature realizations grows exponentially with the possible number of SPL features that can be used in a particular product.

In this paper, we present an approach whose driving objective is to enable consistency checking in the problem space between requirements models such as use scenarios and features. It transforms generic constraints expressions between single features to rules specifically tailored for use scenarios and set of features. Then, it employs propositional formulas to relate these specialized rules to the models involved in the creation of customized use scenarios for specific products. These propositional formulas are produced based on the relationships between: i) *domain constraints* that can be obtained from the SPL feature model, ii) the meaning of the relationships between fragments in the use scenarios and SPL features, and iii) a *composition model* that specifies how to vary SPL use scenarios. Checking if all the products in an SPL satisfy consistency constraints is based on searching for a satisfying assignment of a propositional formula. Therefore, our tool translates propositional formulas that can be evaluated by *satisfiability* (SAT) solvers [1]. In case there are constraints that are not satisfied by the SPL, our tool presents to the developer the particular features and fragments of the use scenarios

involved in the violation of the constraint. In our home automation case study this information was useful to take informed decisions about the modifications and additions of domain constraints, use scenarios and its composition specification. The results of the application of our approach are encoraging because they did not show scalability and performance issues, however, we need more extensive validation of our approach with different case studies.

2 Background and Motivation

To understand consistency between features and use scenarios let us introduce first the models we use: features model, use case/activity diagrams, mapping model between features and use cases/activity models, and a composition specification model. After this, we exemplify inconsistency using these models.

2.1 Models Involved in Consistency Checking

Feature Model. A feature model describes a set of all possible valid product configurations [8]. A configuration specifies a concrete product in terms of its features.

Figure 1-1 shows a sample feature model of part of our running example, the *Smart Home* SPL [18]. Smart Home has four optional features, AUTOMATED WINDOWS(AW), AUTOMATED HEATING (AH), REMOTE HEATING CONTROL (RHC) and INTERNET as a mean to control the heater and other devices remotely. Also, it has a set of common features, such as MANUAL WINDOWS and MANUAL HEATING that will be included in all the target products to be produced using the Smart Home SPL.

Specific product configurations can be defined selecting optional features in the feature model 1-1. Figure 1-2 shows a sample product configuration of the Smart Home SPL called PRODUCT-1 that will be used to illustrate consistency problems between features and use scenarios. PRODUCT-1 has all features except AUTOMATED WINDOWS (AW). Domain constraints in the feature model such as the REQUIRES relationship from RHC to INTERNET, can be added incrementally and in parallel with the creation of use scenarios (discussed below).

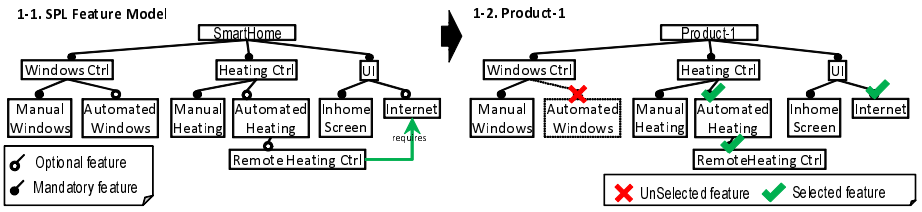

Fig. 1. (1) Simplified sample of the Smart Home feature model; (2) Sample SmartHome configuration that excludes the Automated Windows feature.

Use Scenarios. Features can be realized with other models such as use scenarios. To model use scenarios we employ use case and activity diagrams because they are commonly used in mainstream UML-based methods such as RUP [16] and, in contrast to mere free-form textual scenario descriptions, they help to reduce ambiguity in the specifications [19].

Use case 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 diagrams and specific paths through those use cases represented in activity diagrams. The customization is guided by a composition specification discussed in next subsection.

Figure 2-1 (Left) shows part of the final target model composed for PRODUCT-1. The INCLUDES relationship describes the case where one use case, the *base* use case, includes the functionality of another use case, the *inclusion* use case. 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 two or more 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 CTRLTEMPREMOTELY that includes the use case OPENANDCLOSEWINAUTO may imply that feature REMOTEHEATINGCNTRL(SH) requires feature AUTOMATEDWINDOWS (AW). We discuss this and other consistency constraints in Section 3.

Figure 2-1 (Right) shows an activity diagram that depicts the possible scenarios for the use case CNTRLTEMPREMOTELY that comprises activities for the use cases OPENANDCLOSEWINAUTO, CALCENERGYCONSUMPTION and ADJUSTHEATERVALUE. Within this activity diagram it is possible to select several scenarios that correspond to different paths. Two of all the possible scenarios are: Scenario i) includes reaching the in-home temperature and save energy by means of closing some windows, and Scenario ii) to use the heater to reach the desired in-home temperature. It is important to note that the customization of activity diagrams and scenarios depends on the features chosen for the SPL product and also on the relationship with the use case model. For example, in PRODUCT-1 the feature AUTOMATEDWINDOWS was not selected, therefore the WINACTUATOR actor in the use case diagram as well as the swimlane (also called activity partition) related to WINDOWSACTUATOR should not appear in any diagram. Therefore, scenarios such as i) are not realizable because of the lack of windows actuators. This and other constraints will be discussed in Section 3.

Composition Specification To evidence consistency problems between features and use scenarios we employ a composition process (also called, derivation process) for use cases and activity diagrams. Languages such as the VML4RE (Variability Modelling Language for Requirements) [20,4] help to specify how use scenarios can be customized.

Figure 3 illustrates a composition specification that guides the specification of the transformation of requirements specifications of products in the SmartHome SPL. VML4RE [20,4] is a textual language that allows associating *actions*, that wrap a set of model transformations for specific requirements models such as use cases and activity

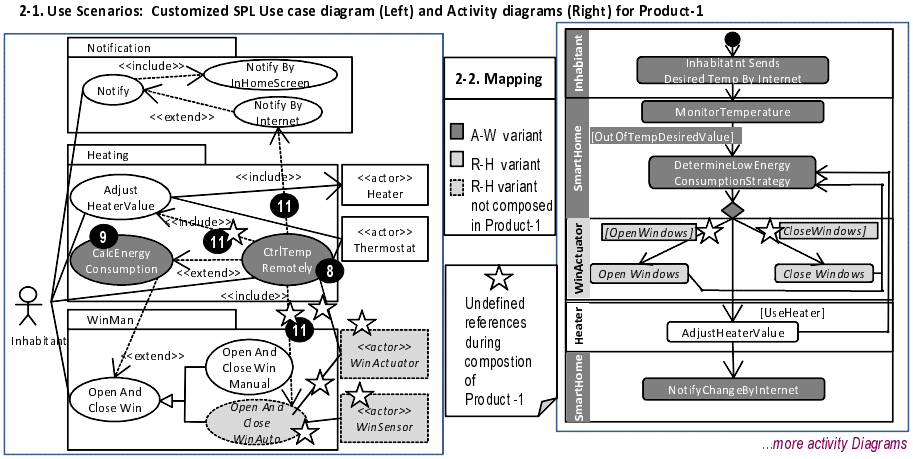

Fig. 2. (1) Referencing undefined model fragments during composition for PRODUCT-1 in the Use Case model (left side) and in the Activity Diagram for the CntrlTempRemotely scenario (right side). (2) Mapping variants to model fragments

diagrams, to combinations of features written as logic expressions that we call *feature expressions*. Feature expressions can be i) *atomic* that represent single features such as "Automated Windows" in Figure 3, Line 1, and ii) *compound* that also contain logic operators such as AND, NOT and OR such as "And ("Remote Heating Ctrl","Automated Heating","Internet")" in line 7. Feature expressions evaluation works as follows: if AUTOMATEDHEATING, REMOTE HEATING CTRL, AUTOMATED HEATING and INTERNET features are selected in a product configuration, the feature expression associated to the variant named "R-H" (i.e., the compound feature expression: And ("Remote Heating Ctrl","Automated Heating","Internet")) will be evaluated to TRUE. The consequence of this is that the actions that are inside the "R-H" variant block (Figure 3, lines 6-13) will be processed and applied to a base model. For example, the CNTRLTEMPREMOTELY use case will be inserted into the package HEATING and then it will be related to other use cases using INCLUDES and EXTENDS relationships. If more than one feature expression is evaluated to TRUE, the default composition order follows a top-down sequence (which corresponds to a left-right sequence in Figure 3).

```
1.  Variant { name : "A-W" for : "Automated Windows"
2.    + UseCase : "OpenAndCloseWinAuto" in Package : "WinMan"
3.    + Partition : "WinActuator" in ActDiagram :"CtrlTempAuto"
4.    ...
5.  }
    ...more variants
```

```
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. }
```

Fig. 3. Composition specification of variants A-W and R-H

Mapping Model. Figure 2-1 (Left) and (Right) show use case and activity diagrams fragments, such as actors and use cases, related with the variants shown in Figure 3. The base mechanism to relate requirements model fragments to features is to use a correspondence table (or mapping table), as presented by [11], [19] and [3]. In our case, we parse the composition specification to generate the mapping between variants and parts of the use cases, therefore, for example if variant named A-W inserts the OPENANDCLOSEWINAUTO use case, we link A-W to OPENANDCLOSEWINAUTO. To facilitate the visualization of such relationships with the models, in the figure we assign different gray tones to the models fragments according to the features that they are related to (see mapping in Figure 2-2). Please note that specific model fragments could be related also to more than one variant. This may be considered as a m-to-n (m and n >= 1) mapping between variants and model fragments and is not illustrated in Figure 2.

2.2 Consistency Checking Motivation

Consistency checking has to ensure that inconsistent requirements do not become part of the requirements specifications of a given product. Our work aims at guaranteeing that *all* the products that could be derived from a feature model indeed have consistent requirements specifications. This is achieved through the description and verification of semantic relationships between feature model and use scenarios. One of the possible inconsistencies between features and use scenarios in the Smart Home SPL happens between the relationship of variants R-H and A-W, and the INCLUDES relationship between the use cases CNTRLTEMPREMOTELY and OPENANDCLOSEWINAUTO which are related to R-H and A-W variants respectively. The domain requirements are:

R1- Only one, none or both R-H and A-W variants can be included in a product. (This is implicit in the feature model and composition model because all the features in the feature expression of R-H variant are optional (i.e., REMOTE HEATING CNTRL, AUTOMATED HEATING and INTERNET are optional features), and the only feature in the feature expression A-W is also optional (i.e., the AUTOMATED WINDOWS feature is optional)); and

R2- If the use case CNTRLTEMPREMOTELY is provided in a product then the use case OPENANDCLOSEWINAUTO must be provided too, (This is implicit in the includes relationship from the use case CNTRLTEMPREMOTELY to OPENANDCLOSEWINAUTO in the use case diagram in Figure 2-1 (Left)).

Figure 2-1 shows PRODUCT-1 built using the composition model shown in Figure 3. In PRODUCT-1 the feature expression of variant R-H (3, line 7) evaluates to TRUE. However, because Figure 1-2 does not include the AUTOMATED WINDOWS feature, the feature expression of variant A-W (i.e., AUTOMATED WINDOWS) (3, Line 1) evaluates to FALSE and the actions inside its variant block are not processed. We annotated the diagrams with numbers that represent the line in Figure 3 where a composition action is specified. Note that we omitted some of the actions, for example, the insertion of some actors such as WINSENSOR and WINACTUATOR and some partitions such as HEATER.

PRODUCT-1 presents inconsistent requirements *R1* and *R2*. This is evident during composition of use scenarios. See lines 10-11 when the action “Includes from UseCase

: "CtrlTempRemotely" to UseCase(s) : "NotifyByInternet" and "OpenAndCloseWinAuto" and "AdjustHeaterValue" ” references elements such as the use case OPENANDCLOSEWINAUTO that do not exist in the model. In this case, PRODUCT-1 fulfills requirement *R1*, but not requirement *R2*. The result is that the functionality provided by OPENANDCLOSEWINAUTO will not be present in the requirements of PRODUCT-1 and therefore it will not be taken into account in later stages of its development process.

It is not too difficult to check consistency manually in small examples with a reduced number of features such as the one mentioned previously. One solution to solve the inconsistency for our example would be to guarantee the presence of the feature AUTOMATED WINDOWS when AUTOMATIC HEATING or REMOTE HEATING CTRL are selected, in every possible feature model configuration using a domain constraint REQUIRES. Another solution is to establish that AUTOMATED WINDOWS will be a mandatory feature in the SPL. However, the number of possible feature combinations may grow exponentially with the number of features of the SPL. The result of this explosion is that it becomes unfeasible to manually check the consistency of all the products.

To guarantee that all the products that could be derived from a feature model indeed have consistent requirements specifications we take into account the relationships between domain requirements specified using use scenarios and feature models to propose rules and constraints to support consistency checking in SPLs use scenarios as it is shown in the next section.

3 Consistency Checking between Features and Use Scenarios

While some product configurations of a feature model may generate consistent use scenarios, other product configurations based on the same feature model could lead to inconsistencies in the requirements specifications. In this section we present our approach for consistency checking between SPL features and use scenarios.

3.1 Approach Overview

Figure 4 presents an overview of our approach. Section 2 explained and exemplified the specification of a feature model, use scenarios (Figure 4, Step 1), and the mapping between variants and fragments of the use scenarios (Step 2). Based on previous work [17], we have developed a consistency checking approach for use scenario composition based on variants. This approach relies on the domain evaluation of feature expressions, written as propositional formulas that are associated to a *variant* and transformations of use scenarios called *actions*. We denote D_f the domain constraints that can be derived from a feature model of an SPL and are expressed in terms of atomic features f (Step 3), and C_{VAR_f} denote composition constraints that will be derived in next section (Step 4) and are expressed in terms of variants (VAR_f). We use propositional logic to express and relate D_f and C_{VAR_f}(Step 5). Because we are interested in verifying that all members of the product line satisfy a given composition constraint, Equation 1 should not be “satisfiable”. If it is satisfied, it means that there is a product of the product line that does not meet constraint C_{VAR_f}. The violating product configurations can be identified

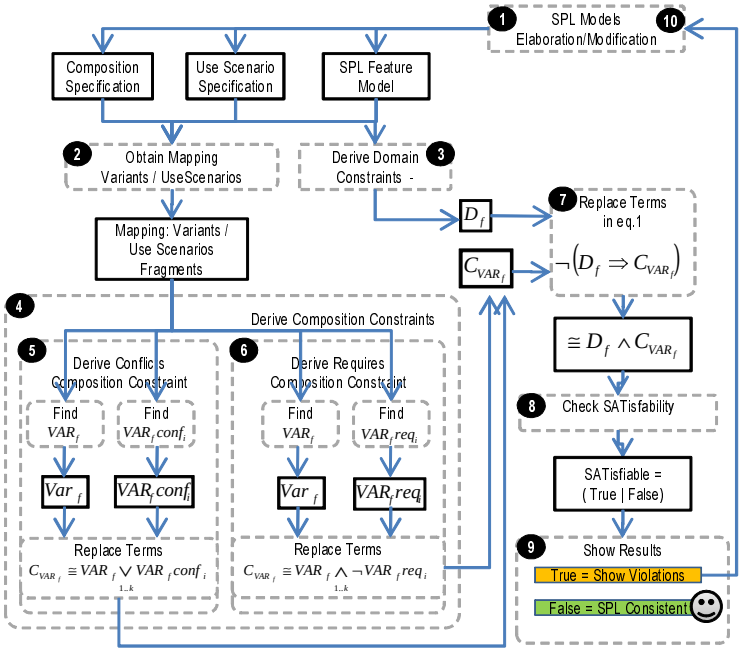

Fig. 4. Overview of Our Consistency Checking Approach

using a SAT solver (Step 7 and 8). This can support the developer to take informed decisions on modifications of the initial SPL models, for example, creating or modifying domain constraints (Step 10).

$$\neg \left(D_f \Rightarrow C_{VAR_f}\right) \tag{1}$$

Section 2-1 shows that at least one product (i.e., PRODUCT-1) from the products that can be configured based on the feature model of the Smart Home SPL is inconsistent. In that case, *composition constraints* (also called *implementation constraints*) between the elements in use scenarios such as the INCLUDES between use cases, imply the application of domain constraints for example, turning the AUTOMATEDWINDOWS feature from optional to mandatory or creating a REQUIRES dependency (also called domain constraint) from AUTOMATEDHEATING to AUTOMATEDWINDOWS. That particular inconsistency that will help to explain our approach can be defined as:

– *Rule Required Inclusion Use Case:* at least one variant $(VAR_f req_i)$, defines an inclusion use case that must be selected in every feature configuration that contains the variant (VAR_f) which introduces a base use case linked to the inclusion use case.

3.2 Deriving Domain Constraints (D_f)

Figure 4 - Step 3 shows that the domain constraints are derived from a SPL feature model. Therefore the D_f in a SPL is the same for all the possible products configura-

tions and do not vary depending on the consistency rule. Using a well-known translation table between feature models and propositional formulas (see Figure 5) helps to get D_f in Equation 1. In Equation 2 we only show the HEATING-CTRL branch because it is the most complex branch in Figure 1-1 and relates directly with our exemplar "Required Inclusion Use Case" rule. The translation obtained in the first line of Equation 1 means that all products unconditionally must contain the root feature SMARTHOME. The second line means that given that HEATING CTRL is a mandatory feature, it must be included in all the products. The third line means that MANUALHEATING is included in all the products that include their parent feature (i.e., HEATINGCTRL), in contrast to AUTOMATEDHEATING and REMOTEHEATINGCTRL (lines 3-4), that may be or not included when their respective parents HEATINGCTRL and AUTOMATEDHEATING are included in a product. Line 5 means that REMOTEHEATINGCTRL *requires* of the INTERNET feature.

$$
\begin{aligned}
&1.(SmartHome \Leftrightarrow TRUE) \wedge \\
&2.\ (SmartHome \Leftrightarrow HeatingCtrl) \wedge \\
&3.\ (HeatingCtrl \Leftrightarrow ManualHeating) \wedge (AutomatedHeating \Rightarrow HeatingCtrl) \wedge \\
&4.\ (RemoteHeatingCtrl \Rightarrow AutomatedHeating) \wedge \\
&5.\ (RemoteHeatingCtrl \Rightarrow Internet)
\end{aligned}
\tag{2}
$$

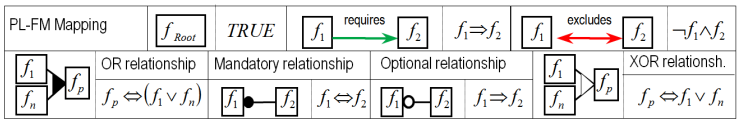

Fig. 5. Mapping from Feature Model to Propositional Logic [6].

In this section we addressed D_f, the first part of Equation 1. Next section presents C_{VAR_f} that comprises a set of constraints that are essential for consistency between use scenarios and the set of domain constraints expressed in Equation 2.

3.3 Deriving Composition Constraints (C_{VAR_f})

Composition constraints act as consistency rules describing the semantic relationships that must hold among the different models. Figure 4-4 shows two kinds of composition constraints that can be expressed in propositional logic. We classified them according to the type of domain constraint that they relate with: i) a constraint that implies a REQUIRES relationship between features that therefore implies dependencies between variants (Figure 4- Step 4), and ii) a constraint that implies a EXCLUDES relationship (Figure 4- Step 6) between features and therefore implies incompatibilities between variants (Figure 4- Step 5). This section shows those constraint equations expressed in propositional logic.

EXCLUDES Relationship: Let VAR_f be a variant that defines a model element e. A variant $VAR_f conf_1$ conflicts with VAR_f if $VAR_f conf_1$ defines a model element c which cannot be present in the same requirements specifications of a product where element e is also present. Therefore, because of the incompatibility between elements

e and c, if variant VAR_f is selected then variant $VAR_f conf_{\text{I}}$ should not be selected in the same product configuration. This is denoted in the following expression where k represents the number of variants in the composition specification:

$$C_{VAR_f} \equiv VAR_f \Rightarrow \neg \left(\bigvee_{1..k} (VAR_f conf_{\text{i}}) \right) \equiv \neg VAR_f \vee \neg \left(\bigvee_{1..k} (VAR_f conf_{\text{i}}) \right) \quad (3)$$
$$\equiv VAR_f \wedge \bigvee_{1..k} VAR_f conf_{\text{i}}$$

REQUIRES Relationship*:* Let VAR_f be a variant that refers to a model element e defined by another variant. To be consistent, the requirements specifications of a product that includes variant VAR_f must also include at least one other variant $VARreq_{\text{I}}$ (required variant) where element e is defined. This is denoted in the following expression where k represents the number of variants in the composition model:

$$C_{VAR_f} \equiv VAR_f \Rightarrow \bigvee_{1..k} (VAR_f req_{\text{i}}) \equiv \neg VAR_f \vee \bigvee_{1..k} (VAR_f req_{\text{i}}) \quad (4)$$
$$\equiv VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_{\text{i}}$$

The rule "Required Inclusion Use Case" mentioned at the beginning of this section is an example of this last kind of constraint expression. An instance of this constraint is found in our motivation example related to the use scenario of CNTRLTEMPREMOTELY. For example, given that the variant VAR_f = 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 CTRLTEMPREMOTELY, we want to guarantee that there are at least one variant (e.g., $VAR_f\ req_{\text{I}}$ = A-W) related to the inclusion use case OPENANDCLOSEWINAUTO (i.e., model element e = use case OPENANDCLOSEWINAUTO), and that its feature expression evaluates to TRUE in all possible feature model configurations. This way, we guarantee the presence of the functionality required by CTRLTEMPREMOTELY, such as to include a WINDOWSACTUATOR that regulates the temperature opening and closing windows. Thus, we can get a constraint instance replacing the variants by their corresponding feature expressions:

$$(\, RemoteHeatingCntrl \wedge AutomatedHeating \wedge Internet \,) \quad (5)$$
$$\wedge \neg (\, AutomatedWindows \,)$$

3.4 Replacing Terms in Equation

The replacing step depicted in Figure 4- Step 7 depends on the kind of constraint that we created in previous section. If we replace C_{VAR_f} of Equation 4 in Equation 1 and perform some logic manipulation to translate expressions of the form $x \Rightarrow y$ to $\neg x \vee y$, and $x \vee y$ to $\neg x \wedge \neg y$ respectively, we obtain the expression in Equation 6.

$$REQUIRES: \neg \left(D_f \Rightarrow \left(VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_{\text{i}} \right) \right) \equiv D_f \wedge VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_i \quad (6)$$

Similarly, if we replace C_{VAR_f} of Equation 3 in Equation 1, and perform some logic manipulation, we obtain the expression in Equation 7.

$$EXCLUDES: \neg \left(D_f \Rightarrow \left(VAR_f \wedge \bigvee_{1..k} VAR_f conf_{\text{i}} \right) \right) \equiv D_f \wedge VAR_f \wedge \bigvee_{1..k} VAR_f conf_i \quad (7)$$

3.5 Checking SATisfability

Figure 4- Step 8 shows that the input for satisfability checking are expressions such as the ones in 6 and 7. Each expression to be checked is instantiated with:

i) the specific domain constraints, D_f of the SPL produced in Equation 2,

ii) the feature expressions related to the variants VAR_f and either the set of required variants$VAR_f\ req_i$, or the set of conflictant variants $VAR_f\ conf_i$.

Equation 4 evaluates to true when any action inside variant VAR_f requires an element or set of *required elements* that are not composed in the use scenarios. It happens because none of the correspondent variants $VAR_f\ req_i$ that introduce the *required elements* was selected in the product configuration. Also, expression 3 evaluates to false when variant VAR_f defines an element or set of elements that are introduced in the use scenarios that also contain elements defined by other variant(s) $VAR_f\ conf_i$.

3.6 Show Results and SPL Models Modification

The possible results generated by a SATisfability checker for each expression (Figure 4- Step 9) can be TRUE (satisfiable) or FALSE (insatisfiable). In case we obtain FALSE for all the expressions, we know that the SPL is consistent because there are not inconsistencies between the relationships and dependencies (e.g., excludes, optional, mandatory, requires) between features depicted in the SPL feature model, and the use scenarios. In case we obtain a TRUE in an expression, our tool based on the mapping between variants and model elements in the use scenarios shows a list of the variants and the model fragments related to the inconsistency. Taking the example of the Smart Home feature model depicted in Figure 1, the result of the SAT solver for the Rule - Required Inclusion Use Case is that it is satisfiable (i.e., it evaluates to TRUE). Which means that there is an inconsistency between the features and use scenarios. An example of the type of message generated by our tool to the user 4 is:

> "...Inconsistent use scenario(s) [CTRLTEMPREMOTELLY] and feature(s) in feature expression(s) of variant(s) [A-W], [R-H]. The Action: [Includes from UseCase: "CtrlTempRemotely" to Use Case(s) "OpenAndCloseWinAuto"] implies a [REQUIRES] relationship between variant [R-H] and required variant(s) [A-W] that is not enforced in the SPL feature model...".

Based on this information, for the SAT solver to evaluate to FALSE, the developers may consider for example to:

- Modify the feature model: the set of SPL domain constraints that can be extracted from the feature model can be modified for example creating a REQUIRES relationship for AUTOMATEDHEATING feature to AUTOMATEDWINDOWS, or changing the AUTOMATEDWINDOWS feature from optional to mandatory.

- Modify use scenarios and composition model: for our particular rule, developers may want to check if in fact the INCLUDES association between use cases CTRLTEMPREMOTELY and OPENANDCLOSEWINAUTO is mandatory for every single product or not.

4 Tool Support

Tools for consistency checking can be highly effective for detecting errors in SPL requirements specifications. Such tools not only can find errors people miss, but also they can alleviate developers from the tedious and error-prone task of checking requirements specifications for consistency. Our tool prototype Variability Consistency Checker for Requirements (VCC4RE) [2] was designed to support the process described in Section 3.1 and consist on several components: (i) composition models editor for the VML4RE language, (ii) two translators: one from propositional formulas in prefix notation to conjunctive normal (CNF) form in DIMACS format [1], and the other from the CNF clauses provided by the feature model editor to DIMACS format; and finally (iii) the consistency checker.

We created the *composition model editor* using EMFTEXT [1]. It provides the software infraestructure to derive an initial concrete syntax and plug-in based on the metamodel of our VML4RE language written in Ecore [2]. We employ this technology mostly because of two reasons: first, it separates concrete syntax and abstract syntax which eases the maintenance of the language, and second, it provides a default Human Usable Notation (HUTN) [3] as concrete syntax. Using the HUNT concrete syntax in comparison with our previous tool version [20] allows a more usable and suitable notation for describing requirements composition.

We created a *translator for feature models* created with the SPLOT editor [4]. We chose SPLOT because it allows us to share and edit our models collaboratively via web, and because it generates the CNF formula that represents the domain constraints (D_f) in our equations that later we transform to a widely accepted standard format for boolean formulas in CNF called DIMACS.

Also, we created another *translator* to obtain the feature expressions related to each variant in $VAR_f \wedge \bigwedge_{1..k} \neg VAR_f req_{\mathrm{i}}$ and $VAR_f \wedge \bigvee_{1..k} VAR_f conf_{\mathrm{i}}$ from our composition model. It translates from a prefix notation of propositional formulas of our composition specification, to CNF formulas in DIMACS format. Composition model, consistency rules, as well as the use cases and activity diagrams modelled in any Ecore-based UML tool are interpreted by our consistency checker to produce a set of constraints expressions in CNF DIMACS format. Then, it is possible to use a standard SAT solver to determine the satisfability of each formula. In our case, we experimented with PicoSAT [5] and SAT4J[6].

5 Evaluation

The complete Smart Home SPL was used to evaluate our approach. We chose this case study because, despite of being a large-scale embedded system, this can be understood

[1] http://www.emftext.org/: Concrete syntax mapper
[2] http://www.eclipse.org/modeling/emf/: Eclipse Modelling Framework based on Ecore
[3] http://www.omg.org/spec/HUTN/: The OMG HUTN specification.
[4] http://www.splot-research.org/: Software Product Line Online Tools
[5] http://fmv.jku.at/picosat/: PicoSAT: Pico satisfability solver
[6] http://sat4j.org/: SAT for Java

by a general reader given its application in everyday's life. Also, we had previous experience modelling variability and part of the use scenarios of the Smart Home supported by one of our industrial partners who set the requirements of the system [18].

Features	59	Variants	27
CNF clauses	79	Rules	6
Use Cases	36	Rule instances checked	74
Activity Diagrams	13	Domain constraints created after consistency checking	16
Scenarios	48	Time taken in consistency checking in milliseconds	810

Table 1. Evaluation results using VCC4RE in the Smart Home SPL

Table 1 summarizes some information about the evaluation. The Smart Home has 59 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. These features describe variability at the use scenarios therefore, it is relevant to all kind of SPL stakeholders which are not necessarily experts in domotics and its implementation technologies. When mapped to propositional formulas the feature model produced 79 clauses in CNF format.

We modelled the use scenarios manually using an open source Ecore-based UML tool called Papyrus [7]. In total we modelled 36 use cases, 13 activity diagrams that can represent 48 different possible scenarios, and an initial set of 6 rules for use scenario consistency that follow a very similar reasoning than the rule *Required Inclusion Use Case* explained in Section 2. They vary only in the kind of model elements and their relationships with other model elements, for example: inclusion, generalization, specialization, aggregation and mapping between activity diagram partitions to actors and use cases. Based on the scenarios and feature model we specify 27 *variant* modules using VML4RE. Before applying our approach for consistency checking, we found that using the Smart Home feature model it was possible to generate ONE BILLION product configurations. This information can be obtained using the feature model analyzer provided by the SPLOT tool and allows us to evidence the complexity of checking consistency without any approach and tool support such as the one that we proposed in this paper.

In our experiments we found in total 74 rules instances to check. Using this information we created 16 domain constraints, mainly dependencies of type REQUIRES between features in the feature model that finally help us to solve consistency between use scenarios and features. 16 errors is a significant number taking into account mainly two things: i) Use scenarios, feature model and composition were first carefully modeled and before applying our approach they were apparently "perfect", and ii) The large number of possible combinations of features, the number of variants and use scenarios makes this task challenging, however our approach and tool support gives results in a "blink of an eye". The time taken to evaluate consistency rules using the Pico SAT solver and produce the results is in the order of milliseconds when run on an Intel

[7] http://www.eclipse.org/modeling/mdt/?project=papyrus : Papyrus

Core-Duo i5 at 2.4 Ghz. Given that in VCC4RE, feature models and constraints are mapped to clauses, 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. However, though encouraging results, the scalability of our approach needs to be more extensively validated with more complex case studies and probably using more consistency rules. Doing that is part of our future work.

6 Discussion and Related Work

An issue in the development of SPLs is the lack efficient approaches for consistency checking among all the artifacts, including requirements specifications. In model-driven development this becomes a crucial 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 (i) the transformations, (ii) the source models of each transformation and (iii) the information added after each transformation. Therefore, to create constraints helps not only to compose models that helps to understand the intended products to the SPL stakeholders, but also to obtain good quality source models that are the base for deriving good quality code.

The idea of this paper was to explore whether it was possible to use so called "hard" methods for consistency checking as early as requirements analysis. Usually such methods are used much later in the development. We believe now 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 transparent for the SPL developer and therefore, it does not add extra complexity to the modeling process. SAT solvers are implemented by libraries that are used internally by VCC4RE.

The effective use of use scenarios in SPL demands mechanisms for consistency checking that cope with variability. However, to the best of our knowledge, this issue has not been extensively researched except by Czarnecki, et al [9]. They observed that implementation constraints should follow from domain constraints. Their findings apply to a different composition technique that uses model templates to generate concrete models for product configurations. That work ensures that no ill-structured template instances (i.e., concrete models of products) will be generated from a correct product configuration. In comparison with that work, we check consistency between use scenarios and feature models of domain requirements specifications and we do not assume that the feature model contains all domain constraints since its creation as it usually happens in incremental SPL development processes. In fact, our approach benefits from the semantic of the use scenarios to deduce domain constraints.

There are different research areas related to our work and that have been taken into account the importance of consistency constraints in models. In the field of well-formedness of models for example Egyed [10]. Also, for single systems modeling, Jacobson [15] used aspect-oriented use case models. However, none of those works check consistency of SPL models, and their composition mechanism does not support model

weaving of model fragments as it is possible with a requirements-tailored composition language as VML4RE.

Previous work [17] addressed consistency in composition in multi-view modeling in SPL following a FOSD [5] approach for models closer to the product implementation. Also, Harhurin and Hartmann [12] provided denotational semantics and a notation called *Service Diagram* to describe system functionality and variability. Both works focus only on depedencies between atomic features. Our work addresses composition of requirements specifications and an advanced way for model composition based on an aspect-oriented framework VML4RE that is capable to manage variants in addition to atomic features.

7 Conclusions and Future Work

This paper establishes constraints and presents tool support for consistency checking between use scenarios and features in the SPL domain, using feature models and VML4RE. However, our approach does not depend on the use of VML4RE. We use it because its actions facilitate expressing the composition in use scenarios. The objective of checking consistency is to guarantee that all the products that could be derived from a feature model indeed have consistent requirements specifications. This means without omitting information or containing conflicting requirements that eventually may cause errors when transformed and implemented into more platform dependent models and code.

The feasibility of our approach was evaluated using a prototype tool and a home automation case study. The results show that performance and scalability were not an issue. However, these aspects need further assessment with larger and more complex SPLs and consistency rules. Such assessment is part of our future work.

We think that the application of constraints is necessary but do not satisfy completely the problem of consistency checking of models. This problem also depends on the composition order of the variants and in the application order of the actions inside each variant block. Currently, we are researching algorithms to calculate the precedence order between variants and its application in non-monotonic composition. Our proposal here is a proof of concept. Our strategy can be extended for other models, for example to model variability of system qualities, that is not within the scope of our paper and is part of our future work. Here, we are addressing part of the problem for some models.

8 Acknowledgements

This work was partially supported by the CITI, Portugal, the European project AMPLE, contract IST-33710 and the grant SFRH/BD/46194/2008 of Fundação para a Ciência e a Tecnologia, Portugal. It was also partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions—IEF project number 254965. We thanks to Alexander Nöhrer for its Java interface for PicoSAT.

References

1. Int. confs. on theory and applications of satisfiability testing, http://www.satisfiability.org/
2. Alférez, M.: Variability consistency checking for requirements tool, http://citi.di.fct.unl.pt/prototype/prototype.php?id=116
3. Alférez, M., Kulesza, U., Sousa, A., Santos, J., Moreira, A., Araújo, J., Amaral, V.: A model-driven approach for software product lines requirements engineering. In: SEKE. pp. 779–784 (2008)
4. Alférez, M., Santos, J., Moreira, A., Garcia, A., Kulesza, U., Araújo, J., Amaral, V.: Multi-view composition language for software product line requirements. In: SLE. pp. 103–122 (2009)
5. Batory, D.: Ahead tool suite, hhttp://www.cs.utexas.edu/users/schwartz/ATS.html
6. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6), 615–636 (2010)
7. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA (2002)
8. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
9. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: Proc. of the GPCE'06. pp. 211–220. GPCE '06, ACM, New York, NY, USA (2006)
10. Egyed, A.: Fixing inconsistencies in uml design models. In: Proc. of the 29th Int. Conf. on Software Engineering. pp. 292–301. ICSE '07, IEEE Computer Society, Washington, DC, USA (2007)
11. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
12. Harhurin, A., Hartmann, J.: Towards consistent specifications of product families. In: FM. pp. 390–405 (2008)
13. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: mapping features to models. In: Companion of the 30th Int. Conf. on Software Engineering. pp. 943–944. ICSE Companion '08, ACM, New York, NY, USA (2008)
14. Jacobson, I.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
15. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series). Addison-Wesley Professional (2004)
16. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edn. (2003)
17. Lopez-Herrejon, R.E., Egyed, A.: Detecting inconsistencies in multi-view models with variability. In: ECMFA. pp. 217–232 (2010)
18. Morganho, H., Gomes, e.a.: Requirement specifications for industrial case studies. Deliverable D5.2, Ample Project (2008), www.ample-project.net
19. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
20. Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U.: Vml* - a family of languages for variability management in software product lines. In: SLE. pp. 82–102 (2009)