Detecting Inconsistencies in Multi-View Models With Variability

Roberto E. Lopez-Herrejon, Alexander Egyed

Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
roberto.lopez@jku.at, alexander.egyed@jku.at

Abstract. Multi-View Modeling (MVM) is a common modeling practice that advocates the use of multiple, different and yet related models to represent the needs of diverse stakeholders. Of crucial importance in MVM is consistency checking — the description and verification of semantic relationships amongst the views. Variability is the capacity of software artifacts to vary, and its effective management is a core tenet of the research in Software Product Lines (SPL). MVM has proven useful for developing one-of-a-kind systems; however, to reap the potential benefits of MVM in SPL it is vital to provide consistency checking mechanisms that cope with variability. In this paper we describe how to address this need by applying Safe Composition — the guarantee that all programs of a product line are type safe. We evaluate our approach with a case study.

1 Introduction

Extensive experience in software architecture and design has shown the importance and necessity of using multiple, different, and yet related models to represent the perspectives and information needs of diverse system stakeholders throughout the development process. This practice is known as *Multi-View Modeling (MVM)*[1,2,3]. UML is an example of MVM where the different types of diagrams can represent distinct views of the same system.

MVM intrinsically requires *consistency checking* whereby all views must adhere to *consistency rules* that describe the semantic relationships amongst their elements [1,2,3]. A classical example of a consistency rule in UML is that if a sequence diagram has a message `m` targeting an object of class `C`, then the class diagram of class `C` must contain method `m`.

Variability is the capacity of software artifacts to vary [4], and its effective management is a core tenet of the research in *Software Product Lines (SPL)*[5,6,7]. On one hand, the significant benefits of applying SPL practices have been extensively documented and corroborated both in academia and industry [6,8,7]. On the other, MVM has proven useful for the development of one-of-a-kind systems. Several research works have added variability into UML modeling because of its extensive use in industry and academia [9,10,11]. However, the effective use of MVM in SPL demands mechanisms for consistency checking that cope with variability. To the best of our knowledge, this issue has not been extensively researched.

In this paper we propose *Safe composition* [12], the guarantee that *all* programs that can be composed according to the product line domain constraints are type safe (i.e. they do not have undefined references to structural elements such as classes, methods or fields), as a technique for consistency checking of MVM with variability. To achieve the same guarantee, conventional consistency checking approaches without support for variability would have to be applied to the models of each single member of a product line which is unfeasible even in small SPL as the number of potential feature combinations can grow exponentially.

We use a representative set of UML consistency rules and a feature composition technique to illustrate how safe composition can be used for consistency checking. However, other modeling artifacts, consistency rules and composition techniques can be used. Furthermore, we define a categorization scheme of consistency rules according to the number of artifact types they use and their relation with the composition technique. This categorization enables the identification of conditions where living with inconsistencies is acceptable (and even expected) and others where inconsistencies are not tolerable. To evaluate our approach, we developed a prototype tool and applied it to a case study.

2 Running Example

SPL approaches can be broadly categorized in two main groups depending on how they express variability in software artifacts. In *integrative* approaches, artifacts contain both the common and variable parts. Building a system means keeping the variable parts of the desired features in the artifacts while removing those parts belonging to unselected features [9,13,14]. In *compositional* approaches, variable parts are encapsulated in modular units which are put together according to the features selected for building a system [15,16,17,18][1]. There are several SPL methodologies that advocate a compositional approach, some of them use multiple views [10,20,15]. To illustrate our work, in this section we describe the core concepts of the compositional approach and the example we use throughout the paper.

2.1 Feature Oriented Software Development

Feature Oriented Software Development (FOSD) provides formalisms, methods, languages and tools for building variable, customizable and extensible software [15]. FOSD has been successfully used in several case studies [21,22]. FOSD advocates modularizing *features*, increments in program functionality [23], as the systems building blocks. At the heart of FOSD is a feature algebra that drives the (de)composition of software artifacts [24,16,25,26,27]. A *feature module* contains all the software artifacts, or parts thereof, required for implementing the feature. In other words, feature modules capture the multiple views of a feature.

In FOSD features are composed hierarchically starting from the root element of the corresponding models. Elements that have the same name and type at the same

[1] This classification appears with different names in the literature, for example *negative* or positive *variability* respectively[19].

hierarchical level are composed together, elements that do not have a corresponding matching element are copied along hierarchically. We illustrate FOSD composition with our running example as we proceed with the explanation of safe composition in next section. For further details please consult [16,28,29].

2.2 Video On Demand Example

The running example to illustrate our work is a hypothetical product line of video on demand systems. In this systems family, a video on demand (VOD) system can record and/or play videos and can be used with either TV sets or mobile phones. Thus our SPL example contains five features: `VOD`, `Play`, `Record`, `TV`, and `Mobile`. In FOSD, each feature is implemented in a feature module which contains all the required software artifacts for its realization. In our example, we use UML class, sequence and state diagrams [2]. Figure 1 shows the diagrams of the five features.

The diagrams of feature `VOD` are shown in Figure 1(a). The class diagram consists of three classes: `Service`, `Streamer`, and `Program`. These classes have some methods, a navigable association going from `Service` to `Streamer`, and one from `Streamer` to `Program`. The sequence diagram illustrates a call of method `select` in a `Service` object and a call of method `stream` from `Service` to `Streamer`. Lastly, the state machine shows two states in which a `Service` object can be in. After receiving a `select` method call a `Service` object initializes its information. Similarly, after receiving a `go` method call it starts streaming the video, and finally when it receives a `stop` it goes to a final state.

Figure 1(b) shows the diagrams of feature `Play`. This feature has a new class `Server`, and an association `manages` with class `Streamer` whose association end name is `handler`. The sequence diagram shows a message `go` from `Streamer` to `Service` objects, and a message `play` from `Service` object to itself. Notice here that message `go` is not defined in this feature but in feature `VOD`. The state machine diagram shows a new state `Frozen` with new actions `resume` and `pause`, as defined in the class diagram of this feature. Note again that action `go` is not defined in this feature but in feature `VOD`.

Figures 1(c)-(e) show the diagrams of features `Record`, `TV`, and `Mobile` respectively. Note for instance in feature `Record` depicted in 1(c) that messages `wait` is not a method of class `Streamer`. A similar case occurs in feature `TV` whose message `caption` is not a method of its class `Program`.

3 Detecting Inconsistencies with Safe Composition

Safe composition is the guarantee that programs composed from multi-view feature modules according to the product line domain constraints are type safe, i.e. they do not have undefined references to structural elements such as classes, methods or fields [12]. Current research on this topic has mainly focused on source code artifacts, particularly in FOSD extensions to Java-like languages. As pointed out by Thaker et al., the

[2] In practice, FOSD feature modules can contain any number of any artifact type (e.g. code, script files, grammars, etc.), for further details consult [15].

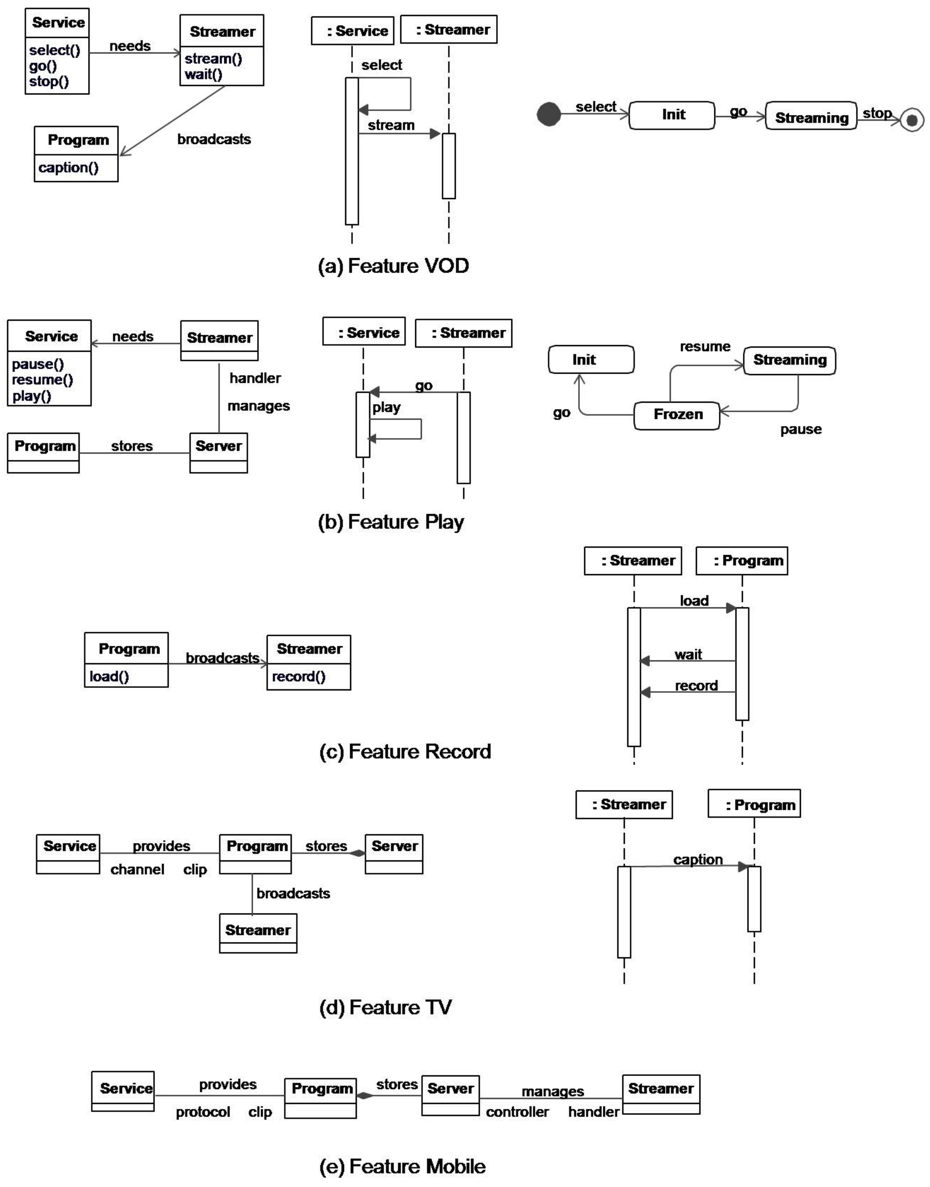


Fig. 1. Features in VOD SPL.

principles underlying safe composition can be also applied to other artifact types. Our work shows how safe composition applies to model artifacts by considering consistency rules that must be met by all composed program models. Safe composition can thus be used to detect inconsistencies not only on a single view (artifact type) but also amongst multiple views and most importantly within and across features.

3.1 Safe Composition Principles

Let us start by giving an example of an application of safe composition. Consider feature `Play` in Figure 1(b). In this feature, the sequence diagram shows a call to method `go` from `Streamer` to `Service`. Notice that this method is not defined in the class diagram of that feature. Safe composition verifies that *all* valid (according to domain constraints) combinations of features that include feature `Play` do also include another feature where `go` is defined (implementation constraints).

Safe composition is based on Czarnecki's et al. observation that implementation constraints should follow from domain constraints [30]. Let $\texttt{PL}_f$ denote the domain constraints and $\texttt{IMP}_f$ denote the implementation constraints of a consistency rule instance. Safe composition uses propositional logic to express and relate these two terms. Because we are interested in verifying that all members of the product line satisfy a given implementation constraint, the following formula should not be satisfiable:

$$\neg(\texttt{PL}_f \Rightarrow \texttt{IMP}_f) \tag{1}$$

In case it is satisfiable, it would mean that there is a member of the product line that does not meet constraint $\texttt{IMP}_f$. By using a *satisfiability (SAT)* solver, the violating feature configuration(s) can be identified. This is done for each instance of each implementation constraint we want to verify. We show next how the propositional formulas of $\texttt{PL}_f$ and $\texttt{IMP}_f$ are obtained.

3.2 Obtaining Domain Constraints from Feature Models

Feature models are a standard way to model the common and variable features of SPL and their relationships [31,32]. In these models, features are depicted as labeled boxes and are connected to other features to form a tree. A feature can be classified as: *mandatory* if it is part of a program whenever its parent feature is also part, and *optional* if it may or may not be part of a program whenever its parent feature is part. Mandatory features are denoted with filled circles while optional features are denoted with empty circles both at the child end of the feature relations denoted with lines. Features can be grouped into: *inclusive-or* relation whereby one or more features of the group can be selected, and *exclusive-or* relation where exactly one feature can be selected. These relations are depicted as filled arcs and empty arcs respectively.

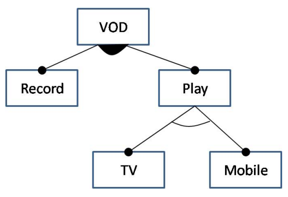


Fig. 2. Example of Feature Model

Figure 2 shows the feature model of our SPL of on-demand video recorders and players. In this hypothetical product line, the root feature VOD provides the basic functionality that the video systems offer. In FOSD, a feature in a feature model closely corresponds to a feature module. Recall that in our example SPL, a video on demand (VOD) system have features Record and Play in an inclusive-or relation, meaning that systems in this product line can: 1) record videos, 2) play videos, 3) record and play videos. Additionally, in those systems with playing capability, the systems either have television screens (TV) or screens of mobile devices (Mobile), corresponding to an exclusive-or relation.

Mapping of Feature Models to Propositional Logic. There exist extensive research on mapping feature models to propositional logic [33,34]. This mapping is summarized in Figure 3. Consider now for example the propositional formula for our model in Figure 2 shown in Equation 2. The first proposition comes from the fact that the root feature is always selected. The second proposition is the application of the inclusive-or rule for features Record and Play, while the last two propositions are the application of the exclusive-or for features TV and Mobile. Thus $\mathtt{PL}_f$ for our example is:

$$\begin{aligned}
&(VOD \ \Leftrightarrow \ true\ )\wedge \\
&(VOD \ \Leftrightarrow \ Record \vee Play\ )\wedge \\
&(TV \ \Leftrightarrow \ \neg Mobile \wedge Play\ )\wedge \\
&(Mobile \ \Leftrightarrow \neg TV \wedge Play)
\end{aligned} \tag{2}$$

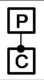
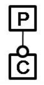
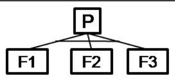
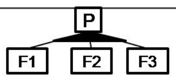
Name	Diagram Notation	Propositional Logic
Mandatory		$P \Leftrightarrow C$
Optional		$C \Rightarrow P$
Exclusive-Or		$(F1 \Leftrightarrow (\neg F2 \wedge \neg F3 \wedge P)) \wedge$ $(F2 \Leftrightarrow (\neg F1 \wedge \neg F3 \wedge P)) \wedge$ $(F3 \Leftrightarrow (\neg F1 \wedge \neg F2 \wedge P))$
Inclusive-Or		$P \Leftrightarrow F1 \vee F2 \vee F3$
Requires	Cross feature arrow	$A \Rightarrow B$
Excludes	Cross feature arrow	$A \Rightarrow \neg B \equiv \neg(A \wedge B)$

Fig. 3. Mapping a feature model to propositional logic.

3.3 Using Consistency Rules as Implementation Constraints

Consistency rules describe the semantic relationships that must hold amongst the different elements of the views. Consistency rules can be categorized according to the number of views they involve [35,36]:

- *Intra-view*: Exactly one view or artifact type is used by a rule.
- *Inter-view*: Multiple views or artifact types are used by a rule.

Our extension of safe composition for MVM adds another classification dimension that depends on the number of features involved:

- *Intra-feature*: Only one feature is needed to verify a constraint.
- *Inter-feature*: More than one feature are needed to verify a constraint.

It should be noted here that traditional consistency checking approaches fall into intra-feature category because they do not address variability issues in their models. Safe composition, on the other hand, allows us to extend the scope of current consistency checking approaches to address variability.

Consistency rules are usually specified as well-formedness rules [37], or emerge as standard best practices in certain domains [38,39]. In our previous work, we developed `UML/Analyzer`, a tool that incrementally checks consistency of UML class, sequence, and state machine diagrams. This tool checks over 30 distinct rules. For our work on safe composition, we selected seven representative structural rules from this set. Next we describe in detail each of the selected rules, their categorization, and use in safe composition with FOSD approach to model composition.

Rule 1. Method parameters should have different names. This rule specifies that in class diagrams the parameter names of methods in classes or interfaces must be unique. Clearly, this is an intra-view rule as it uses only class diagram views.

FOSD model composition. In FOSD, methods are matched based on their signature. This means that if two matching classes with two methods of different signatures are composed, then both methods are copied along to the result. In other words, FOSD does not support the addition of new parameters to methods. Thus, rule 1 is also an intra-feature rule because to validate this constraint it is only necessary to verify the feature where the method is defined. Because it is an intra-feature rule, safe composition does not apply as meeting this constraint is independent of how the feature being checked is composed. Notice however, that if a different composition approach were used that allows adding new parameters when methods are composed, then this rule would be classified as inter-feature.

Rule 2. An interface can only contain public operations. This rule specifies that the methods defined in an interface should have public visibility, i.e. accessible to any code that references it. Rule 2 can also be categorized as intra-view as it only uses class diagram views.

FOSD model composition. In FOSD, access modifiers are not composable. Thus, this rule is also an example of intra-feature rules because it is only required to verify the feature where the method is defined. Because it is an intra-feature rule, safe composition does not apply as meeting this constraint is independent of how the feature being checked is composed. Again, if a different composition approach were used that allows access modifiers to be composed, then this rule would be categorized as inter-feature.

Inter-feature Consistency Rules and Safe Composition. The properties denoted by inter-feature consistency rules make use of safe composition for two distinct purposes: 1) to assert the presence of a structural element that a feature requires, or 2) to assert the exclusion of a structural element that a feature conflicts with. We refer to these two kinds of rules as *requiring* and *conflicting* respectively.

Requiring rules. Let `F` be a feature that refers a model element `e` defined in another feature. For a system program that includes feature `F`, it must therefore also include at least one other feature $\texttt{Freq}_i$ where element `e` is defined. This is denoted in the following expression [3] :

$$IMP_f \ \equiv F \Rightarrow \bigvee_{i=1..k} Freq_i \tag{3}$$

By substituting $\texttt{IMP}_f$ in Equation 1, we obtain the logical expression that is passed to the SAT solver. In this case is the conjunction of all the terms of the features that define an element that feature `F` requires.

$$\neg(PL_f \Rightarrow IMP_f) \equiv PL_f \wedge F \bigwedge_{i=1..k} \neg Freq_i \tag{4}$$

When feature `F` requires an element that is not defined in any other features, that is expression $\bigvee\texttt{Freq}_i$ evaluates to `false`, it means that such element is not defined in the entire product line. This situation is clearly an error and renders unnecessary to verify this constraint with the SAT solver.

Conflicting rules. Let `F` be a feature that defines a model element `e`. A feature $\texttt{Fconf}_i$ conflicts with feature `F` if it has an element `d` which cannot be present in the same program where element `e` is also present. Put in different words, because of the conflict between elements `e` and `d`, if feature `F` is selected as part of a system program, then feature $\texttt{Fconf}_i$ cannot be selected. The propositional logic expression is thus:

$$IMP_f \ \equiv F \Rightarrow \ \neg(\bigvee_{i=1..k} Fconf_i) \tag{5}$$

By substituting $\texttt{IMP}_f$ in Equation 1, we obtain the logical expression that needs to be passed to the SAT solver. In this case, they are `k` disjunctions, one for each feature `F` has conflicting elements with. Thus it requires `k` calls to the SAT solver.

[3] For notational simplicity in the rest of the paper, we overload feature terms such as `F` or $\texttt{Freq}_i$ to mean propositional logic terms and the set of software artifacts. We make the distinctions explicit when necessary.

$$\neg(PL_f \Rightarrow IMP_f) \equiv \bigvee_{i=1..k} (PL_f \wedge F \wedge Fconf_i) \tag{6}$$

In the case where feature `F` has no conflicts with any other features, that is expression $\bigvee$ $\texttt{Fconf}_i$ evaluates to `false`, it is thus unnecesary to evaluate this constraint for element `e`.

Rule 3. Association ends must have a unique name within the association. This rule specifies that for any given association the names of its ends must not be repeated.

FOSD model composition. To illustrate this rule please consider features `Mobile` and `TV` in Figure 1(e) and Figure 1(d) respectively. FOSD composition of the corresponding class diagrams dictates to compose association `provides` between classes `Service` and `Program` because their names are the same and their types (association between `Service` and `Program`) also match. Notice however that the association end names of class `Server` are `channel` and `protocol`. This name mismatch violates this rule as the association end of `Server` has more than one name. This means that if feature `TV` is selected then feature `Mobile` cannot be selected because of this naming conflict. From this example, for FOSD composition technique, Rule 3 is then an example of intra-view and inter-feature rule.

Consider now feature `Mobile` Figure 1(e) feature in `Play` in Figure 1(b). FOSD dictates to compose association `manages` between `Server` and `Streamer` because their matching names and types. The association end of class `Streamer` is named `handler` on both features so no naming conflict arises on this class. The association end of class `Server` is named `controller` in feature `Mobile` and it is undefined in feature `Play`. In FOSD, the composed end name is `controller`. Therefore, in the composition of this association there is no conflict between features `Play` and `Mobile`.

Rule 3 is an example of a conflicting rule because of the naming conflicts in the end names. More formally, and using Equation (5), let `F` be a feature of the SPL that contains association `assoc` between classes `A` and `B` with respective association end names $\texttt{assoc.A}_{name}$ and $\texttt{assoc.B}_{name}$ [4]. A conflicting feature $\texttt{Fconf}_i$ is then defined as follows:

$\texttt{Fconf}_i$ contains association `assoc` between classes `A` and `B`,
and $[(\ \texttt{F.assoc.A}_{name} \neq \texttt{Fconf}_i\texttt{.assoc.A}_{name} \wedge \texttt{F.assoc.A}_{name} \neq \text{null} \wedge$
$\texttt{Fconf}_i\texttt{.assoc.A}_{name} \neq \text{null}\ ) \vee$
$(\ \texttt{F.assoc.B}_{name} \neq \texttt{Fconf}_i\texttt{.assoc.B}_{name} \wedge \texttt{F.assoc.B}_{name} \neq \text{null} \wedge$
$\texttt{Fconf}_i\texttt{.assoc.B}_{name} \neq \text{null}\ )]$

(7)

In words, this condition establishes that two features conflict in an association if they define non-null names that are different. Applying Equation (5) to the two examples just

[4] As notational convention we use qualified names to denote containment of elements and subscripts to refer to their values.

illustrated, we have that feature Mobile conflicts with feature TV but does not conflict with feature Play. Thus in this example $\texttt{IMP}_f \equiv \texttt{Mobile}\Rightarrow\neg\texttt{TV}$.

Rule 4. At most one association end may be an aggregation or composition. This rule specifies that any given association can only have either an aggregation or a composition but not both.

FOSD model composition. As an example, consider features TV and Mobile in Figure 1(d) and Figure 1(e) respectively. Both features have in their class diagrams an association stores between Program and Server. Notice however that the composition lies at different sides of the association. Thus, selecting both features together violates this rule. Clearly because this rule involves only class diagrams and two features it is and example of intra-view and inter-feature rule. Furthermore, it is a conflicting rule because the existence of an aggregation or composition in one feature excludes the existence of another aggregation or composition at another feature.

More formally, and using Equation (5), let F be a feature of the SPL that contains association assoc between classes A and B. A conflicting feature $\texttt{Fconf}_i$ is then defined as follows:

$$\begin{aligned}&\texttt{Fconf}_i \text{ contains association } \texttt{assoc} \text{ between classes } \texttt{A} \text{ and } \texttt{B},\\ &\text{and } [(\texttt{F.assoc}_{type}\text{=aggregation } \vee \texttt{F.assoc}_{type}\text{=composition })\wedge\\ &\quad(\texttt{Fconf}_i\texttt{.assoc}_{type}\text{=aggregation } \vee \texttt{Fconf}_i\texttt{.assoc}_{type}\text{=composition })]\end{aligned} \tag{8}$$

In words, this condition establishes that two features conflict if an association defines either an aggregation or composition in feature F and on the same association but in feature $\texttt{Fconf}_i$ there is either an aggregation or a composition. Applying Equation (5) give us $\texttt{IMP}_f \equiv \texttt{Mobile}\Rightarrow\neg\texttt{TV}$.

Rule 5. Message action must be defined as an operation in receiver's class. This rule specifies that in a sequence diagram a message action should have a corresponding operation defined in the class diagram of the message receiver's class.

FOSD model composition. As an example for this rule, the sequence diagram of feature Play in Figure 1(b) refers to method go but feature Play does not define it in its class diagram. Thus, every time that feature Play is selected, another feature that defines method go must also be selected. In this example, the class diagram of feature VOD in Figure 1(a) provides such definition and can thus be selected when feature Play is selected. This rule is then an inter-feature rule, and because it involves class diagrams and sequence diagrams an inter-view rule. Furthermore, it is a requiring rule because the existence of a message action demands the existence of a method that defines it in the target class.

More formally, and using Equation (3), Let F be a feature of the SPL that contains message action msg with receiver's class Cls. A requiring feature $\texttt{Freq}_i$ is then defined as follows:

$$\texttt{Freq}_i \text{ contains method } \texttt{msg} \text{ in class } \texttt{Cls} \text{ in a class diagram} \tag{9}$$

In words, this condition establishes that a feature whose sequence diagram references a method requires the definition of that method in the class diagram of another feature. Applying Equation (3) thus give us $\texttt{IMP}_f \equiv \texttt{Play}\Rightarrow\texttt{VOD}$.

Rule 6. State machine action must be defined as an operation in owner's class. This rule specifies that in a state machine associated to a class the actions should be operations defined in the class diagram of such class.

FOSD model composition. This rule is similar to Rule 5. Consider now the state machine diagram of feature `Play` in Figure 1(b) that has transition method `go`, but again it is not defined in the class diagram of this feature. Thus, whenever feature `Play` is selected there must be another feature where method `go` is defined. As we have seen, this method is defined in feature `VOD` in Figure 1(a). Because this rule involves class and state machine diagrams in more that one feature, it is an example of inter-view and inter-feature rule. Furthermore, it is a requiring rule because the existence of an action in a state machine diagram requires its definition in another feature's class diagram.

More formally, and using Equation (3), Let `F` be a feature of the SPL that contains a state machine action `msg`. Let `F` be a feature module of the SPL that has transition method `msg` in state machine of class `Cls`. A requiring feature $\texttt{Freq}_i$ is then defined as follows:

$$\texttt{Freq}_i \text{ contains method } \texttt{msg} \text{ defined in class } \texttt{Cls} \tag{10}$$

In words, this condition establishes that a feature that has a state machine diagram that references a method requires the definition of that method in the class diagram of another feature. Applying Equation (3) thus give us $\texttt{IMP}_f \equiv \texttt{Play} \Rightarrow \texttt{VOD}$.

Rule 7. Calling direction of message must match calling direction of association. This rule specifies that if a sequence diagram has a message going from an object of class `A` to an object of class `B` then in the class diagram the relationship between both classes should be navigable in that direction.

FOSD model composition. As an example of this rule, consider feature `Record` in Figure 1(c) that has message `load` from `Streamer` to `Program`. Notice however that in this feature the direction of the association between these two classes is the opposite. Therefore, if feature `Record` is included there must be another feature that defines a navigable association from `Streamer` to `Program`, in our case feature `VOD` in Figure 1(a). Because this rule involves sequence and class diagrams on multiple feature it is an example of inter-view and inter-feature rule. This rule is requiring because the existence of a message in the sequence diagram demands the existence of an association navigable in the direction of the message in a class diagram.

More formally, and using Equation (3), Let `F` be a feature of the SPL that contains a message going from an object of class `Src` to an object of class `Tgt`. A requiring feature $\texttt{Freq}_i$ is then defined as follows:

$$\texttt{Freq}_i \text{ contains navigable association from class } \texttt{Src} \text{ to class } \texttt{Tgt} \tag{11}$$

Applying Equation (3) thus give us $\texttt{IMP}_f \equiv \texttt{Record} \Rightarrow \texttt{VOD}$.

3.4 Analysis

This section summarizes the main insights gained with our application of safe composition for MVM consistency checking.

Safe composition granularity. Table 1 shows the classification of our rules along the two dimensions. Rule 1 and Rule 2 highlight the fact that not all consistency rules are applicable to safe composition. The distinctive characteristic of both rules is that their level of granularity, method parameter names for Rule 1 and access modifiers for Rule 2, falls below the granularity level of FOSD composition. In other words, FOSD composes elements such as methods or classes (coarser granularity) but not their nested elements (finer granularity). This observation is summarized in the following principle:

***Principle of Safe Composition Granularity**: Safe composition is applicable to consistency rules that operate at the granularity level supported by the model composition mechanism.*

Tolerable and intolerable inconsistencies. Our categorization of consistency rules along two dimensions allows us to further distinguish two types of inconsistencies from a compositional perspective. We call *intolerable* inconsistencies those that arise from violations to rules that are both intra-view and intra-feature because they render features unfit for composition. On the other hand, we call *tolerable* those inconsistencies arising from violations to inter-feature rules because it is expected that they be fixed by composition with other features. Finally, it should be noted that in the sample of consistency rules we analyzed there was no rule categorized as inter-view and intra-feature. In the case of FOSD, this follows in part from the fact that feature composition can add elements to any views.

	Intra-view	Inter-view
Intra-feature	Rule 1 Rule 2	
Inter-feature	Rule 3 Rule 4	Rule 5 Rule 6 Rule 7

Table 1. Classification of consistency rules

Multi-feature consistency rules. The inter-feature rules we presented involved only two features. Our consistency checking tool `UML/Analyzer` uses 34 consistency rules, out of those there are only two rules that can involve more than two features. One of such rules checks that circular inheritance does not occur. A solution would be along the lines proposed by Thaker et al. that would collect the inheritance information by succesively composing all features and relying on the monotonicity of the composition detect the circular references [12]. The implementation of this alternative and its evaluation are part of our future work.

More expressive formal representation and automated rule generation. Currently, our rules have been manually implemented following their OCL description in relation to the FOSD approach for model composition. However, we believe that some (if not all) the implementation could be generated directly from formal rule specifications and the underlying semantics used for model composition. This is a topic of our future research.

3.5 Evaluation

We used the *Graph Product Line (GPL)* [40] as case study for our approach. The features of this product line are basic graph algorithms and data structures. A GPL program is a combination of different algorithms implemented on different data structures. There are implementations of GPL available in several programming languages. The models used in our study were manually drawn in the Eclipse UML editor from a Java version of GPL.

We implemented a prototype tool that uses EMF to parse and gather information from the EMF models [41], and PicoSAT SAT solver to test for satisfiability [42]. Despite of being a short example, we found a total of 298 distinct instances of consistencies rules. When mapped to propositional logic, the FODA model of GPL consists of 22 domain constraints: 5 mandatory, 2 optional, 2 exclusive-or, 1 inclusive-or, 1 excludes, and 11 requires. These domain constraints amounted to 39 propositional clauses when normalized to CNF for use by PicoSAT.

Our experiments showed that the time taken to evaluate consistency rule instances by the SAT solver was negligible (in the magnitud of nanoseconds when run on an Intel Core-Duo at 2.8 GHz) as the number of clauses involved and the number of variables (one for each of the nineteen features) are of relatively small size for what SAT solvers such as PicoSAT can effectively handle. Though encouraging results, the scalability and performance of our approach needs to be more extensively validated with more complex examples of SPL that contain larger models on which to validate more consistency rules instances. Doing that is part of our future work.

4 Related Work

There is a significant amount of related literature. We focus on the research that most closely relates to our work and divide them in three categories.

FOSD Model Composition. Our previous work has shown the applicability of an algebraic representation to describe model composition in *use case slices*, an Aspect-Oriented modeling techniques based on UML diagrams [43], when used for SPL modeling [44]. Work by Umapathy developed basic composition of UML diagrams using `XAK`, a FOSD composer of XML-based artifacts [45]. Our recent work has shown the applicability of rewriting technologies for composing UML class diagrams exploiting the native support of algebraic properties of operators in Maude [28]. Work by Apel et al. uses superimposition to compose simple UML diagrams that are treated as trees [29]. These technologies are different alternatives to support model composition for FOSD.

Models and Software Product Lines. Product Line UML-based Software engineering (PLUS) [9] is a method that brings FODA ideas to UML. PLUS uses features throughout the entire product line development process, however their boundaries are lost in the model diagrams. In other words, most of the diagrams in this approach show elements that either belong to all the product line or to those of a particular product configuration (i.e. a selected set of features). This is an example of the integrative approach to variability management. Jayaraman and Whittle have developed a compositional approach to PLUS whereby models are modularized in *feature slices*, collections of fragments of UML diagrams, that are composed via graph transformations [10]. To the best of our knowledge their work does not make any provisions for consistency checking of the composed feature slices.

Safe Composition and Well-formedness. Work by Czarnecki et. al uses OCL constraints to specify and verify well-formedness in model templates. In contrast to our work, this is an integrative approach for variability modeling [30]. Work by Kästner et al. follows an integrative approach whereby program elements are annotated with distinct colors to visually indicate the features they belong to [46]. It enforces two simple structural rules to guarantee syntactic correctness of the programs derived.

5 Conclusions and Future Work

In this paper we showed how safe composition principles can be applied for MVM consistency checking in the context of SPL. We used a representative set of UML consistency rules as illustration of our approach. These rules were categorized according to the number of views and their relation to feature composition. Though our work is presented in the context of UML and FOSD, our results can be mapped to other modeling artifacts, constraints, and composition approaches.

We implemented a prototype tool and used it in a case study to evaluate the feasibility of our approach. Performance and scalability were not an issue for this case study. However, these aspects need further assessment with larger and more complex product lines as well as considering more consistency rules. Such an assessment is part of our future work. FOSD composition has been defined as a monotonic operation. Recent work by Kuhlemann relaxes this requirement to consider non-monotonic composition [47]. We plan to investigate alternatives for non-monotonic model composition along the lines of this work. SAT solvers are just one technology used for consistency checking. Because of the incremental nature of feature composition, we will explore the applicability of incremental consistency approaches, like `UMLAnalyzer`[38,39], to safe composition .

Acknowledgments. We thank Alexander Nöhrer for his help with PicoSAT solver, Leticia Montalvillo for drawing the diagrams used in the case study, and Maider Azanza for her reviews of early drafts. This research was partiually funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965.

References

1. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering **2**(1) (1992) 31–57

2. Nuseibeh, B., Kramer, J., Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specification. IEEE Trans. Software Eng. **20**(10) (1994) 760–773
3. Finkelstein, A., Gabbay, D.M., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multperspective specifications. IEEE Trans. Software Eng. **20**(8) (1994) 569–578
4. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw., Pract. Exper. **35**(8) (2005) 705–754
5. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE TSE **30**(6) (2004)
6. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
7. Pohl, K., Bockle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)
8. van d. Linden, F.J., Schimd, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer (2007)
9. Gomaa, H.: Designing Software Product Lines with UML. From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)
10. Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In: MoDELS. (2007)
11. Käkölä, T., Dueñas, J.C., eds.: Software Product Lines - Research Issues in Engineering and Management. Springer (2006)
12. Thaker, S., Batory, D.S., Kitchin, D., Cook, W.R.: Safe composition of product lines. In Consel, C., Lawall, J.L., eds.: GPCE, ACM (2007) 95–104
13. Gomaa, H., Olimpiew, E.M.: Managing variability in reusable requirement models for software product lines. In Mei, H., ed.: ICSR. Volume 5030 of Lecture Notes in Computer Science., Springer (2008) 182–185
14. Zhang, H., Jarzabek, S.: Xvcl: a mechanism for handling variants in software product lines. Sci. Comput. Program. **53**(3) (Hongyu Zhang and Stanislaw Jarzabek) 381–407
15. Batory, D.: AHEAD Tool Suite (2008) http://www.cs.utexas.edu/users/schwartz/ATS.html.
16. Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Software Eng. **30**(6) (2004) 355–371
17. Mezini, M., Ostermann, K.: Variability management with feature-oriented programming and aspects. In Taylor, R.N., Dwyer, M.B., eds.: SIGSOFT FSE, ACM (2004) 127–136
18. Groher, I., Völter, M.: Using aspects to model product line variability. In Thiel, S., Pohl, K., eds.: SPLC (2), Lero Int. Science Centre, University of Limerick, Ireland (2008) 89–95
19. Groher, I., Völter, M.: Aspect-oriented model-driven software product line engineering. T. Aspect-Oriented Software Development VI **6** (2009) 111–152
20. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design. The Theme Approach. Addison-Wesley (2005)
21. Trujillo, S., Batory, D., Diaz, O.: Feature Oriented Model Driven Development: A Case Study for Portlets. In: ICSE. (2007)
22. Batory, D., O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. ACM Transactions on Software Engineering and Methodology (TOSEM) **1**(4) (1992) 355–398
23. Zave, P.: Faq sheet on feature interaction http://www.research.att.com/ pamela/faq.html.
24. Lopez-Herrejon, R.E.: Understanding Feature Modularity. PhD thesis, Department of Computer Sciences, The University of Texas at Austin (2006)
25. Batory, D.S., Lopez-Herrejon, R.E., Martin, J.P.: Generating product-lines of product-families. In: ASE, IEEE Computer Society (2002) 81–92
26. Lopez-Herrejon, R.E., Batory, D.S., Lengauer, C.: A disciplined approach to aspect composition. In Hatcliff, J., Tip, F., eds.: PEPM, ACM (2006) 68–77

27. Batory, D.S.: Using modern mathematics as an fosd modeling language. In Smaragdakis, Y., Siek, J.G., eds.: GPCE, ACM (2008) 35–44
28. Lopez-Herrejon, R.E., Rivera, J.E.: Realizing feature oriented software development with equational logic: An exploratory study. In Vallecillo, A., Sagardui, G., eds.: JISBD. (2009) 269–274
29. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model superimposition in software product lines. In Paige, R.F., ed.: ICMT. Volume 5563 of Lecture Notes in Computer Science., Springer (2009) 4–19
30. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L., eds.: GPCE, ACM (2006) 211–220
31. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
32. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
33. Benavides, D., Segura, S., Trinidad, P., Cortés, A.R.: Fama: Tooling a framework for the automated analysis of feature models. In Pohl, K., Heymans, P., Kang, K.C., Metzger, A., eds.: VaMoS. Volume 2007-01 of Lero Technical Report. (2007) 129–134
34. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Proceedings of the International Software Product Line Conference (SPLC). (2005) 7–20
35. Lucas, F.J., Molina, F., Álvarez, J.A.T.: A systematic review of uml model consistency management. Information & Software Technology **51**(12) (2009) 1631–1645
36. Usman, M., Nadeem, A., Kim, T.H., Cho, E.S.: A survey of consistency checking techniques for uml models. In: Advanced Software Engineering and Its Applications, 2008. ASEA 2008. (2008) 57–62
37. OMG: Uml infrastructure specification v2.2 (2009)
38. Egyed, A.: Fixing inconsistencies in uml design models. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 292–301
39. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in uml design models. In: ASE, IEEE (2008) 99–108
40. Lopez-Herrejon, R.E., Batory, D.S.: A standard problem for evaluating product-line methodologies. In Bosch, J., ed.: GCSE. Volume 2186 of Lecture Notes in Computer Science., Springer (2001) 10–24
41. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. 2 edn. Addison-Wesley Professional (2008)
42. Biere, A.: Picosat http://fmv.jku.at/picosat/.
43. Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley (2004)
44. Lopez-Herrejon, R., Batory, D.: Modeling Features in Aspect-Based Product Lines with Use Case Slices: An Exploratory Case Study. In: Workshops and Symposia at MoDELS. (2006)
45. Umapathy, S.: Extension of UML models to Support Feature Modularization of Software Product Lines. Master's thesis, Computing Laboratory, University of Oxford (2007)
46. Kästner, C., Apel, S., Trujillo, S., Kuhlemann, M., Batory, D.S.: Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In Oriol, M., Meyer, B., eds.: TOOLS (47). Volume 33 of Lecture Notes in Business Information Processing., Springer (2009) 175–194
47. Kuhlemann, M., Batory, D.S., Kästner, C.: Safe composition of non-monotonic features. In Siek, J.G., 0002, B.F., eds.: GPCE, ACM (2009) 177–186