Modeling Variability from Requirements to Runtime

Mathieu Acher[1], Philippe Collet[1], Philippe Lahire[1]
[1]*Université de Nice Sophia Antipolis*
I3S - CNRS UMR 6070
06903 Sophia Antipolis Cedex, France
{acher,collet,lahire}@i3s.unice.fr

Sabine Moisan[2], Jean-Paul Rigault[1,2]
[2]*INRIA Sophia Antipolis Mediterranée, France*
06902 Sophia Antipolis Cedex, France
{moisan,jpr}@sophia.inria.fr

***Abstract*—In software product line (SPL) engineering, a software configuration can be obtained through a valid selection of features represented in a feature model (FM). With a strong separation between requirements and reusable components and a deep impact of high level choices on technical parts, determining and configuring an well-adapted software configuration is a long, cumbersome and error-prone activity. This paper presents a modeling process in which variability sources are separated in different FMs and inter-related by propositional constraints while consistency checking and propagation of variability choices are automated. We show how the variability requirements can be expressed and then refined at design time so that the set of valid software configurations to be considered at runtime may be highly reduced. Software tools support the approach and some experimentations on a video surveillance SPL are also reported.**

***Keywords*-Software Product Lines ; Feature Models ; Requirement specification ; Adaptive systems**

I. Introduction

Software Product Line (SPL) approaches promise to improve time-to-market, deployment cost, and quality of application development. SPL engineering usually promotes systematic reuse through two complementary processes [19]. First, *domain engineering* models the reusable platform on top of which the different products are built and makes explicit both reusable points (i.e., common parts of products) and variation points (i.e., differences between products). Second, *application engineering* makes it possible to derive, as quickly and efficiently as possible, an appropriate product from the platform. The idea behind this approach to SPL engineering is that the investments required to develop the reusable artifacts during domain engineering are outweighed by the benefits in deriving the individual products during application engineering [10].

Nevertheless, there are several software areas in which *i)* there is a strong separation between high-level requirements and reusable implementation-oriented components and *ii)* decisions of experts at a very high level deeply impact the realization on the platform. In these contexts, realizing and using a SPL is a very cumbersome process since deriving individual products from shared software assets is a time-consuming and expensive activity [10], [13], [24], [26].

As a representative case study, we have identified such issues in video surveillance product lines [1], [3], in which many design decisions, both at specification and implementation levels, require broad expertise and involve several stakeholders. A major problem [3] lies in the configuration phase of the video processing chain from the specific requirements (e.g., the positioning of cameras and the type of lights highly impact many parameters of several tasks in the chain). This task can last from one week to several months in order to deploy the software solution. First, reasoning directly at the level of software components is far from obvious, even for an expert, since selecting among the software choices must respect a large number of constraints (e.g., night surveillance in natural light necessitates specific algorithm). Second, an unique characteristic of dynamic, self-adaptive software systems, such as video surveillance systems, is that, at *runtime*, only a small part of the model related to requirements has to be kept – for example, features related to the context – so that this sub-model can be efficiently used to pilot self-adaptation mechanisms (as we have shown in [1]). Although technical know-how may help to reduce possible choices, it is difficult to determine which combination of features remain valid at runtime according to the various possible *contexts* (e.g., lighting conditions, information on the objects to recognize).

To tackle this problem, we follow a model-based approach in which both variability spaces are described through two feature models (FMs) ; the first one describes the domain and the related requirements, while the other one is an abstract representation of the code. The relationships between variants are described as propositional rules relating features either in the same model or across models. Such an approach is similar to other related work that promotes the (systematic) use of FMs [11], [13], [16], [17], [24].

The contribution of this paper is to propose a *modeling process* in which the FMs are systematically used and step-wise specialized, from requirements to runtime. Consequently, from a domain expert specification, the possible configurations space is highly reduced, as no more valid platform configurations can be automatically removed by transformation. We present techniques, based on propositional logic and fully supported by a tool, to assist SPL

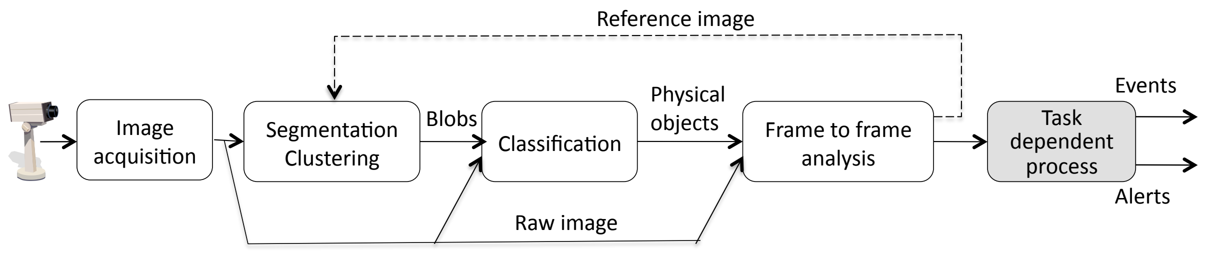


Figure 1: A simplified video surveillance processing chain

practitioners in the deployment of video surveillance processing chains while ensuring the end-to-end consistency of the manipulated models.

This paper first motivates (Section II) the addressed issues by using a running example related to the video surveillance domain. Section III describes our approach, which explicitly separates the variability models and aims at transforming application requirements into a software configuration. Section IV presents the resulting process and shows that the reasoning operations needed can be encoded in propositional logic. Section V describes tool support, some validation experiments on the video surveillance case study and discusses related work, while Section VI concludes this paper.

II. Motivation

Taking Video Surveillance (VS) as a representative domain of the targeted software areas, we now determine our motivating issues. The purpose of VS is to analyze image sequences to detect interesting situations or events. Depending on the application, the corresponding results may be stored for future processing or may raise alerts to human observers. There are several kinds of VS tasks according to the situations to be recognized: detecting intrusion, counting objects or events, tracking people, animals or vehicle, recognizing specific scenarios... Apart from these functional characteristics, a VS task also sports non functional properties, such as quality of service: typical criteria are robustness, characterized by the number of false positive and negative detections, response time, or recognition accuracy.

Moreover, each kind of task has to be executed in a particular context. This context includes many different elements: information on the objects to recognize (size, color, texture...), description and topography of the scene under surveillance, nature and position of the sensors (especially video cameras), lighting conditions... These elements may be related together, for example, an indoor scene implies a particular lighting. They are also loosely related to the task to perform since different contexts are possible for the same functionality. For instance, intrusion detection may concern people entering a warehouse as well as pests landing on crop leaves.

The number of different tasks, the complexity of contextual information, and the relationships among them induce many possible variants at the specification level, especially on the context side. The first activity of a video surveillance application designer is to sort out these variants to precisely specify the function to realize and its context. Then the designer has to map this specification to software components that implement the needed algorithms.

At the implementation level, a typical VS processing chain (Figure 1) starts with image acquisition, then segmentation of the acquired images, clustering, to group image regions into blobs, classification of possible objects, and tracking these objects from one frame to the other. The final steps depend on the precise task. Additional steps may be introduced, such as reference image updating (if segmentation steps need it), data fusion (in case of multiple cameras) or even scenario recognition. All steps correspond to software components that the designer must correctly assemble to obtain a processing chain. Moreover, for each step, many variants exist, along different dimensions. For instance, there are various classification algorithms with different ranges of parameters, using different geometrical models of physical objects, with different merge and split strategies to identify relevant image blobs. The situation is similar for the other algorithms.

The domain of VS is now mature enough to provide components covering all classical steps and to provide unifying and stable frameworks to compose them, such as our platform [4]. However these frameworks can only be mastered by video analysis specialists. The specification phase is not supported by tools, and component assembly seldom is. Thus designers work directly at the component assembly level, following an informal view of the specification. As a consequence, there is no guarantee that specification and implementation are consistent and that internal dependencies are respected, at specification as well as implementation level; moreover, tracing between specification and implementation is hard. Hence, designing a video surveillance system faces typical difficulties of current information systems in which domain and application engineering must be cleverly combined. Indeed this requires to cope with multiple sources of variability, both on the task specification side and on the implementation one. To bridge this gap, designers must have an extensive experience of the application domain as well as a deep knowledge of the software components. Yet the process remains tedious, error-prone, and possibly long.

III. Modeling Variability

A. Feature Modeling

Modeling variability has been explored and used in several domains. One of the most practical techniques is feature modeling which aims at representing the common and variable *features* (or concepts) of a family. Several definitions of feature appear in the literature, ranging from "anything users or client programs might want to control about a concept" [7], "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems" [16] to "an increment in product functionality" [5]. Therefore, feature modeling is not only relevant to requirement engineering but it can also be applied to design or code levels (e.g., see [14]).

FMs hierarchically structure domain concepts into multiple levels of increasing detail. When decomposing a feature into subfeatures, the subfeatures may be optional or mandatory, or may form Xor-, Or-, or And-groups[1]. FMs describe the variability and the commonality of features and represent a set of valid *configurations*. A valid configuration is obtained by selecting features while respecting the parent-child and decomposition semantics: *i)* if a feature is selected, so should be its parent; *ii)* if a parent is selected, all the mandatory subfeatures of its And group, exactly one of its Xor-group(s), and at least one of its Or group(s) must be selected; *iii)* internal constraints relating features (e.g., feature dependencies) in different subtrees must hold.

Definition 1 (Feature Model, Configuration): A feature model defines a set of valid configurations, where a configuration is defined as a set of features selected. $[\![FM_i]\!]$ denotes the set of valid configurations of a feature model FM_i.

B. Variability Models

1) Domain and Software Variability: In the VS case, we have multiple variability factors, both for specifying an application and for describing the software platform. On the platform side, stakeholders manipulate solution-oriented artifacts. There, variability is entirely related to the software and alludes to "the ability of a system to be efficiently extended, changed, customized or configured for use in a particular context" [21]. The software variability is expressed in a dedicated FM, called the *PlatForm Configuration* (PFC) model representing a view of implementation modules provided by the software platform. For deriving an actual product, we advocate the use of domain knowledge which contains relevant information to reason about and to select among variants at a higher level of abstraction. The domain variability is expressed through the *Video Surveillance Application Requirement* (VSAR) model which comprises, in our case, the task specification, the scene context, the object of interests, the Quality of Service (QoS) – see below.

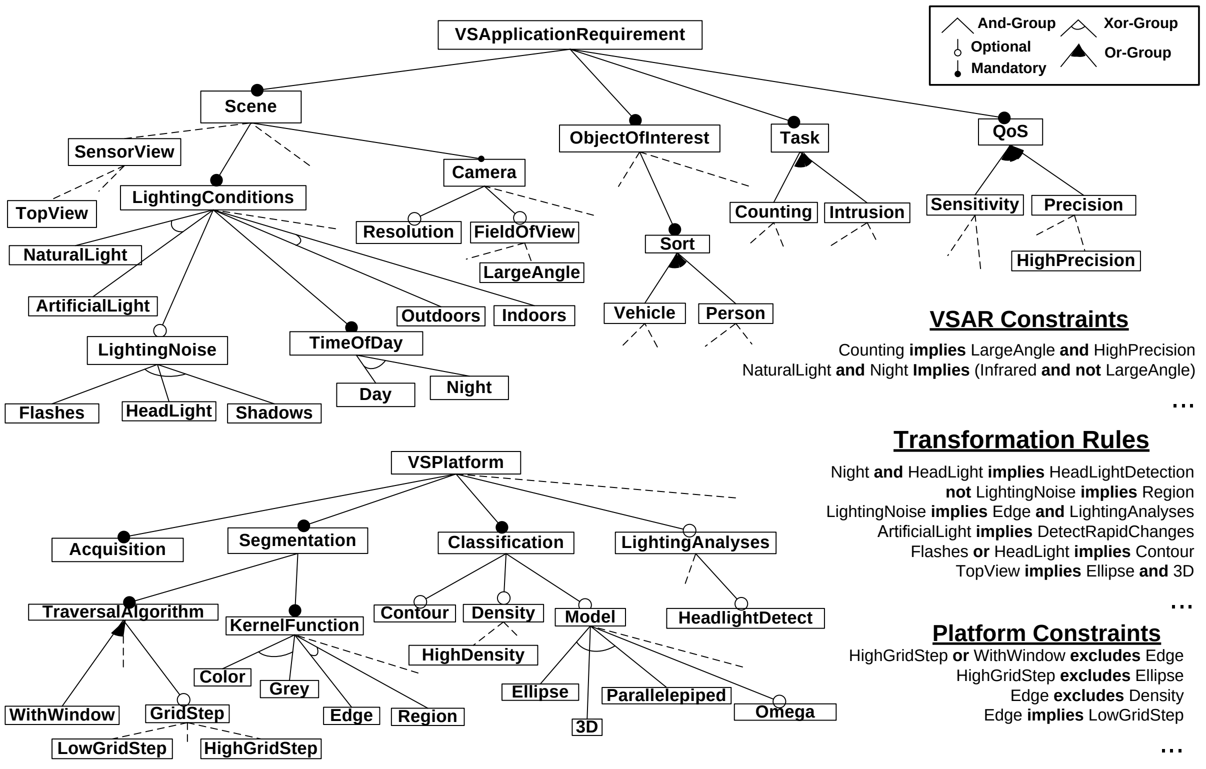
[1]We consider only FMs in their basic, propositional form [9]. We do not consider other notations and richer formalisms, for example, cardinality-based FMs [8] or FMs with attributes [6]. See Section V-C for a discussion.

2) Separation of Concerns: One option would be to transfer VSAR variability (e.g., all the possible variations in the context) into the PFC model, thus concentrating variability in one unique model. A major drawback is then to swamp domain variability concerns with the mass of platform details. Our approach is rather to separate variability concerns into two interrelated FMs: the VSAR FM represents the experts' business knowledge while the PFC one deals with the platform implementation. This separation of concerns offers several benefits. Each variability model addresses a different level of expertise. On the requirement side, users can follow their usual practices, use the domain vocabulary, and the specific definition of variation types. Confining the variability in a dedicated *space* thus improves the modeling process. During the application engineering process, users can take decisions only related to their know-how and domain. The impact of a modification in the platform model (e.g., code module added) or in the VSAR model (e.g., object of interest added) is clearly localized. Co-evolution and maintenance of both variabilities are facilitated.

We now describe more precisely the two VSAR and PFC models, together with transformation rules relating features of the two FMs.

3) VSAR *model:* Figure 2 (upper part) shows an excerpt of the FM corresponding to the VSAR side. This model describes the relevant concepts and features from stakeholders' point of view, in a way that is natural in the VS domain. To enforce separation of concerns, we identified four top level features. The Task feature expresses the precise function to perform. QoS corresponds to the non-functional requirements, especially those related to quality of service. Then, we need to define the Objects of interest to be detected, together with their properties. Finally, Scene context is the feature with the largest sub-tree; it describes the scene itself (its topography, the nature and location of sensors) and many other environmental properties (only some of them are shown on the figure). In this model, the (sub-)features are not independent, for example, selecting a feature may impact other choices. Thus, we have enriched the feature models by adding internal constraints to cope with relations local to a model. So far, we have identified two kinds of constraints. Choosing one feature may **imply** or **exclude** to select another specific one. For example, if feature Counting is selected, this implies high precision and thus a Field Of View which is a Large Angle (among others). The intra-constraints in the VSAR model show a strong dependency between the four top level features which confirms the need for integration into the same variability model.

4) PFC *model:* All steps of the VS processing chain correspond to software components that the designer must correctly assemble to obtain a coherent processing chain. A mandatory task is to acquire images. Then, for each



VSAR Constraints

Counting **implies** LargeAngle **and** HighPrecision
NaturalLight **and** Night **Implies** (Infrared **and not** LargeAngle)
...

Transformation Rules

Night **and** HeadLight **implies** HeadLightDetection
not LightingNoise **implies** Region
LightingNoise **implies** Edge **and** LightingAnalyses
ArtificialLight **implies** DetectRapidChanges
Flashes **or** HeadLight **implies** Contour
TopView **implies** Ellipse **and** 3D
...

Platform Constraints

HighGridStep **or** WithWindow **excludes** Edge
HighGridStep **excludes** Ellipse
Edge **excludes** Density
Edge **implies** LowGridStep
...

Figure 2: VSAR, PFC feature models and transformation rules

step, many variants (e.g., alternative algorithms) exist. The platform FM describes the different software components of the platform, their parameters and their assembly constraints. Figure 2 displays a highly simplified form of the corresponding model. As shown, the top level features mainly correspond to the different steps of the processing chain. The figure focuses on the segmentation step. Similarly to the previous model, we also need to introduce internal constraints. They have the same form as before. For instance, Edge segmentation **implies** a thin image discretization, thus a Low Grid Step.

5) Transformation Rules: Besides internal constraints, there are other constraints across models. These inter-model constraints make dependencies and interactions between features explicit by mapping the task specification to the software implementation. More importantly, such constraints prevent forbidden combinations of features, thus dramatically reducing the configuration sets. In our approach, these constraints correspond to model transformations from the VSAR model to the PFC model. They allow deriving automatically, or semi-automatically, a suitable processing chain from an application specification (see next Section). For instance, a Top View **implies** the use of both an Ellipse and a 3D model to describe persons.

IV. From Requirements to Deployment and Runtime

The platform FM compactly represents the set of software configurations available for each category of components that can be activated to achieve the tasks of the processing chain. This is highly desirable for VS applications to cope with possible runtime change of implementation triggered by context variations. The goal is at runtime to deploy a VS processing chain able to adapt its configuration according to a valid contextual information. A key idea is that only a *specialized* (see Definition 2) VSAR FM is needed since the context features that may influence the runtime execution of the system is most of the time only a part of the VSAR FM.

A. Process

In Figure 3 we present a process together with sound formal basis to support rigorous reasoning and assist stakeholders until the application is deployed. In this Figure, we show also the behaviour when adaptations are performed at runtime and the operations required to ensure systematic consistency and end-to-end transformation. The two stakeholders (VS expert and software application engineer) of the approach interact with the FMs during modeling (see ❶), specialization (see ❸ and ❹) and transformation (see ❺).

During the elaboration activity, the two FMs and rules are designed by the VS expert and the software application engineer. A precondition to the configuration process is that FMs and rules are both consistent individually as well as considered altogether. It means that each FM contains at least one valid configuration and that transformation rules have no contradictions. It is possible that each FM is consistent individually whereas putting them together and applying rules leads to inconsistencies. For instance, a transformation rule may express that a *core* feature (see Definition 3) of the VSAR model excludes another core feature in the PFC model. Hence, consistency checking must also be achieved globally ❷.

Definition 2 (Specialization): Let FM_1 and FM_2 be two FMs. FM_2 is a specialization of FM_1 if and only if all the configurations admissible by FM_2 are also admissible by FM_1, that is, $[\![FM_2]\!] \subset [\![FM_1]\!]$

Definition 3 (Core and dead features): A *core* feature is included in every configuration of an FM. A *dead* feature does not appear in any valid configuration of an FM.

Once checked consistent, the two FMs are then edited, independently, during the *specialization* process. The specialization process conducted by the VS expert consists in a sequence of edit operations on the VSAR model, for example, the removal of a feature. The VS expert produces a new VSAR model, say Y, so that the original VSAR model, say X, is a specialization of Y if the set of configurations in X is a subset of that in Y. The VS expert reasons over a complex set of constraints and typically removes/adds the desired features over a series of steps, rather than in a single iteration, in a multi-stage process.

It is thus important, not to say mandatory, to ensure that each edit operation is permitted during the entire specialization process ❸. Once checked, the specialization edits achieved on VSAR or PFC can lead to the automatic simplification of both FMs. For instance, when users decide that a feature of an Xor-group is necessary included in any configuration of an FM, the other features of the Xor-group cannot be selected and are no longer included in the FM. An automated support for *i)* systematic specialization checking and *ii)* propagation of edits allows the VS expert to prevent unauthorized choices and better understand consequences of his/her decision at each step. At the end of the specialization process, the VS expert gets a specialized VSAR FM where some parts still exhibit some variability (e.g., contextual information) and some other parts have been fully configured (e.g., object of interest).

From now, the VS expert can trigger the automatic transformation ❺. We propose a transformation operator **transform** that takes as input VSAR, PFC and the transformation rules, $rules$. The operator computes two FMs, VSAR_{spe} and PFC_{spe}, such that VSAR_{spe} (resp. PFC_{spe}) is a refactoring [22] or a specialization of VSAR (resp. PFC).

$$\textbf{transform} : \text{VSAR} \times \text{PFC} \times rules \rightarrow \begin{cases} \text{VSAR}_{spe} \times \text{PFC}_{spe} \\ No\ solution \end{cases}$$

The operator is a partial function since the two FMs together with rules can be inconsistent. The transformational intent is to *propagate* edits to related FMs. The transformation operator reasons at the configuration level and tries to select or eliminate automatically undecided variability choices in VSAR and PFC. One of the result of the transformation, PFC_{spe}, is a specialization of the platform FM. A set of variability choices still needs to be resolved in this model. In some deployment scenarios, the PFC model of the VS processing chain has to be manually fine-tuned by the software engineer, during a specialization process ❹, as similarly done by the VS expert. The specialization processes can be reiterated by the two stakeholders.

When the application is run for the first time, the variable contextual information is set and an initial configuration of the VSAR model is produced (❻). This leads to the generation of an initial configuration for the PFC model thanks to a transformation (❼), and the corresponding software platform configuration is finally generated (❽). At runtime, the VSAR contextual information can be changed according to modifications of the real environment, enabling dynamic self-adaptation of the configuration of the running system [1].

B. Reasoning and Verification Techniques

Manually checking properties of FMs is error-prone and time-consuming. We thus need to reason on the VSAR FM, the PFC FM, and both of them together with the transformation rules. We propose to translate FMs and rules into propositional logic formula. This gives a direct and explicit *semantics* to these models, suitable for consistency and specialization verification, as well as model-to-model transformation.

Propositional Logic Encoding. An FM determines the combinations of features that are allowed and prevents the derivation of illegal configurations. FMs have been semantically related to propositional logic [5]. Each feature is associated with a Boolean variable that can be either true (included in the configuration) or false (excluded from the configuration). The set of configurations represented by a FM is thus described by a propositional formula. This translation into logics turns the configuration of the FM into a Boolean satisfaction problem (SAT), so that reasoning can rely on a precise semantics of FMs. Moreover, automated analyses can be performed by off-the-shelf, state-of-the-art SAT solvers or using Binary Decision Diagrams (BDD). The FMs used in the approach are trees with Xor-, Or-, and And-groups plus additional constraints. Batory presents how to map such FMs to propositional logic [5]. Therefore, VSAR (resp. PFC model) is translated into a Boolean formula ϕ_{vsar} (resp. ϕ_{pfc}).

ϕ_{vsar} and ϕ_{pfc} need to be linked by the transformation rules. Abstract syntax rules consist of a Left hand side (LHS) and Right hand side (RHS). Both of them address features possibly connected with "and", "or", "not". Well-formed rules state that the LHS (resp. RHS) considers only features of the VSAR model (resp. PFC model). Then the operators "implies", "excludes", "and", "or", "not" used in LHS and RHS may be mapped straightforwardly with the corresponding Boolean operators (respectively $\Rightarrow$, $\wedge$, $\vee$ and

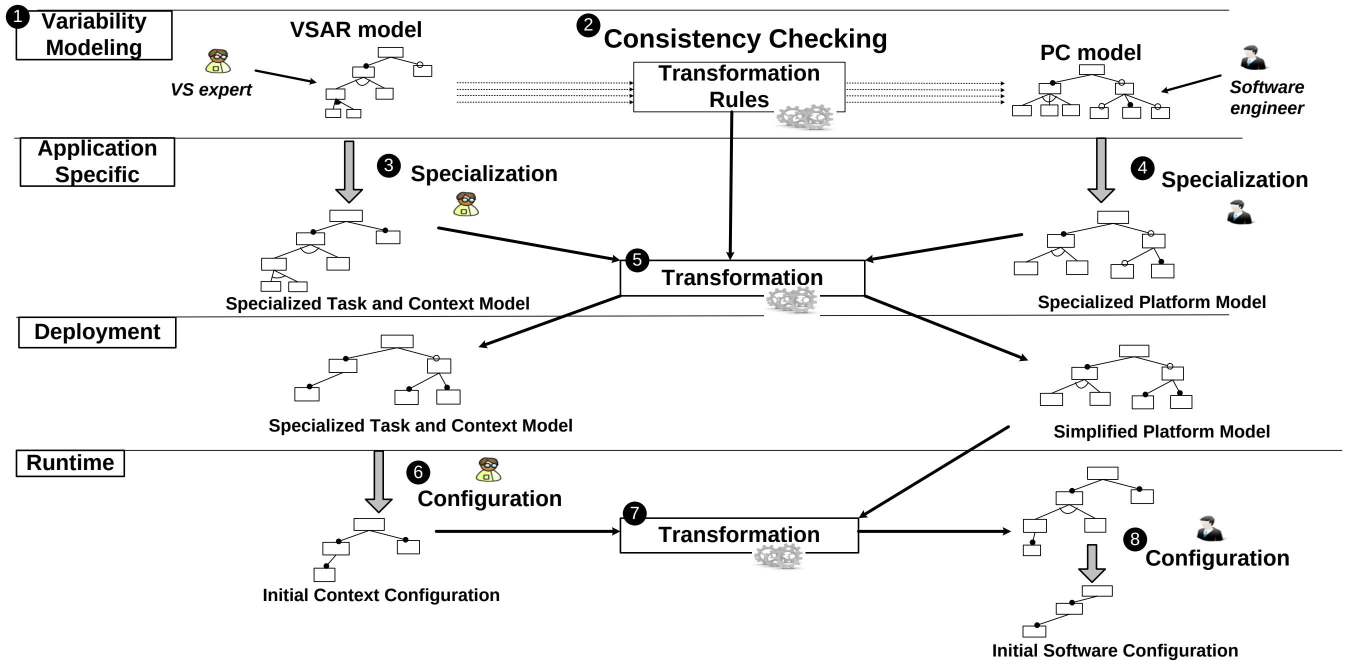


Figure 3: From Requirements to Deployment and Runtime: Process

$\neg$) into the propositional logic. This gives to the rules a clear semantics. As an example, the rule "high precision in Counting **implies** to use a low grid step during Segmentation **and** With window", expressed in the natural language, is translated into the formula: $HighPrecision \wedge Counting \Rightarrow Segmentation \wedge LowGridStep \wedge WithWindow$.

The entire set of rules is translated into a Boolean formula ϕ_{rules}. It is the conjunction of Boolean formulas associated to each rule. Once FMs and rules are translated into SAT, several reasoning operations can be realized, all described subsequently.

Consistency checking. An FM is *inconsistent* (or void or incorrect) if it does not contain any valid configuration [5], [20]. This is equivalent to stating that no solutions can be found to the Boolean formula associated with the FM. Inconsistent models may appear when, for example, FM designers specify constraints. Hence, the two FMs have to be checked by verifying ϕ_{vsar} and ϕ_{pfc}. This checking can also be used to ensure that the evolution and the incremental changes in both models are correct. Contradictory relationships are likely to occur when constraints are specified among the two FMs since the larger the FMs, the stronger the probability of inconsistency. In particular, if one configuration of the VSAR model is valid then it should lead, after applying rules, to one *specialization* (or configuration) of the FM[2]. Hence, we have to check the satisfiability of the following formula: $\phi_{global} = \phi_{vsar} \wedge \phi_{rules} \wedge \phi_{pfc}$

[2]A different and much stronger property is the specification *realizability* [17]: *every* configuration of the VSAR model should lead to (at least) one specialization of the PFC model.

Specialization checking. Let FM_1 and FM_2 be two FMs and P the function that takes a FM and returns the corresponding Boolean formula. To verify that FM_2 is a specialization of FM_1, we need to check that $P(f_2) \Rightarrow P(f_1)$ is a tautology [22]. Such checking can be applied when VS expert (resp. software engineer) edits the VSAR (resp. PFC) model to ensure that edit operations performed actually specialize the models.

Transformation through rule propagation. Specialization choices in the VSAR FM have possible complex consequences on features of the PFC model (and vice versa). It is thus suitable to propagate the choices *i)* internally to ensure the consistency of the two FMs and *ii)* externally to operate upon the other model (e.g., when a feature is removed).

We translate each specialization edit on FMs as a propositional constraint. For example, when a feature f is removed in VSAR model, we add the constraint $\neg f$ in the Boolean formula ϕ_{global}. If $\neg f \wedge \phi_{global}$ is not satisfiable, it reveals model contradictions and that the specialization processes are not coherent. In the case it is satisfiable, a mechanism then eliminates values that can not be part of any solution and/or infers some variability information associated to features. A first possible realization of such mechanism is to transform the Boolean formula $\neg f \wedge \phi_{global}$ as an FM using the synthesis algorithm described in [9]. The possible drawback is to obtain a non satisfying hierarchy, different from the original hierarchy of VSAR. Our strategy consists in *i)* removing all dead features (see Definition 3) detected in $\neg f \wedge \phi_{global}$; *ii)* then to simplifying the FM. For example, when there is only one feature in a Xor-group

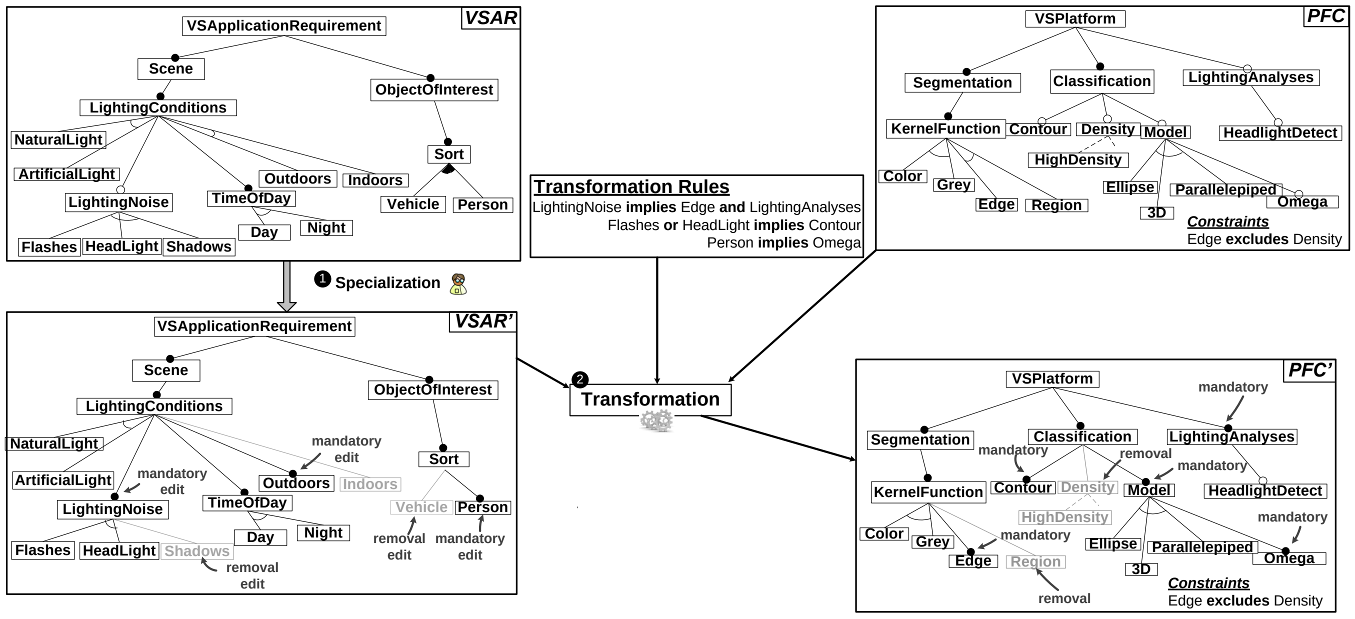


Figure 4: An example of specialization and transformation.

after the removal of the dead features, this feature becomes mandatory. In addition, core features that have an optional status are automatically corrected.

Example. In Figure 4, we illustrate the specialization activity and the automatic transformation using an excerpt of VSAR and PFC FMs and some transformation rules. First, a video surveillance expert *edits* VSAR FM (see ①) by removing the feature Shadows and setting the mandatory status to the features Outdoors, Lighting Noise and Person. As a result, some features no longer appear in VSAR FM (grey features in Figure 4). Then, the transformation operator takes the new specialized VSAR FM, VSAR', the transformation rules and PFC FM[3] (see ②). Hence several choices are automatically deduced in the platform FM: the features Density, Region and High Density are removed while the features Contour, Lighting Analyses, Omega and Model are core features.

Configuration validation. The configuration of an FM can be treated as a SAT problem. A solution to the SAT problem is an assignment that satisfies all the constraints. Hence, selecting (resp. not selecting) a feature instantiates the associated Boolean variable to the true (resp. false) value in ϕ_{vsar}, ϕ_{rules} or ϕ_{pfc} and can be checked at each configuration step.

V. Evaluation

A. *Tool Support*

In order to support the proposed process, we provide a comprehensive environment to both VS experts and software engineers. At its heart is FAMILIAR, a scripting language dedicated to the management of FMs. An Eclipse plugin provides editing and interactive interpretation, and a connection to FeatureIDE[4] enables users to graphically and interactively edit FMs while producing the equivalent FAMILIAR scripts. In particular, the language[5] allows FMs users to separate and inter-relate several FMs while automating the reasoning on their compositions – from validity checks to configuration process. These powerful FM composition operators are unique to FAMILIAR [2]. In our context, the language is used for realizing in an uniform way all the operations previously identified in the process:

- **Modeling variability.** VSAR and PFC FMs are specified separately, using either the FAMILIAR concise notation or by importing FMs from different formats (e.g., FeatureIDE, TVL). Internal constraints are specified either during the specification or, in a modular way, using the aggregate operator. We only show here the VSAR model (handling of the PFC model is similar):

```
scene_fm = FM (Scene :  AprioriKnowledge Environment;
     Environment: [Noise] LightingConditions ;
     Noise: [BackgroundMovement] [LightingVariation] ;
     ... ; LightingConditions: (Indoors |
     Outdoors | LightingType)+ ; ... ; )
ooi_fm = FM (ObjectOfInterest: Cardinality ... ;
     Cardinality: (SingleObject | GroupOfObjects) ; )
QoS_fm = FM (QualityOfService: (ComputerLoad
     | ResponseTime |Quality)+ ; ... ; )
task_fm = FM (Task : (Counting | BehaviourRecognition
     | Tracking |...); ... ; )
VSARrules = constraints ( Counting -> Large & Precision
     & VingtCinqFrSec; NaturalLight & Night ->
     Infrared & !Large ; ... ; )
VSAR = aggregate { task_fm ooi_fm scene_fm QoS_fm }
```

[3]The transformation really produces two FMs but we only depict PFC' since no variability choices are inferred in VSAR'.

[4]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

[5]A tutorial, a reference manual and script examples are available online: https://nyx.unice.fr/projects/familiar/

```
                              withMapping VSARrules
serialize VSAR into fml // VSAR.fml
```

- **Consistency checking.** Once the two FMs have been developed, transformation rules are specified and then mapped to features using a generic transform script, also implemented using the modular capabilities of FAMILIAR. Its call produces a new FM that can be further analyzed, for example to check that no *dead*[6] features are present:

```
VSAR_fm = FM ("VSAR.fml")
PC_fm = FM ("PFC.fml")
// transformation rules
trRules = constraints (Counting ->  ReferenceImageUpdating
     & LowGridStep; Night & HeadLight -> HeadLightDetection ;
     Flashes | HeadLight -> Contour ; LessPrecision ->
     (GridStep | WithWindow) & !SegmFineTune;
     ArtificialLight <-> ScenarioRecognition ; ... )
run "generic_transfo" { VSAR_spe PFC_spe trRules }
assert (size (deads deployment_fm) == 0)
```

The generic script follows, it notably checks that there exists at least one valid configuration to ensure that the transformation rules do not lead to contradictory relations:

```
// generic_transfo.fml
parameter fm1 : FeatureModel
parameter fm2 : FeatureModel
parameter rules : Set
deployment_fm = aggregate { VSAR_fm PFC_fm }
                              withMapping trRules
assert (isValid deployment_fm)
export deployment_fm
```

- **Specialization.** Using the graphical editor, the VS expert (resp. the software engineer) can use different modifier commands to specialize the VSAR (resp. PFC) model (e.g., **removeFeature**, **setMandatory** when translated into FAMILIAR):

```
// automatically generated from the interactive session
VSAR_spe = copy VSAR_fm
removeFeature VSAR_spe.LightingVariation
removeFeature VSAR_spe.Outdoors
removeFeature VSAR_spe.Counting
setMandatory VSAR_spe.Indoors
setMandatory VSAR_spe.Tracking
// ...
assert ((compare VSAR_spe VSAR_fm) eq "SPECIALIZATION")
```

At the end of the process, some generated code controls that the edited result is a specialization of the original VSAR model, as some modifications may lead to an *arbitrary edit* (e.g, by removing a *core* feature).

- **Transformation.** Once the two FMs have been edited, they are again related with the `generic_transfo` operation so that contradictory choices can be detected. If a propositional constraint involves a feature that no longer appears in the aggregated FM (e.g., due to a removal of a feature during the specialization process), the feature is set to false. In addition, the aggregated FM is simplified, for example, dead features that have eventually appeared are automatically removed (using **cleanup**):

[6]During the elaboration of the FMs, it can be considered as an error since it introduces an incorrect definition of relationships that does not match the intention of the FMs' developers.

Scenario	#edits	#configurations	#cores	#removes	#significants
1	13	48384	19	4	28
2	18	106560	18	2	31
3	12	24192	18	4	29
4	18	118656	18	1	32
5	16	32256	24	4	23
6	15	22608	21	2	28

Table I: Measurements on the application of the process

```
run "generic_transfo" { VSAR_spe PFC_spe trRules }
cleanup deployment_fm
```

Finally, the PFC sub-model can be extracted from the aggregated, simplified FM (by construction, it is a specialization of the original PFC model). The number of configurations in the specialized PFC model can be checked or simply visualized. The specialization/transformation process can be reiterated if needs be.

```
new_platform_fm = extract deployment_fm.VSPlatform
println ("#new_platform_fm=" counting new_platform_fm
         "vs #platform=" counting PFC_fm)
```

- **Configuration.** Finally, using the graphical tool, the software engineer selects / deselects features of the PFC model and obtains a valid software configuration that can be used to execute a VS processing chain at runtime.

B. Case Study and Experiments

To gather some validation elements, we experimented the proposed process on several deployment scenarios (detecting intrusion into building and a car park, counting persons, etc.) conducted by a VS expert using the FAMILIAR environment. Currently, the VSAR model has 77 features and its number of valid configurations is more than 10^8 ; the PFC model has 51 features and the number of valid configurations is more than 10^6 ; there are 22 transformation rules. For each scenario, we report the number of edits specialization used (#edits), the number of valid configurations in the PFC model after the specialization process (#configurations), the number of features removed in the PFC model (#removes), the number of non core features that remain to be chosen at runtime (#significants), which is equal to the total number of features in the transformed PFC model minus the number of core features (#cores).

For the experiments, we only considered the specialization of the VSAR FM by the VS expert and we do not consider the *direct* specialization of the PFC FM. Results show that, after the specialization and transformation, the number of valid configurations in the PFC FM is significantly reduced, at least of an order of magnitude. It confirms that the VSAR model can facilitate the specialization/configuration of the PFC model using high-level, domain-specific concepts. The transformation process allows to infer new core features (see #cores) and remove some features (see #removes). The initial PFC model contains 13 core features (and thus 51 - 13 = 38 significant features). In the six deployment scenarios, at least 6 and at most 15 features have been impacted during the transformation. Another observation is that fully specializing the PFC model using exclusively the VSAR model is impracticable. The intervention of the software engineer to specialize the PFC model is thus required, even if the process simplifies his task. As proposed in Section IV, the specialization edits produced by the VS expert and the software engineer have to be synchronized (and possibly reiterated). The figures of Table I give a good indication about the reduction of the models but it is difficult to determine, in the general case, when the specialization process of the VSAR and PFC models is *satisfying* and can be stopped.

C. Discussion and Related Work

1) Systematic Use of FMs: Several approaches promote the (systematic) use of FMs during SPL engineering [1], [13], [16], [17], [24]. FORM promotes a systematic organization of features in four layers, represented as four FMs related by dependency relations [16]. Metzger et al. distinguish product line from software variability, each being represented in two separated FMs [17]. The work presented in [24] proposes to generalize the previous approaches by providing a systematic way of separating and linking FMs in order to reason about relationships between requirements and feature configurations. We found many similarities with this work – the requirement FM and the problem world FM can be assimilated to the VSAR FM while the specification FM is similar to the PFC FM – though we do not consider quality attributes (see below). Other works take the challenge to refine FMs to software variability or design [15], [25]. For example, Janota et al. describe a formal approach, based on propositional logic, that supports the refinement of FMs in component models through some expressed dependencies [15]. However all above-mentioned papers do not consider software applications, such as video surveillance systems, in which some variability choices have to be delayed at runtime.

2) Reasoning about FMs: Our contribution is based on the semantic relation between FMs and propositional logic [5]. Even though FMs find a good reasoning basis with propositional logic, other formalisms have to be used when FMs and rules are somehow extended, for example, cloning or numerical values support [6], [26]. A possible drawback is that reasoning techniques (e.g., specialization checking) have to be adapted. As the complexity can dramatically increase, setting the right notational spectrum for each modeling step is of prior importance. The choice of a richer formalism has to be justified and carefully studied given the expected performance and needed operations. In our context, the formalism used is simple enough, yet expressive, to be used by VS experts and they can be given a formal semantics with useful outcomes.

Research work in the field of (automatic) reasoning about FMs leads to the (formal) definition of several operations and development of tools (e.g., FAMA framework [6], [23]). What we show in this paper is how to reuse and *combine* such operations in a rigorous process.

3) Dynamic SPL: Dynamically adaptive systems exhibit degrees of variability that depend on user needs and runtime fluctuations in their contexts. The goal of *dynamic* SPLs is to bind variation points at runtime, initially when software is launched to adapt to the current environment, as well as during operation to adapt to changes in the environment (e.g., see [12], [18]). Our contribution is a modeling process in which variation points related to the environment's static properties are bound before runtime so that the set of other variation points related to the dynamic, adaptive properties is automatically reduced.

VI. Conclusion

In this paper, we have presented techniques and tools to model and exploit software variability from requirement to runtime. The proposed approach distinguishes between *domain variability* and *software variability* and explicitly breaks the variability spaces into two FMs. In order to take into account the interactions between specification and implementation choices the FMs are interrelated with transformation rules. Specifying choices in the domain space then prunes the variability in the software platform and leads, at best, to a full configuration of the platform, and at least to a set of valid configurations. The remaining requirement specialized model then represents the contextual part and can be exploited for runtime adaptation, relying on some of our other work [1]. A tool currently supports the approach using a domain specific language for FM reasoning and connects with a graphical tool to facilitate interactive steps of the process. Moreover the tool support is not limited to two FMs, so that multi-level and multi-stage FM configurations can be reliably supervised and achieved.

The presented contributions are validated on a Video Surveillance (VS) software product line. As a result, the VS expert can select the features in his/her domain and obtains a combination of features that can be successively exploited by a code generation process and the software application engineer. First experiments show that, following our approach, the configuration spaces is reduced by an order of magnitude and enables the use of the other tools in the end-to-end engineering process, whereas it would not have been possible without.

In the short term, we expect to fully integrate our proposal in the end-to-end engineering of the VS product line. The applicability of the approach will also be demonstrated as an other work applies it to a different domain (medical imaging) with a different architecture (service-based grid).

Acknowledgment

The authors are grateful to the Keeneo company, especially to its R&D Head, Christophe Tornieri, for precious help in modeling knowledge about video surveillance.

References

[1] Mathieu Acher, Philippe Collet, Franck Fleurey, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Modeling Context and Dynamic Adaptations with Feature Models. In *4th International Workshop Models@run.time'09*, , pages 89–98. CEUR-WS.org, October 2009.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. A Domain-Specific Language for Managing Feature Models. In *Symposium on Applied Computing*, (SAC'11), Taiwan, March 2011. Programming Languages Track, ACM.

[3] Mathieu Acher, Philippe Lahire, Sabine Moisan, and Jean-Paul Rigault. Tackling High Variability in Video Surveillance Systems through a Model Transformation Approach. In *MiSE '09: Proceedings of the international workshop on Modeling in software engineering at ICSE 2009*, Vancouver, Canada, May 2009. IEEE Computer Society.

[4] A. Avanzi, F. Brémond, C. Tornieri, and M. Thonnat. Design and assessment of an intelligent activity monitoring platform. *EURASIP Journal on Applied Signal Processing, Special Issue on "Advances in Intelligent Vision Systems: Methods and Applications"*, 14(8):2359–2374, 2005.

[5] Don S. Batory. Feature models, grammars, and propositional formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC'05*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortes. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[7] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[8] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. In *Software Process Improvement and Practice*, pages 7–29, 2005.

[9] Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *SPLC'07*, pages 23–34. IEEE Computer Society, 2007.

[10] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[11] Franck Fleurey and Arnor Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *MODELS '09*, pages 606–621. Springer, 2009.

[12] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41:93–95, 2008.

[13] Herman Hartmann and Tim Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *SPLC '08*, pages 12–21. IEEE Computer Society, 2008.

[14] Florian Heidenreich, Pablo Sanchez, Joao Santos, Steffen Zschaler, Mauricio Alferez, Joao Araujo, Lidia Fuentes, Uira Kulesza amd Ana Moreira, and Awais Rashid. Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. *Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling*, 6210:69–114, 2010.

[15] Mikoláš Janota and Goetz Botterweck. Formal approach to integrating feature and architecture models. *Fundamental Approaches to Software Engineering (FASE)*, pages 31–45, 2008.

[16] Kyo Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[17] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering (RE'07)*, pages 243–253, 2007.

[18] Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132. IEEE, 2009.

[19] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[20] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2):456–479, 2007.

[21] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[22] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning about edits to feature models. In *ICSE'09*. IEEE, May 2009.

[23] Pablo Trinidad, David Benavides, Antonio Ruiz Cortés, Sergio Segura, and Alberto Jimenez. Fama framework. In *SPLC*, page 359. IEEE Computer Society, 2008.

[24] Thein Than Tun, Quentin Boucher, Andreas Classen, Arnaud Hubaux, and Patrick Heymans. Relating requirements and feature configurations: A systematic approach. In *SPLC'09*, pages 201–210. IEEE, 2009.

[25] Tijs van der Storm. Generic feature-based software composition. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition*, volume 4829 of *LNCS*, pages 66–80. Springer, 2007.

[26] Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating product-line variant selection for mobile devices. In *SPLC'07*, pages 129–140. IEEE, 2007.