Using Domain Features to Handle Feature Interactions

Sébastien Mosser  
SINTEF IKT  
INRIA Lille–Nord Europe  
Oslo, Norway  
first.last@sintef.no

Laurence Duchien  
INRIA Lille–Nord Europe  
LIIF (UMR CNRS 8022)  
Univ. Lille 1, France  
first.last@inria.fr

Carlos Parra  
INRIA Lille–Nord Europe  
LIIF (UMR CNRS 8022)  
Univ. Lille 1, France  
first.last@inria.fr

Mireille Blay–Fornarino  
I3S (UMR CNRS 6070)  
University of Nice  
Sophia Antipolis, France  
blay@polytech.unice.fr

ABSTRACT
Software Product Lines in general and feature diagrams in particular support the modeling of software variability. Unfortunately, features may interact with each other, leading to feature interaction issues. Even if detected at the implementation level, interaction resolution choices are part of the business variability. In this paper, we propose to use a stepwise process to deal with feature interactions at the domain level: the way an interaction is resolved is considered as a variation point in the configuration process. This method and the associated implementation are applied onto a concrete case study (the jSeduite information system).

Categories and Subject Descriptors
D 2.2 [Software]: Software Engineering—Design Tools and Techniques; D 3.3 [Software]: Programming Languages—Language Constructs and Features

1. INTRODUCTION
One of the most important challenges of Software Product Line (SPL) engineering concerns variability management, i.e., how to describe, manage and implement the commonalities and variabilities existing among the members of the same family of software products. In front of humongous SPL (containing hundreds of features, and describing thousands of products), all the possible interactions between domain features cannot be identified and accurately tackled a–priori. This issue is identified as the optimal feature problem: two features identified as independent at the domain level are actually dependent at the implementation level [16]. In this paper, we focus on feature interaction as the identification of a mismatch between the intention of the user and the obtained product. When several alternative strategies can be used to accurately resolve a given interaction (to fulfill designers’ intentions), these strategies are intrinsically part of the SPL: they rely a business know–how. Based on recent approaches that start to introduce extra–knowledge in the Feature Diagrams (FDs), e.g., multi–view configuration [13], we propose in this contribution to model these strategies as part of the FD to address the optional feature problem.

Consequently, we aim at defining an incremental process (which complements other approaches for product generation) that supports the stepwise identification of interactions. More specifically, we introduce in this paper an algorithm to support the endogenous capitalization of the resolving intentions. It enables developers to identify conflicts between features and to create resolving strategies based on their own intentions. Such strategies can then be capitalized in the Fd as new features. Consequently the Fd becomes more and more accurate with every new product derivation, as it is enriched with the intentions defined by previous designers for resolving the same interactions (based on their own experience). We apply this method on a service–oriented product family that uses behavioral assets, i.e., orchestrations of Web Services. The way an interaction is resolved directly interacts with the behavior of the derived product. Nonetheless, the approach proposed here is not specific to Service–Oriented Architectures1 and can cover different kinds of products.

2. RUNNING EXAMPLE: jSEDUITE
jSeduite is an information system designed to fit academic institution needs [20]. It supports information broadcasting from academic partners (e.g., transport network, school restaurant) to several devices (e.g., user’s smartphone, PDA, desktop, public screen). This system is used as a validation platform by the FAROS project2. The current stable version was released in February 2010 (development started in 2004), and represents ∼ 70,000 lines of code. jSeduite is now deployed inside three institutions: POLYTECH’SOPHIA engineering school & two institutions dedicated to visually impaired people (Clément Ader institute dedicated to childhood and the IRSAM association for adult

1We do not rely on any property specific to SOA, such as stateless systems or business orientation.
2http://www.lifl.fr/faros (French only)
people). Additional information can be found on the project website\(^3\).

For end-users, the entry point of the system is an information provider, implemented as a business process. In the Service-oriented Architecture (SOA) domain, a business process describes how existing services (e.g., components, web services) are orchestrated to perform a mission-critical and value-added task. In the context of jSEDUITE, each academic institution defines one or more providers, according to their specific needs. At a coarse-grained level, the goal of a provider business process is (i) to retrieve the data available on several sources of information and then (ii) to deliver their concatenation to the user. In addition, several broadcasting policies can be used to customize the way a source is handled, e.g., adding a cache, truncating the information set or using profile-specific value to filter the information set.

We formalize these variations through the definition of a FD and present in Fig. 1 a subset of the jSEDUITE FD. We differentiate (i) abstract features used to structure the FD [23] and (ii) normal features that are bound to implementation artifacts. This subset contains only two sources (i.e., News and Timetable, the actual system implements 19 sources) and six policies (e.g., Cache, Profile, 9 in the complete system). Even if restricted, this feature diagram can derive up to 500 different providers. We used the FAMILIAR language [2] to model the FD and compute the number of available configurations.

\(\text{Figure 1: JSEDUITE product line (subset)}\)

A jSEDUITE asset corresponds to existing artifacts that model the legacy system, including both structure (i.e., using class diagrams) and behavior (i.e., using business process formalism). We describe in Fig. 2 the assets associated to the feature Provider. Its structural part (Fig. 2(a)) defines the data types used in the system (i.e., Information and InformationSet), and describes a Provider service that defines two operations. The empty operation is internal, and is used to initialize an empty information set. The getInformation operation is publicly exposed, and implements the information retrieval process previously described. From a behavioral point of view, this process is described in Fig. 2(b). It is composed by three activities \((pg1, pg2, pg3)\), sequentially scheduled. In its initial version, this process simply computes an empty set of information. It starts with the reception of a given username \((pg1)\), and initializes info with an empty set \((pg2)\). This set is then replied to the caller \((pg3)\). Using a product-driven approach, it is possible to automatically derive a Provider according to user needs [8]. The designer selects the features he/she expects in the system (i.e., define a configuration), and automated generative techniques are used to generate a structurally correct system [8].

**Challenge.** However, semantic interactions can still be encountered at the behavioral level (see Sec 3.2). Considering that multiple strategies can be used to fix these interactions, it is up to the designer to select the right one, according to his/her intention. Thus, the way a designer resolves such an interaction is part of the domain variability: it represents a variation point at the domain level. Consequently, the FD is not expressive enough and must be enriched to take into account this new feature.

### 3. BUSINESS INCONSISTENCIES

In this section, we describe the product derivation process used to derive jSEDUITE products (SOA structure and behaviors of business processes). As the intrinsic design of jSEDUITE relies on orthogonal artifact from the class-diagram point of view, we only describe here inconsistencies that can be detected in the generated business processes.

#### 3.1 Product Derivation Principles

At the structural level, all the assets are orthogonal, and the generation of a complete class model for a given product can be automatically obtained using classical model composition mechanisms (e.g., Kompose, [10]). At the behavioral level, we use the composition framework ADORE [19] to support the generation of concrete providers from this FD. Anyhow, the proposed approach is not specific to this framework, as explained in section 4. ADORE is dedicated to the generation of complex business processes through a compositional approach. It supports the definition of business process fragments, which aim to be integrated into others. It accurately composes both sources and policies, generating providers behavioral implementation. We represent in
Fig. 3 two fragments used to support this generation. The fragment depicted in Fig. 3(a) models how information retrieved from the source News should be composed with the legacy provider (represented with dashed entities). The second fragment (doFilter, Fig. 3(b)) models how an information set can be filtered according to a given user profile.

$$\text{news} := \text{News}::\text{getContents()}$$

$$\text{legacy} := \text{union}(\text{legacy}, \text{news})$$

**Fragment: AddNews** (a) Handling News

$$\text{all} := \text{hook()}$$

$$\text{criteria} := \text{profile}::\text{getSet}(\text{user}, \text{key})$$

$$\text{all} := \text{Filter}::\text{run}(\text{all}, \text{criteria})$$

**Fragment: doFilter** (b) Filtering information (doFilter)

We consider here (Fig. 4) a product called $P_{\text{valid}}$, which reifies a configuration of jSeduite that broadcasts profiled news. In this product, the designer asks the system to generate a Provider that holds the News source, associated to a Profile policy (used to filter out the available news according to the profile of the user).

$$P_{\text{valid}} = \{\text{Provider, News, doFilter}\}$$

We represent in Fig. 4(c) the result of the derivation process. From the structural point of view, we generate the architecture of the SOA, using models merging. It includes the required services and the associated data types. From the behavioral point of view, we generate a business process that includes the two previously described fragments, based on a weaving algorithm.

3.2 Inconsistency Detection

We consider now the two following products, derived according to the principles previously described:

- $P_{\text{concurrent}} = \{\text{News, Provider, Timetable}\}$. In this product (depicted in Fig. 5(a)), one wants to combine two sources of information in the same Provider: News and Timetable. The latter introduces two activities $\{t_2, t_3\}$ which retrieve the set of current lectures in the school ($t_2$) and append it with the legacy information set. This product leads to the derivation of a non-deterministic process, since the activities $\{a_2, a_3\}$ define a concurrent access to the variable info. At the domain level, prioritizing the information retrieval tackles the issue.

- $P_{\text{term}} = \{\text{Authentication, Log, Provider}\}$. In this product (depicted in Fig. 5(b)), the features Log and Authentication are selected to enhance the initial Provider. On the one hand, the feature Log brings activities $\{l_1, l_2\}$, logging the user access and throwing an error if there is no available logger. On the other hand, the feature Authentication adds activities $\{a_1, a_2\}$, which respectively check user authentication token and throw an error when the token is rejected by the security service. It is then not possible to predict its behavior when both error conditions are triggered (i.e., there is no available logger and the token is rejected). As multiple business choices can be used to tackle this issue [6], it is also part of the domain variability.
Figure 4: Generation of $P_{valid} = \{Provider, News, doFilter\}$ artifacts
\{f_1, f_2\}, one can choose in this context up to four\(^4\) different interaction resolution rules to fix the issue: (i) ignore the interaction, (ii) keep only one of the two features, (iii) order the composition finally, (iv) manually tailor a composed feature. Even if the solution is actually implemented at the code level, the underlying business intention is part of the domain variability. We summarize in Tab. 1 how all these choices can be applied to \(P_{\text{concurrent}}\).

\[
P_i | \text{Choice} | \text{Business Intention}
\begin{align*}
P_0 & \quad \text{Ignore the interaction} \quad \text{The built provider will be explicitly non-deterministic} \\
P_1 & \quad \text{Exclude News} \quad \text{The News feature is finally irrelevant in this specific product} \\
P_2 & \quad \text{Exclude Timetable} \quad \text{The Timetable feature is finally irrelevant in this specific product} \\
P_3 & \quad \text{Give priority to News} \quad \text{News information must be retrieved as a priority} \\
P_4 & \quad \text{Give priority to Timetable} \quad \text{Timetable information must be retrieved as a priority} \\
P_5 & \quad \text{Tailored solution} \quad \text{e.g., News and Timetable information must be interlaced}
\end{align*}
\]

Table 1: Resolving “variations” for \(P_{\text{concurrent}}\)

These choices intrinsically implement variations of the same product. This fact leads to the following conclusion: the iSEDUITE FD depicted in Fig. 1 is not expressive enough to accurately model the real domain variability. A naive solution is to systematically model all these variations as part of the FD, introducing “resolving rules” as features. In this FD, features associated to the Provider contain the composition directives used to derive the final product, and “resolving rules” contain additional information intended to the weaver (to resolve the interaction, e.g., ordering). As the number of feature interactions in the product line is unknown a-priori, we cannot restrict the cardinality of the interaction space. In the worst case, each configuration of a given FD \(F\) will trigger a different interaction. Considering that one can use up to four alternative choices to automatically resolve these conflicts (automatic resolution intrinsically excludes tailored solutions, which must be manually written), the number of associated features that should be automatically added in the FD reaches the cardinality of \(F\) powerset, i.e., \(4 \times (2^{13} - 1)\). Considering the iSEDUITE restriction previously described (containing only 13 features but up to 500 available configurations), it will introduce \(4 \times (2^{100} - 1)\) features, which is not reasonable.

4. FD STEPWISE ENRICHMENT

Considering that user intentions when resolving interactions are part of the intrinsic variability of an FD, we propose to store in the associated FD this knowledge, reified as features. Unfortunately, this approach does not scale as is. It is not possible to automatically fill the FD with all the potential rules, as (i) it will overwhelm the FD (combinatorial explosion), (ii) not all the possible resolution choice makes sense at the business level and (iii) it is not possible to foresee all the potential tailored rules. We propose a stepwise approach to support interactions resolving in an incremental way. The key idea is to rely on an interaction detection engine to start a dialog with the designers, leading to the enrichment of the FD at the domain level. Using the algorithm described to support the approach, the FD is only enriched with relevant knowledge, and becomes more and more accurate (in terms of interaction resolving) at each step.

Let \(f_d\) be a feature diagram, and \(cfg\) be a configuration. If the analysis of \(cfg\) identifies interactions [11], an inference engine will explore \(f_d\), mining in the existing resolving features potential candidates (if any). The user is then asked to (i) pick one or more existing resolving rules in the candidates set, or (ii) to enrich \(f_d\), adding a new resolving rule according to his/her intention. The enriched SpE is now called \(f_d'\), and will be used as a reference for the upcoming configurations. We propose the use of features to model resolving strategies. The asset associated to a strategy is a set of resolving rules\(^5\). Resolving strategy features are discriminated against “usual” features according to a boolean predicate (isStrategy?). One can use a fixes function to retrieve the set of interacting features fixed by a given resolving strategy. Based on this definition of interaction resolving strategies, it is possible to formally define how an automatic engine can propose strategies to fix a given set of interactions. Let \(c\) be a given configuration of a \(FD\) \((f_d)\), and \(I_e\) a set of interacting features in \(c\). A resolving strategy \(s\) may interest the user since it fixes several interactions identified in \(I_e\) (i.e., the intersection of \(I_e\) and \(\text{fixes}(s)\) is not empty). We define a findCandidates function to automate this task, described as follows:

\[
\text{findCandidates} : \text{Feature}^* \times \text{FD} \to \text{Feature}^* \\
(I_e, f_d) \mapsto C
\]

\[
C = \{s | \exists s \in f_d, \text{isStrategy}(s), \text{fixes}(s) \cap I_e \neq \emptyset\}
\]

4.1 Illustrating Scenario in iSEDUITE

We depict in Fig. 6 how this approach can be efficiently used to resolve interactions in the iSEDUITE context. We restrict the initial FD to only two relevant features (i.e., News and Timetable) for concision reasons. At the beginning (step \(S_0\), Fig. 6(a)), the left part of the FD contains only “system-driven” features. Using this FD, one expresses a configuration \(c\) which involves both News and Timetable features. The derived product holds a concurrent access interaction (see Fig. 5(a)). As there is no available strategies in the FD, the user is asked to model his/her intention as a new resolving strategy. In this case, a new feature is added (External), where the user states that external information must be prioritized\(^6\) (see \(P_3\) in Tab. 1). At the implementation level, the associated resolving rule indicates to the composition engine how the composition directives must be executed to properly implement this choice (using ordering directives). At the end of this step (\(S_t\)), the user retrieves an interaction—\(^7\)a resolving rule as an atomic information intended to the weaver. Such information is used to resolve a conflict at the composition engine level.

\(^4\)The resolution rules presented here can be complemented by others. The key idea is that variations exists in the way a given interaction can be resolved, according to the intention of the user.

\(^5\)A resolving rule as an atomic information intended to the weaver. Such information is used to resolve a conflict at the composition engine level.

\(^6\)The News source is an external entity (typically a syndication feed retrieved from a national channel). On the contrary, the Timetable source is an internal entity, using a specific system deployed only on the school network.

\(^7\)A resolving rule as an atomic information intended to the weaver. Such information is used to resolve a conflict at the composition engine level.
free product \( p \), and the FD is enhanced with the new strategy (Fig. 6(b)), which crystallizes the user resolving intention at the FD level.

Now, if another user expresses a configuration involving News and Timetable. The same interaction is detected, and the system identifies\(^7\) that the External strategy can be used to resolve it. Unfortunately, this strategy does not match user’s intention (which is antagonist, i.e., prioritize internal information, see \( P_1 \) in Tab. 1). A new alternative strategy is then added to reify this decision (Internal). At the end of this step (\( S_2 \)), the user retrieves another interaction-free product \( p' \neq p \), and the FD now contains two alternative strategies to handle News and Timetable interactions.

### 4.2 Algorithm Description: derive

We describe in this section how the approach can be fully-supported through the implementation of an automated algorithm. We define this derive algorithm (Fig. 7) as independent of the underlying composition engine. It only rely on the two following assumptions on the underlying composition engine: (i) a compose function used to generate the concrete system implementation associated to a given configuration and (ii) a check function able to detect interactions between features involved in a configuration\(^8\).

The derive algorithm receives as input a configuration \( \text{cfg} \), and the associated feature diagram \( \text{fd} \). It starts by retrieving the composition directives associated to the features \( (l_1) \), and then runs the check function to identify interactions \( (l_2) \). If no interaction is detected, the valid product associated to the execution of the compose function \( (l_4) \) is returned to the user \( (l_5) \).

If the engine identifies interactions associated to \text{cfg}, the findCandidates function is executed on \text{fd} to identify relevant strategies able to resolve this interaction \( (l_5) \). Then, the user is asked to choose in the candidates set which strategies he/she wants to use in the product \( (l_6,l_{10}) \). If no existing strategy fits his/her intentions, the system asks the user to enrich the existing FD with a new strategy \( (l_8) \).

Then, the engine retrieves the composition directives associated to the enhanced configuration, including the chosen or written strategy \( (l_4) \). The check function is now re-played \( (l_{15}) \), and if a choice which leads to the identification of more interactions than initially is detected, this decision will be rejected by the engine (divergent decision, \( l_{17} \)). The algorithm is then recursively called to continue the synthesis process. It produces as output the expected product \( p \), and an enriched feature diagram \( f'd \).

Based on the divergence test performed on line \( l_{16} \), we ensure the termination of the algorithm (the interaction set cardinality decreases at each call, and the empty set detection triggers the synthesis of the product implementation). We plan to extend this restriction in future work to support an historic-based approach. That is, choosing a feature that actually increases the interaction set cardinality will be accepted by the system. But a choice that generates previously encountered interactions will be rejected, ensuring termination.

Thus, the derive algorithm automatically supports the enhancement of the FD with accurate resolving strategies. Qualitatively, the associated knowledge is capitalized in the

---

\(^7\)Let \( I_c = \{ \text{News, Timetable} \} \), and \( f_1 \) the feature diagram depicted in Fig. 6(b). findCandidates\((I_c,f_1) = \{ \text{External} \} \).

\(^8\)In ADORE, the compose function is the call to the weaver, and the check function is the execution of logical interaction detection rules on the composed business process.
FD and becomes available for next configurations. The following section illustrates quantitative benefits of our approach.

5. VALIDATION ON jSEDUITE

The complete FD associated to jSEDUITE follows the same principles that the ones described in this paper. It contains 19 sources, and up to 9 policies can be applied on each source. This FD is intrinsically optional, and according to the FAMILIAR counting algorithm, the number of available configurations is consequent.

\[ |jSEDUITE| = 1.408845839590877 	imes 10^{63} \]

This gigantic set of potential configurations takes its root in the intrinsic goal of jSEDUITE: the definition of information providers according to users needs, which are highly variable. It is obvious that the \( 4 \times 2^{jSEDUITE} - 1 \) potential resolution rules cannot be modeled in the FD. In this section, we illustrate how the previously described algorithm tackles such a complexity, using real providers deployed in existing academic institutions.

As a first example, we consider a product \( P_{hall} \) used to generate a public provider, broadcasting information in the main entrance of the Polytech’Nice School of Engineering. This product select 7 sources of information, and applies up to three policies on the same source (20 features were selected in the FD). Considering \( P_{hall} \) as the first product selected in the jSEDUITE FD, there are no available resolution strategies, and the designer is asked to provide such strategies to resolve the automatically detected interactions. We summarize in the following list the different interactions automatically detected by the composition framework:

\( I_1: \{\text{Cache, Diet}\} \): The Diet policy reduces the weight of an information, deleting several parts of the content to only keep essential data. It interacts with the Cache mechanisms, as we need to know which data (initial or after deletion) should be cached. For instance, the main entrance screen broadcasts a lot of different information, potentially overwhelming the cache server with irrelevant information. In this situation, our business intention is to minimize the size of the cache.

\( I_2: \{\text{Cache, Profile}\} \): The Profile policy changes the content of the retrieved set of information, according to a user-specific profile. Such profiled information set should not be stored in the cache. In this situation, our business intention is to avoid user-specific data caching.

\( I_3: \{\text{Profile, Truncate}\} \): The Truncate policy restricts the cardinality of an information set up to a given threshold. The Profile also deletes information from the information set. Thus, these two filtering policies must be ordered. In this situation, our business intention is to emphasize user experience (information accuracy), even if it degrades the response time (matching a large set of information with a profile is more time consuming than a truncated one).

9For simplification purposes, we consider that two interacting sources of information can be arbitrarily ordered (automatically), reducing the interaction space. This heuristic is used in the jSEDUITE running system.

\( I_4: \{\text{Cache, Profile, Truncate}\} \): A Cache–Profile interaction was handled in \( I_2 \), and a Profile–Truncate interaction was handled in \( I_3 \). When these three features interact together, the selection of the two previously described strategies resolves the conflict.

To resolve the interactions identified in \( P_{hall} \), we introduced three new features (to resolve \( I_1, I_2 \) & \( I_3 \)), and reused two of them to resolve \( I_4 \). The \( I_2 \) interaction was also detected twice and resolved with a single feature. Using the enriched FD obtained as output of the previous step, one can now select a new product, and restart the process.

As a second example, consider now the product \( P_{staff} \), dedicated to a broadcasting screen located in the staff cafeteria of the same school of engineering. This product targets 8 sources, and the final selection contains 22 features. The interaction detection steps identify the following interactions: Cache–Diet (\( \times 2 \)), Profile–Truncate (\( \times 2 \)) and Cache–Shuffle. The latter is resolved through the incrementation of the asset associated to the GlobalTruth feature (it does not make sense to shuffle the content of the information set before storing it in the cache), and the others through the selection of available strategies. That is, the conflicts are handled through the reuse of 2 existing strategies, and the introduction of a new one.

We continue this process to derive concrete products, that are deployed and used daily in academic institutions. For 5 different product configurations, we have identified 18 interactions (Tab. 2). However, we have only added 6 resolving strategies in the jSEDUITE FD, as we were able to reuse 10 times the resolving strategies that had been previously capitalized as features. This point emphasizes the benefits of the approach: modeling resolving decision as features supports designers during the generation process, allowing immediate reuse of previously defined intentions. We are developing a software factory environment intended to end-users. The idea is to let the school headmasters configure, generate, and finally deploy their school-specific providers in an automated way.

<table>
<thead>
<tr>
<th>Product</th>
<th>Features</th>
<th>Interactions</th>
<th>Resolving strategies</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{hall} )</td>
<td>20</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>( P_{staff} )</td>
<td>22</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>( P_{stud} )</td>
<td>20</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>( P_{nder} )</td>
<td>25</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>( P_{exam} )</td>
<td>14</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: 18 | 6 | 12

Table 2: Incremental derivation of providers

6. RELATED WORK

Several approaches deal with the conflicts among features and ways to solve those conflicts at the level of assets. For example, in [9] the authors present an automated procedure for verifying that a given feature configuration will lead to a correct product model. The notion of correctness they consider verifies that the resulting product model conforms to the meta-model of the target modeling language. Our proposal is complementary to these approaches: we start from the assumption that interaction detection already exists, and we rely on such detection. In our case, we focus
on the capitalization of reusable resolution strategies, incrementally enriching the initial FD. We propose an iterative algorithm that allows the identification of resolving strategies based on users’ intentions, and the reification of such strategies as new features to be used by subsequent users. In the Aspect-oriented paradigm, the MATA approach [25] supports the composition of models aspects using a graph-based approach. This approach supports powerful conflict detection mechanisms, used to support the “safe” composition of models [21]. The underlying formal model associated to this detection is based on critical pair analysis [12]. Initially defined for term rewriting system and then generalized to graph rewriting systems, critical pairs formalize the idea of a minimal example of a potentially conflicting situation. This notion supports the development of rule-based detection mechanisms, used to support the “safe” composition of conflicts in the software product line domain [15]. Our proposition is complementary to this one, as we define the algorithm as independent of the underlying conflict detection mechanisms. Thus, powerful approaches such as the previously described can be easily reused in our approach.

Algebraic models can be used to model features. This method is used by state-of-the-art approaches, such as the AHEAD method [5] that support step-wise refinement of products. Clark et al. follow this path and propose an algebraic model used to model features delta [7]. According to their approach, a feature asset is implemented as a delta (i.e., an increment) to be added to the system core to derive the expected product. When two conflicting deltas $\delta_1, \delta_2$ are added, a resolution delta is inserted after the conflicting ones to fix this interaction. This approach assumes that a conflict resolution delta already exists for all conflicting pairs. Thus, all the conflicting pairs have to be identified a-priori. Our approach is complementary, as we propose to identify the unforeseen conflicting pairs during derivation. It is then possible to introduce this new knowledge in their reasoning capabilities, and validate the enhanced product line.

Feature-aware verification [4] advocates the detection of feature interactions at the product line level. Based on model-checking techniques, it is possible to automatically check a given product line to identify interactions a-priori. This approach relies on the existence of a semantic specification (e.g., using the Alloy modelling language [14]) of each feature, used as input by the reasoning engine [3]. Our approach is complementary, as we tackle the interaction resolution without requiring the existence of a semantic specification. Thus, the proposed algorithm strengthens feature-aware verification mechanisms, providing an automated method to handle under-specified interactions.

White et al. propose a strategy to derive a product configuration that meets a set of requirements over a span of configuration steps [24]. They argue that, when a product is modified from one current configuration to a target one, even if the target configuration is correct, several FD constraints might be violated in the intermediate steps of the modification. To face this problem they transform the FD into a constraint satisfaction problem and propose an associated solver. The multi-step constraint is then tackled as a set of transformations. In our case, we do not face directly the problem of deriving a product in multiple stages. We assume a single-staged derivation process. However, since each product generation might arise interactions among the set of selected features, we propose to incrementally learn from the choices made at each generation, and reify such knowledge as new features in the FD. Abbas et al. propose the notion of autonomic software product lines [1]. The main idea is to dynamically change the configuration of a given product using as input the context information (i.e., information about the application and its environment available only at runtime). They propose a learning process in which, based on the history of changes, a solver can decide which configuration is more appropriate for the current context situation. Our approach differs in two ways. First of all, we allow the definition of new features. In their approach, the authors do not discover new features, their solver selects the best configuration from the same set of features defined before the execution of the product for the whole SPL. Second, they work at runtime and use context information, whereas we base our approach on the intentions of every different user, and the incremental process is performed at design with multiple users and multiple products being generated.

Finally, in [22] Stoiber et al. propose a tool for feature unwrapping. It consists in identifying the variability of an application from a graphical software requirements model. They infer a semantically equivalent model that groups the elements belonging to the same feature into aspects. Contrary to their approach, to discover new features we follow an incremental process where the initial FD is enriched with resolving strategies for interactions between features. The features are reified from previously chosen strategies and not from assets of an application already developed.

Liu et al. propose to use derivative features to tackle the feature optionality problem [17]. This approach relies on an algebraic representation of the features (based on AHEAD [5]). Our approach is complementary to this one, as instead of writing a resolution strategy, one can implement a derivative feature to resolve the interaction. However, the originality of our approach is to consider that multiple resolution rules (in this case, derivatives) can simultaneously exist, with regard to the intention of the user.

Broy proposes a service model to support multifunctional systems [18]. He uses modes to model service dependency, as well as feature interactions in this domain. Our work is also complementary, as we propose a way to enrich a given model to support multiple resolution rules associated to the same identified interaction.

7. CONCLUSION & FUTURE WORKS

In this paper, we have presented an approach that tackles the incremental handling of feature interactions. This approach supports the enrichment of a SPL through the identification and reuse of resolving strategies among features. We start with a product family represented through a FD. For every feature in the FD, we define an associated model defining a particular business process. We use the ADORE framework and in particular, the composition approach based on aspect weaving for building business processes from a given product configuration. We focus on the incremental discovery of strategies for interactions among features. Concretely, we have defined an algorithm that helps developers during the product configuration process. It is executed every time
a new product is derived and looks for interactions among the features selected in the configuration. For each interaction found, it asks the developer to decide whether to select an existing resolving strategy or to create a new one. In the latter case, the strategies are integrated as new features in the initial Fd to be reused again for successive products.

We have applied our strategy on the jSeduite product family. The results show that our approach increases the accuracy of the Fd over time, and highlights the importance of evolving the Fd with the knowledge and intentions of previous users of the SPL. Using incremental code-generation techniques to enhance the generation process is an upcoming perspective.

An immediate perspective of this work is to assess it on a larger case study. We are now focusing efforts on the SensApp platform\textsuperscript{10}, a highly configurable platform that deals with sensor network. The platform can handle a large variety of sensors description standards and protocols, generating a humongous numbers of potential products.

For future work, we would like to explore two main fields. First of all, our approach can be extended to face the challenges of SPL refactoring. The refinement process here can be considered as an alternative to SPL evolution. Thus, it can be further improved to use information that comes not only from previously chosen strategies but from already built products as well. Second, our approach could be extended to support also architectural constraints. Up until now, since we are based on the ADORE weaving for the services, we do not tackle directly the possible interactions that may exist in the structure of the products.

Acknowledgments. This work is partially funded by the French Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER–CIA) 2007-2013. This work is also supported by the SINTEF strategic projects SiSaS and MODERATES.

8. REFERENCES


