Context Awareness for Dynamic Service-Oriented Product Lines

Carlos Parra, Xavier Blanc, and Laurence Duchien
INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, Université de Lille 1, France
{carlos.parra, xavier.blanc, laurence.duchien}@inria.fr

Abstract

This paper presents a Context-Aware Dynamic Software Product Line (DSPL) *for building service oriented applications and adapting them at runtime in accordance with their using context. This DSPL, named* CAPucine *for* C*ontext-*A*ware Service-Oriented* P*rod*uct *L*ine *is based on two different processes for product derivation. The first process uses assets that represent features of the product family. The assets, represented as models, get composed and transformed in order to generate the product. The second process relates to dynamic adaptation. This process introduces context-aware assets that operate at runtime. These context-aware assets contain three kinds of data: the context when the assets can be modified, the place where the assets must be applied and the change that must be performed. The realization of these context-aware assets combines two runtime platforms. On the one hand,* COSMOS *is a context-aware framework connected to the environment by the use of sensors. On the other hand* FraSCAti *is a* Service Component Architecture *(SCA) platform with dynamic properties that enables to bind and unbind components at runtime.*

1 Introduction

Context-aware systems are systems that can react to changes of their environment [6, 12]. The classical and historical example of such systems is the one of a mobile system whose behavior changes depending on its location [28]. Nowadays, requirements for context-aware software, specially for mobile computing, are more and more difficult to handle as many context situations have to be considered such as limited connectivity, hardware heterogeneity, and changes of user preferences.

Context-aware systems can also benefit from the *Software Product Line* (SPL) paradigm. SPL focuses on variability management and aims at deriving different products from a same product family [8]. In SPL, feature diagrams express the variability of a same product family by defining its variants and its variation points [29, 30]. Based on a feature diagram, one of the SPL's key principles is the *Product Derivation* (PD) that is the complete process of constructing a product from an SPL [11]. The PD defines how assets are selected in order to cover a selection of features of a feature diagram and specifies how they are integrated in order to compose the final product. It should be noted that PD is not confined to design or architectural phases, but it can also be applied at runtime in order to address the iteration phase of [11], making the SPL more dynamic [13]. Dynamic SPL (DSPL) then produces systems that can be adapted at runtime in order to dynamically fit new requirements or resources changes.

Service-Oriented Architectures (SOA), whose recent achievements have enabled the development of mobile systems [18], is another technology that can be used to build context-aware systems [5]. With SOA, software is decomposed into *services*. One advantage of SOA is that services can be defined, invoked, and composed considering their well-defined interfaces, but without considering their implementation, i.e. their provider. SOA has several standards for service-oriented modeling such as WSDL [31] and BPEL[21], but no complete methodological approach from the requirement until the implementation is proposed for building service architecture. The *Service Component Architecture* (SCA), proposes a reconciliation between SOA and *Component-Based Software Engineering* (CBSE). In fact, SCA defines a framework for describing the composition and the implementation of services using components [22]. Nevertheless, SCA provides no support for context awareness which is one of the keystones for the next generation mobile systems.

SOA follows an approach by reuse and composition of services whereas SPL corresponds to an approach by building and decomposition. Few recent research approaches emphasize on a reconciliation between SPL and SOA, but no one proposes a convincing methodology for building service applications with SPL [32], [2], [9], [16].

Our claim is that using an SPL paradigm to build context-aware systems based on SOA services, enables a complete service development from requirements to imple-

mentation, and a management of context throughout the software lifecycle.

In this paper, we propose an homogeneous Context-Aware *Dynamic Service-Oriented Product Line* (DSOPL) named *CAPucine*. Our goal is to define at the same time a service-oriented and context-aware PD that monitors the context evolution in order to dynamically integrate the appropriate assets in a running system. Our target platforms follow the service-oriented approach. We use FraSCAti [23], an SCA platform with dynamic properties enabling binding and unbinding of components at runtime. We also emphasize in the use of sensed information from the environment, to dynamically realize the PD. In particular, we are based on COSMOS [26], which is a context-aware framework connected to the environment by the use of sensors. Thanks to COSMOS, the environment is abstracted by a set of software components — the so called context nodes — that offer runtime operations reflecting the environment state.

The reminder of this paper is organized as follows. In section 2, we present a motivating scenario that will be referred throughout the paper to illustrate our approach. Section 3 introduces our proposal for a context-aware product derivation. Section 4 describes the product derivation technologies and implementation for the motivating scenario. In section 5, we briefly compare and position our work with other proposals found in the literature. Finally, in section 6, we conclude and present some ideas for future work.

2 Motivation

The objective of this paper is to provide a Context-Aware DSOPL. The DSOPL needs to be able to derive service-oriented products and to monitor their context in order to adapt their architecture at runtime. In this section, we present a motivation scenario (see subsection 2.1) that highlights the challenges (see subsection 2.2) to be addressed in order to build such DSOPL.

2.1 Scenario

Our scenario consists in defining a DSOPL that builds a family of systems used to obtain and display information about movies. In SPL, feature diagrams are used to classify all the requirements that can be fulfilled by the product family [29]. Figure 1 presents such a diagram for the movie system family, using the notation presented in [10]. In this diagram, features are presented in a tree-like form. Dark circles of every connection represent mandatory features whereas white circles represent optional features. An inverted arc represents a set of alternative features meaning that exactly one feature has to be chosen.

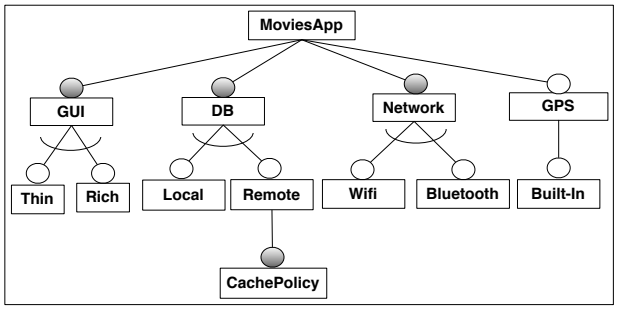

Figure 1. Feature model of the Movie System

The feature diagram of Figure 1 expresses the fact that all movie systems derived from the SPL require a *Graphic User Interface* (`GUI`), a *Database* (`DB`), a `Network Interface`, and optionally a `GPS`. Moreover, it defines that the `DB` can be either `Remote` or `Local`. If the database is remote, then the movie system has a `Cache Policy`. The `Cache Policy` enables the system to decide whether it first queries the information in the cache or in the `Remote DB`. Finally, the feature diagram defines that the `GUI` can be either `Rich` or `Thin`, that the `GPS` may be present as a `Built-in` device and that the network interface may correspond either to `Bluetooth` or `Wifi`.

To derive a final product, the first step is to select from a feature diagram the list of features that has to be supported. This list of features should be compliant with the constraint of the feature diagram. For our scenario, we propose to select the following features: `Rich GUI`, `Remote DB` with `Cache Policy` and `Wifi Network`. This list is compliant with the constraints defined by the feature diagram. Indeed, all mandatory features have been selected (`GUI`, `DB` and `Network`), the `Remote DB` has been selected as the `Cache Policy` was also selected. Once the features selection is done, the final product can be derived. The two approaches that can be followed to perform this step are the *Selection approach* or the *Assembly approach* [11]. Whatever the approach, this step mainly consists in (1) either generating or creating software assets, (2) assembling and configuring them in order to build the final product and finally, (3) validating the result to be sure that the selected features are supported. For our scenario, we propose that the assets are composed of services realized by software components following the SCA architecture style [22].

Figure 2 illustrates an assembly of components that represents the architecture of this product. Following the SCA notation, arrows to the left of each component represent the services that it provides, arrows to the right represent the references or services it requires. In this case there is a `GUI` component which offers a `run` service, and is bound with the `DB` component through the `getDesc` reference. Al-

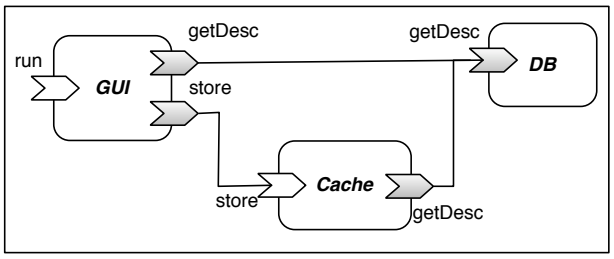

Figure 2. SCA assembly.

though it is not represented in the diagram, the remote BD implies that the SCA binding links components running in different places. In this case, the connection takes place through a `Wifi` network. `GUI` is also bound to the `Cache` component through the `store` reference in order to retain information locally.

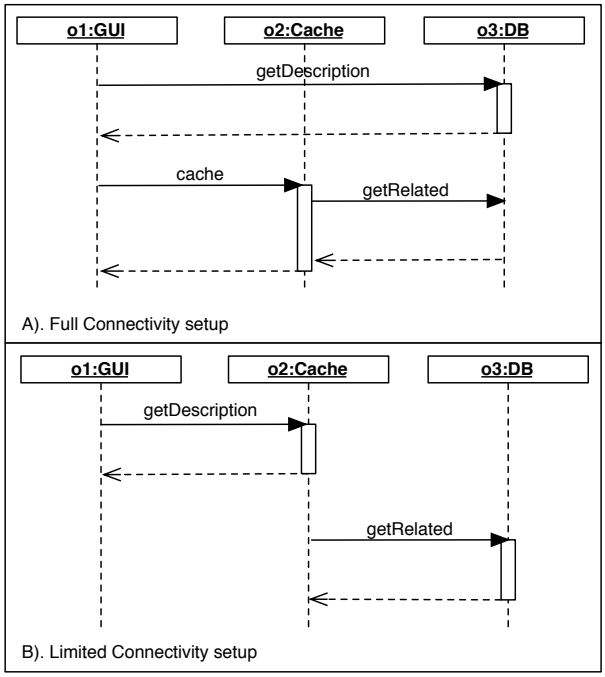

Figure 3. Sequence diagrams for low and full connectivity setup.

Once the product has been derived, it can be deployed and started. At runtime, the product can be dynamically adapted to fit its environment changes. In our scenario, we propose to dynamically adapt the `Cache` of the Movie System, depending on the bandwidth quality. On the one hand, if the bandwidth is high (*full connectivity*), then the `GUI` directly interacts with the database without using the cache. During this period of time, the cache stores locally all the movie descriptions browsed by the user, but also requests other ones that have been defined to be closely related. On the other hand, if the bandwidth is low (*limited connectivity*), then the `GUI` uses the cache, that is still connected to the database. Figure 3 shows the two alternative configurations that work respectively with a full connectivity and a low connectivity bandwidth. Moreover, to keep the system from constantly changing due to a non-stable bandwidth, we propose to apply a classical *double threshold* pattern. Thanks to such a pattern, the architecture of the movie system will be adapted to the full connectivity alternative only when the bandwidth goes over the *Maximum threshold* and likewise if it falls below the *Minimum threshold*.

2.2 Challenges

The scenario presented in subsection 2.1 helps us to identify challenges that have to be faced in order to define a context-aware DSOPL. We classify those challenges in two categories that correspond to the two well-known phases of the SPL product derivation, i.e. the *initial phase* and the *iterative phase*) [11].

2.2.1 Initial phase challenges

The initial phase starts with the selection of the supported features and ends with the deployment of the first version of the final product. To achieve a PD, a link between the selected features and the assets that realize them have to be established. Those assets, representing the architecture in terms of structure (elements that constitute the product) and behavior (interaction between elements), have to be composed in order to obtain a product. The result of such composition is a set of well-connected assets that represent the user requirements. Ideally, these assets, representing the structural and behavioral parts of the product, are platform-independent. After the composition, a refining process is needed to transform the assets and match the deployment restrictions. This issue relates to the capability of the DSOPL to successfully separate business concepts from the details of the underlying platform. Two main challenges arise in this phase:

- *Composition of Assets:* There is a need for a process that integrates multi-view assets (behavioral and structural design) of each selected feature to build up a product that satisfies user requirements.
- *Transformation of Assets:* There is a need for a separation of concepts in different domains and a transformation of the first ensemble of platform-independent assets into the real components that match the execution platform.

2.2.2 Iterative phase challenges

The iterative phase starts as soon as the system has been started and ends at the end of the system life cycle. The iterative phase deals with context-aware variability as it is related to the dynamic adaptation of the system in response to context changes. Context-aware systems always have alternative architectures that fit to precise context state. In our example, two alternative architectures have been introduced and mapped to the full and the limited bandwidth context states. We distinguish two main challenges to achieve dynamic adaptation in context-aware systems:

- *Context-aware assets:* The first challenge relates to clearly define how to introduce alternative architectures. This problem is composed of two subproblems dealing with (1) how to represent context-aware alternative architectures and their related context states as SPL runtime assets and (2) how to bind them with features that appear in a feature diagram.

- *Context-aware variability realization techniques:* The second challenge refers to manage the acquisition of context information and to realize the dynamic adaptation of the system. The DSOPL should integrate a runtime platform that can monitor the context and, depending on changes, dynamically realize the adaptation of the running system. The platform has to be context-aware and has also to deal with runtime adaptation. Moreover, this platform should be driven by context-aware assets defining context states that have to be monitored, but also adaptations that have to be realized.

3 CAPucine

Section 2 presented a motivating scenario and ended with the presentation of the challenges that have to be faced in order to build DSOPL. This section presents CAPucine, that is our proposal for a DSOPL. CAPucine covers the initial and the iterative phases of the product derivation.

3.1 CAPucine for the Initial Phase

In section 2.2.2. we presented two challenges for the integration of assets in the initial phase: asset composition and transformation. To face these challenges, CAPucine's initial phase is based on a model-driven approach. For every selected feature, there is an associated asset that in our case, corresponds to a partial model of the product itself. This is similar to what has been proposed in [25] which advocates the use of *Aspect Oriented Modeling* (AOM) and proposes an automatic merge mechanism that integrates the models corresponding to the system features. In our case, we have specified the application metamodel illustrated in Figure 4. The application metamodel describes from a high level of abstraction, the structure and behavior of an SCA application. The goal is to represent every feature as a partial model that conforms to this metamodel and afterwards, compose the parts to have one integrated model that represents the product. The application metamodel is formed by two different parts: one for structure modeling and one for behavior modeling.

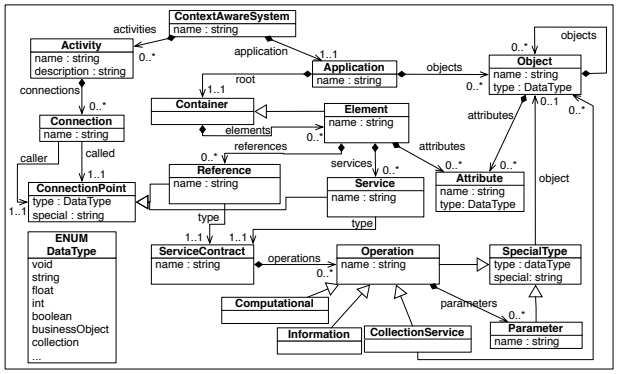

Figure 4. Application Metamodel

- *Structural and Business Modeling:* In the structural and business part, there are different meta-classes like `Element`, `Service`, and `Reference` that enable the description of a generic structure of a service-oriented application. The information of these elements is used to build the SCA components and their implementations in Java. By specializing the `Operation` elements specified inside the `ServiceContracts`, we bound the scope of the line by defining a set of standard functionalities for communication, collections, persistence, and user interface.

- *Behavior:* The behavior is represented by two meta-classes: `Activity` and `Connection`. An activity is composed of many connections. The general idea behind these elements is to represent a sequence of services communicating with each other. Every connection has two references to two different `ConnectionPoints` (e.g. services or references), the one that makes the call and the one that is called. An `Activity` is essentially a basic set of links to represent a business process, hence, the `Connection` meta-class can be further specialized to support different types of BPEL-like activities like: splits, forks, etc.

The composition of assets to obtain the integrated application model is the first step of product derivation. The next step is to transform this model to enrich it with concepts of the platform, and the implementation language. This is done by performing a series of *model to model* transformations towards the platform and the implementation domains. Finally, the product is built by generating the code from the target domains. Figure 5 summarizes the different steps of the initial phase of CAPucine's product derivation.

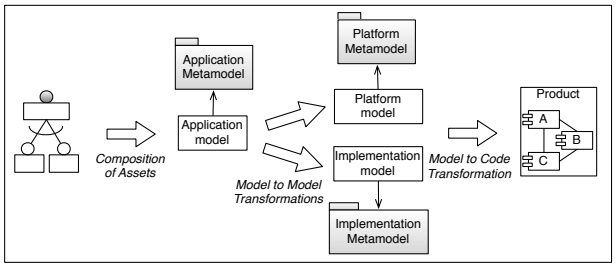

Figure 5. CAPucine's initial phase.

3.2 CAPucine for the Iterative Phase

In section 2.2.2 we identified two challenges for the iterative phase. These challenges correspond to context-aware assets and variability realization techniques. In this subsection we present in detail the CAPucine approach to offer a solution to each one of these challenges.

3.2.1 Context-aware assets

We call *context-aware assets*, the assets that can be integrated at runtime, depending on the environment state. Context-aware assets own the different alternative architectures of a system and their conditions of existence that depend on the environment state.

To model context-aware assets, we define the Context-Aware Assets metamodel (see Figure 6) which extends the application metamodel of Figure 4. This metamodel defines that a context-aware asset is composed of several `clauses`. Each clause is composed of one `test` and one `body`. The clause `test` is a classical condition expressed on an `observable` that is an object abstracting a state of the environment, i.e., the expression property of the metaclass Condition is used to specify the condition. The clause `body` is a dynamic adaptation composed of two parts: a `place` and a `change`. The `place` defines the places in the system where the dynamic adaptation will be realized. The `change` defines the change actions that will be realized to adapt the system at runtime.

We propose to make explicit the fact that a context-aware asset may be deterministic if at most one of its clauses's test succeed whatever the state of the environment. It should be noted that non-deterministic context-aware assets introduce non-deterministic behavior because more than one dynamic adaptation can be realized for a particular environment state. Therefore they may cause conflicts or unexpected behavior depending on the order in which adaptations are realized.

Context-aware assets are, as defined by their name, dependent of the context. Our metamodel makes this dependency explicit thanks to the `Observable` concept. Our intention is to use this explicit dependency as a constraint for the software product line runtime platform. Indeed, the platform has to provide context information acquisition in order to support the realization of the assets.

It should be noted that context-aware assets have many differences in comparison to classical assets. In particular, the decision of their integration into the system is undertaken dynamically and driven by context changes, their realization is done by adapting the behavior of the system at runtime, and they depend on an existing system that is running and therefore is bound to classical assets.

Figure 7 illustrates a partial model that represents the asset of our example. This asset contains two clauses (for the full and the limited connectivity). Each clause contains one condition that is linked to an observable (`BANDWIDTH`) and one dynamic adaptation. The dynamic adaptations correspond to a set of change actions that will be processed in the runtime platform and will be detailed in section 4.

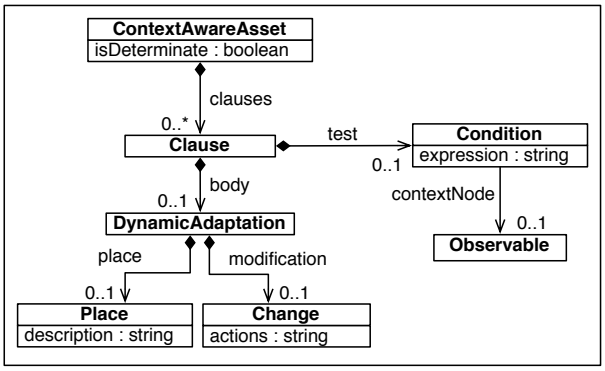

Figure 6. Metamodel of context-aware assets

3.2.2 Context-aware variability realization techniques

In [30], Svahinberg et al. define that variability realization techniques are used to integrate assets while building the final products. Moreover, authors clearly identify *Component-Based Software Engineering* (CBSE) as one of the variability realization techniques that can be used at runtime. In this section, we propose a CBSE platform as a vari-

```
context-aware-asset isDeterminate='true'{
 clause {
  condition
     expression='if value > 14'
     contextNode='BANDWIDTH'
   dynamic-adaptation {
    point description='...'
    change change-actions='...'
   }
 }
 clause {
  condition
     expression='if value < 8'
     contextNode='BANDWIDTH'
    dynamic-adaptation {
      point description='...'
      change change-actions='...'
    }
  }
}
```

Figure 7. The context-aware cache asset

ability realization technique. This platform is similar to an SCA platform where applications are constituted by components that offer services. This platform will support both acquisition of context information and dynamic adaptation:

- *Acquisition of context information:* As it was explained in subsection 3.2.1, the context-aware assets include a definition of a context information that corresponds to the decision whether or not to adapt the system. Hence, the platform needs a context aggregation mechanism. Such a mechanism is in charge of getting the information from multiple sources and to provide a high-level view of information, so that, it can be evaluated.
- *Dynamic Adaptation:* Successful product derivation depends also on the ability of the platform to reconfigure the system. The platform has to be able to suspend and resume the execution of the system, modify its structure by performing different operations like deploy, add, bind or delete components. This enables dynamic adaptation for each context-aware asset.

The platform architecture is depicted in Figure 8. The *Context Manager* element is composed of several nodes. Every node is in charge of recovering context information from different sources like a sensor layer who captures raw data from the environment, user preferences, and the *Runtime Platform* who provides information about current state and configuration of applications. Eventually, the *Context Manager* can also perform a processing of data, so that, it is presented as single values which can be evaluated in the condition of each context-aware asset.
The *Decision Maker* element is in charge of evaluating the context and decide whether or not to modify the application. It is linked to a repository of rules. The rules represent the clauses of each context-aware asset.
Finally, the *Runtime Platform* element is where *Application Components* are executed. It controls the life cycle of all the application components and has access to their control mechanisms.

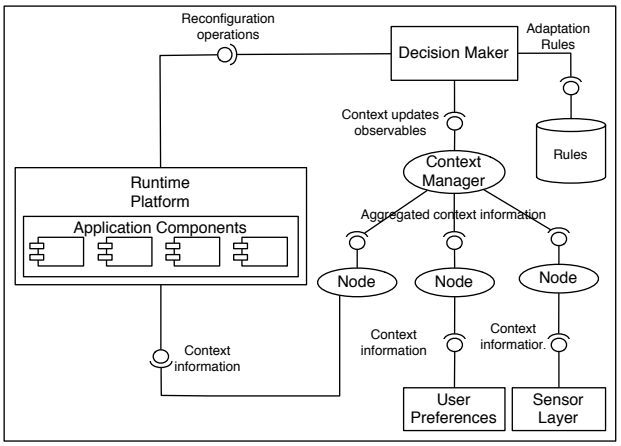

Figure 8. Platform Architecture

4 Validation

To validate our approach, we have designed and implemented the Context-Aware DSOPL *CAPucine*. Given that *CAPucine* target applications present context-aware features, we have implemented the *initial* and *iterative* phases of product derivation. The next subsections describe the technologies and implementation details of each phase.

4.1 Initial phase

In the initial phase we implement two main processes: (1) a composition of assets (related to selected features) using the application metamodel, and (2) a transformation from this metamodel to the platform and implementation domains, and later on, to source code. To explain these processes we refer to the example of section 2. We start by selecting a set of features. Figure 9 illustrates this process. For every feature, there is an associated asset that in our case corresponds to a partial model of the product itself. Dashed arrows link the features with their model part. The application model that results from the chosen features conforms with the application metamodel detailed in section 3. Stereotype notation (<<>>) is used to represent the conforms-to relationship of every element of the model with a given meta-class in the application metamodel. For example, the feature *CachePolicy* is represented by several elements in the model. First, there is a link to the context

manager to verify the bandwidth, represented as the bandwidth observable. Also, there is a *FullLimitedConnectivity* element that references both the bandwidth and the *FullConnectivity* and *LowConnectivity* activities. These activities represent the binding of the `getDesc` reference of the user interface with the remote database and the cache respectively. This enables switching between both activities and changing the architecture of the product from one activity accessing directly to the remote database to another one accessing a local cache during disconnection or weak signal periods.

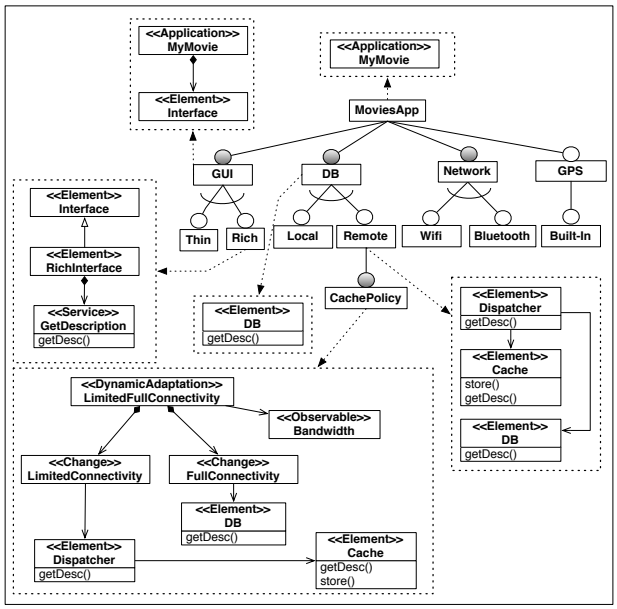

Figure 9. Derivation of products

4.1.1 Metamodels

Once the application model is obtained, it becomes the source of model to model transformations towards two different domains: the platform domain, in our case SCA, and the implementation domain, in our case Java. Each domain is represented as a metamodel. For SCA we have used the metamodel proposed by the *Open Service-Oriented Architecture* (OSOA) collaboration group. Although we do not modify the proposed metamodel, we only reference the elements that we need in order to generate the SCA composite [22]. For Java we have used *Spoon EMF* to model the concepts that have been defined to express the structure of a Java application [20].

4.1.2 Transformation and code generators

In order to transform the model that represents a product, we use model transformation and code generation. To transform the model, we use the *Kermeta* language to obtain SCA and Java elements out of application elements [17]. Each operation is basically a mapping between the concepts of the application metamodel and the consequent elements in Java and SCA. Once we obtain models that conform to SCA and Java, we proceed with code generation. To achieve this, we use *Acceleo* language [1] which navigates the model and creates the source code (`*.composite` files for SCA and `*.java` files for Java).

4.2 Iterative phase

Dynamic product derivation is made by reconfiguring the system at runtime. This requires a platform with the characteristics described in section 3. Here we present a detailed description of each part of the platform addressing requirements for context management, execution platform, decision making and product derivation.

4.2.1 Runtime Platform

We use the FraSCAti platform [23]. FraSCAti is a Fractal-based SCA implementation.

SCA establishes that components are the basic building blocks. Each component requires and provides services. SCA supports several service description languages like WSDL and Java interfaces, several programming languages such as Java, C++, and BPEL, several communication protocols between applications such as SOAP, CORBA, Java RMI, and JMS. Fractal [7], on the other side, is a hierarchical and reflective component model intended to implement, deploy, and manage complex software systems. Fractal offers several features like composite components (components containing subcomponents), sharing (multiple enclosing components for the same subcomponent), introspection, and re-configuration. A Fractal component can expose elements of its internal structure and offer introspection and intercession capabilities. Several controllers have been defined in the Fractal specification like the binding controller that allows the dynamic binding and unbinding of component interfaces, and the life-cycle controller that allows to perform operations like stop and start the execution of a component.

In FraSCAti, Java-based SCA components are simultaneously both SCA-compliant and Fractal-compliant. The main benefit of this particular property is that all the components can be dynamically reconfigured at runtime.

In Figure 10, we present an extended SCA assembly of the example of section 2, that proposes two different configurations. We add a composite named `Channel`. This is a composite that offers a `getDesc` service to the `GUI` component. It also requires `getDesc` reference from the `DataBase`. Inside this composite, there are

two components: `Dispatcher` and `Cache`. The `Store` and `getDesc` references of the `Dispatcher` component, can be bound in two different ways. In this case, stronger lines represent the full-connectivity configuration and dashed lines represent the limited-connectivity configuration. The `GUI` component has also references to the `DecisionMaker` to monitor the context information.

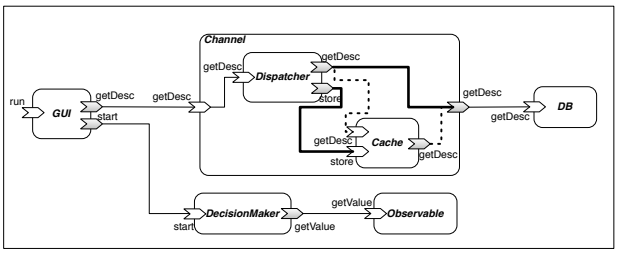

Figure 10. FraSCAti example with limited and full connectivity.

4.2.2 Context manager

Our context manager is COSMOS [26]. It is a component-based framework for gathering and processing context information. It obtains the context information from different sources like sensors, network probes, or systems, and processes it according to defined policies. These policies are described as hierarchies of context nodes using a dedicated composition language. Each COSMOS node in the top of the hierarchy is connected to a single *observable* of the context aware asset.

4.2.3 Decision Making

The Decision Making component is attached to the COSMOS notification service. It receives a notification each time a context change occurs. The notification is linked with an observable in order to filter the context-aware assets that are impacted by the change. The decision algorithm is quite straightforward. It evaluates only the clauses that have the observable linked with the change notification as a `contextNode` (see Figure 6). Then, for the ones that have been evaluated to true, it realizes their body. It should be noted that such a decision algorithm is quite trivial and is safe only for determinate context-aware assets. We think about defining other ones, based on inference techniques, in order to make our platform more customizable.

4.2.4 Adaptation

The adaptation takes place at runtime. Two activities have to be performed to adapt a running system. First, the *places* where the changes have to be realized should be identified. Second, components of the system should be adapted safely regarding their states.

We use FPath and FScript [24] for these two activities. FScript and FPath notations are two domain-specific languages to code dynamic adaptation of Fractal-based systems. FPath eases the navigation inside a Fractal architecture with simple and readable queries. FScript, that makes use of FPath, enables the definition of adaptation scripts to modify the architecture of a Fractal application. One of the key advantages of FScript is that it provides a transactional support for architectural reconfigurations and then guarantees that the application remains consistent if the reconfiguration fails at a given point. Figure 11 shows the action that adapt the system to its full connectivity alternative architecture. The first part of the code is the `place` part of the adaptation. It is written in FPath and consists in identifying the three sub-components that will be adapted. The second part of the code is the `body` part of the adaptation. It is written in FScript and consists in (1) stopping the identified sub-components, (2) changing their bindings and (3) restarting them. If one of these FPath or FScript instructions fails, then the complete action is rolled-back thus keeping the system in a consistent state.

```
action adaptToFull() {
 --Place
 dis = $explorer/descendant::dispatcher;
 ca = $explorer/descendant::cache;
 ch = $explorer/descendant::channel;
 --Body
 stop($dis);
 stop($ca);
 unbind($dis/interface::getDesc);
 unbind($ca/interface::store);
 bind($dis/interface::getDesc,$ca/interface::getDesc);
 bind($ca/interface::getDesc,
      $ch/internal-interface::getDesc);
 start($dis);
 start($ca);
}
```

Figure 11. FPath and FScript adaptation.

5 Related Work

Few successful approaches refer to the definition SPL for SOA. Most of these approaches are in a preliminary state or propose some ways for a reconciliation between SOA and SPL [32], [2], [9], [16]. In [3], Bastida et al. develop dynamic self-reconfiguring and context-aware compositions by applying a multi-step methodology based on product-line engineering notions of variability management. They base their service composition infrastructure on Event-Condition-Action Rules. Their approach is mainly based on a composition service at design-time. At runtime, a BPEL engine and a rule engine compose the middleware part. The context part is not explicitly defined in the complete approach.

In a different way, work on adaptive systems and context awareness in SPL is prolific. In [4], Bencomo et al. propose software product lines for adaptive systems. In their approach, a complete specification of the context and supported changes has to be provided thanks to a state machine. Each state then represents a particular variant of the system and transitions between states define dynamic adaptations that are triggered by events corresponding to context changes. The main limitation of this approach is that the state machine has to be provided during the design step and cannot be extended, making the system quite static as it can only be adapted for a fixed set of change events. Nevertheless, they manage to introduce optimizations for the selected transitions.

In [19], Morin et al. propose an approach that deals with dynamic variability in software product lines and that encompasses the limit of [4]. Their approach relies on AOM for specifying variants and for realizing the binding of variation points. This use of AOM replaces the use of the state machine of the [4] approach and then makes the adaptation more dynamic. Our approach is similar to their approach for this particular point. Authors also claim to propose a context-aware adaptation model that is in charge of selecting adequate variants depending on the context. Unfortunalely, no detail is presented in the paper in order to understand how the context is specified and monitored. Our approach can then be considered as an extension of their work.

In [25], Perrouin et al. propose a SPL based on AOM. In their proposal, variants are specified thanks to model fragments and the product derivation is done automatically by merging them together. The initial phase of our approach is based on the same principle but it extends the product derivation with context-based adaptations.

In [14], Hallsteinsen and al. use product lines techniques for building adaptive systems. In their proposal, adaptive systems are implemented using component-based architecture and variability modeling and they delegate the adaptation complexity to a reusable adaptation platform. Our approach is similar. It is based on the same kind of platform but we integrate at the same time the notion of context in the product line and in the runtime platform.

In [27], Salifu et al. propose an approach to variability for software product-families that deal with context-awareness. They link requirements to software architecture using product-family paradigms, but they do not support the context integration in the runtime platform.

Finally, in [15], Hartmann et al. propose the concept of a context variability model that contains the primary drivers for variation (e.g. different geographic regions). This model constrains the feature model in order to choose one dimension in the context space. This context variability model remains as a static way to manage a product derivation in accordance with an orthogonal variation description.

6 Conclusions and Future work

In this paper, we identified and faced four challenges for context-aware adaptation in a DSOPL. The two first ones relate to the initial phase. They concern (1) the need for a process that integrates multi-view assets, and (2) the need of separation of concerns in different domains. To face the first one, we used an aspect-oriented modeling for an automatic merge of models that represent the different features of the product family. For the second one, we proposed a separation in three different domains that represent the application, the platform and the implementation, and a set of model transformations to obtain the source code. The two last challenges concern the iterative phase. They deal with (1) the context-aware assets and (2) their realization techniques. First, we defined a context-aware metamodel that represents the context that triggers the modification, the place where the asset must be changed and the new configuration that must be applied. Second, we proposed a platform that enables the realization of such assets, and finally, we validated our approach using several component-based and service-oriented technologies.

Our approach is homogeneous as we propose two different processes for the initial and iterative phases of product derivation. The advantages of CAPucine are two-folded. First, we propose a context-aware asset that introduces alternatives in the software product line and that is considered at runtime. The context-aware asset defines the information of an adaptation when the context changes. Second, we are based on a set of realization techniques for context-aware variability by the way of context-aware tools, such as COSMOS and dynamic reconfiguration tools such as FraSCAti.

As for future work, we plan, in the short term, to study the reduction of non-deterministic behaviours when we introduce non-deterministic context-aware assets. Indeed, we consider an open environment where more than one dynamic adaptations can be realized for a particular environment state. Therefore, they may cause conflicts or unexpected behavior depending on the order in which adaptations are realized. We plan to reduce these conflicts by a set of strong pre-conditions that will precise the condition for introducing new behaviours.

References

[1] ACCELEO. Code generator, 2008. `http://www.acceleo.org`.

[2] S. Apel, C. Kaestner, and C. Lengauer. Research challenges in the tension between features and services. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems*

development in SOA environments, pages 53–58, New York, NY, USA, 2008. ACM.

[3] L. Bastida, F. J. Nieto, and R. Tola. Context-aware service composition: a methodology and a case study. In *SDSOA '08: Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 19–24, New York, NY, USA, 2008. ACM.

[4] N. Bencomo, P. Sawyer, G. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *2nd International Workshop on Dynamic Software Product Lines (DSPL 2008)*, Limerick, Ireland,, 2008.

[5] J.-Y. Bitterlich and al. JSR 172: J2ME Web Services Specification, Java community process, 2004.

[6] P. J. Brown. The stick-e document: a framework for creating context-aware applications. In A. Brown, A. Brüggemenn-Klein, and A. Feng, editors, *Special Issue: Proceedings of the Sixth International Conference on Electronic Publishing, Document Manipulation and Typography, Palo Alto*, volume 8, pages 259–272, John Wiley and Sons, June 1996.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[8] P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.

[9] S. Cohen and R. Krut, editors. *Proceedings of the First Workshop on Service-Oriented Architectures and Software Product Lines (CMU/SEI-2008-SR-006)*. Software Engineering Institute, Carnegie Mellon University, 2008.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[11] S. Deelstra, M. Sinnema, and J. Bosch. Experiences in software product families: Problems and issues during product derivation. In R. L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2004.

[12] A. K. Dey, G. D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, 2001.

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[14] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[15] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference*, pages 12–21, Washington, DC, USA, 2008. IEEE Computer Society.

[16] A. Helferich, G. Herzwurm, S. Jesse, and M. Mikusz. Software product lines, service-oriented architecture and frameworks: Worlds apart or ideal partners? In *Trends in Enterprise Application Architecture*, volume 4473/2007, pages 187–201. Lecture Notes in Computer Science, Springer, 2007.

[17] INRIA Project-Team Triskell. Kermeta tool suite, 2008. `http://www.kermeta.org/`.

[18] N. Josuttis. *SOA in Practice, The Art of Distributed System Design*. O'Reuilly, August 2007.

[19] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. S. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 782–796, 2008.

[20] C. Noguera and L. Duchien. Annotation framework validation using domain models. In *Fourth European Conference on Model Driven Architecture Foundations and Applications*, pages 48–62, Berlin, Germany, June 2008.

[21] OASIS Standard. Web Services Business Process Execution Language Version 2.0, April 2007.

[22] Open SOA. Service component architecture specications, Nov. 2007. www.osoa.org/display/Main/ Service+Component+Architecture+Home.

[23] OW2 consortium. FraSCAti project. `http://frascati.ow2.org`.

[24] OW2 consortium. Fscript and Fpath. `http://fractal.objectweb.org/fscript/`.

[25] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling automation and flexibility in product derivation. In *12th International Software Product Line Conference (SPLC 2008)*, pages 339–348, Limerick, Ireland, Sept. 2008. IEEE Computer Society.

[26] R. Rouvoy, D. Conan, and L. Seinturier. Software architecture patterns for a context-processing middleware framework. *IEEE Distributed Systems Online*, 9(6):1, 2008.

[27] M. Salifu, B. Nuseibeh, and L. Rapanotti. Towards context-aware product-family architectures. In *IWSPM '06: Proceedings of the International Workshop on Software Product Management*, pages 38–43, Washington, DC, USA, 2006. IEEE Computer Society.

[28] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *In Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, 1994.

[29] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International*, pages 136–145, 2006.

[30] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Softw., Pract. Exper.*, 35(8):705–754, 2005.

[31] W3C. Web Services Description Language (WSDL) 1.1, 2001 March. `http://www.w3.org/TR/wsdl`.

[32] C. Wienands. Studying the common problems with service-oriented architecture and software product lines. In *Service Oriented Architecture (SOA) & Web Services Conference*, Oct. 2006.