Modeling the Variability Space of Self-Adaptive Applications

Gilles Perrouin, Franck Chauvel, Julien Deantoni, Jean-Marc Jézéquel

Modeling the Variability Space of Self-Adaptive Applications*

Gilles Perrouin
IRISA (INRIA & Université de Rennes 1)
Campus de Beaulieu
F-35042 RENNES (FRANCE)
Gilles.Perrouin@irisa.fr

Franck Chauvel
VALORIA & Université de Bretagne Sud
Centre de Recherche Yves Coppens
Campus de Tohannic
56017 VANNES Cedex (FRANCE)
Franck.Chauvel@univ-ubs.fr

Julien DeAntoni, Jean-Marc Jézéquel
IRISA (INRIA & Université de Rennes 1)
Campus de Beaulieu
F-35042 RENNES (FRANCE)
{Julien.DeAntoni,Jezequel}@irisa.fr

Abstract

Modeling self-adaptive applications is a difficult task due to the complex relationships they have with their environments. Designers of such applications strive to model accurately a few (re)-configuration possibilities deemed to be the most relevant with respect to environmental changes. This deliberate restriction of the variability space is cumbersome and may unnecessarily reject interesting (re)-configuration possibilities. We employ software product-line techniques to properly cover the whole variability space of a self-adaptive application. This variability space is partitioned across three dimensions. Functional variability is modeled through a feature diagram whose features are realized by a set of components to be deployed on a platform. Topological variability is modeled via an UML collaboration excluding irrelevant configurations. Platform variability is modeled through constraints to be satisfied by configurations. For each dimension, we exhibit properties capturing the environment. Our modeling approach is illustrated on a web-server example.

1. Introduction

The fast emergence of dynamic environments (such as mobile systems, web-services, peer-to-peer networks) requires more flexibility from software systems running in these environments. This specificity raises new development challenges. One emerging possibility to address these challenges consists in including self-adaptation capabilities into the application. So-called self-adaptive systems observe their environment and accordingly adapt their internal configuration in order to reach some quality objectives [4, 19, 14].

However, due to the lack of specific methods and tools, engineers often have to design a set of possible architectural configurations and associate to each of them a possible state of the environment. To do so, they restrict the variability space of the self-adaptive application in a drastic way. However, a self-adaptive application has a complex relationship with its environment which impacts application functionalities, performances and depends on the abilities of the platform on which the application is running. Trying to capture this relationship directly in a few architectural configurations is inherently difficult and exposes engineers to the risk of overlooking important environmental states and thus missing interesting architectural configurations.

We believe that designers should properly model the variability space of a self-adaptive application first. This enables designers to hold all cards before modeling architectural configurations of the application. The contribution of this paper is a modeling process supporting the definition of variability spaces. Each variability space is defined with respect to one of the following dimensions: *functional*, *platform* and *topological*. The first dimension is defined via a feature model that defines variability across application functionalities. These functionalities are then related to a set of components gathered in a repository. It is important to notice that rather than modeling exhaustively architectural configurations in the topological space, we

*This work has been partially supported by the DiVA STREP project.

only constrain them via a UML collaboration so that irrelevant configurations can be excluded during the design process. Finally, we illustrate how platform variability can be modeled through non-functional properties.

The remainder of the paper is organized as follows. Section 2 introduces the web-server that will be used as a running example. Section 3 defines our modeling process while Section 4 formalizes the models offered. Section 5 highlights some related work. Finally, Section 6 wraps up with conclusions and outlines future research directions.

2. Motivating Example

Let us consider a simple web server architecture that processes HTTP requests such as the Apache Web Server or the Microsoft IIS solution. In such architectures, various quality properties can be identified: security level, response time, memory footprint, relevance of the response, etc. Objectives that must be reached for each of these properties depend on the environment state and can depend on each other. For instance, reaching a good security level is important but the required filters increases the web server response time. Classical non adaptive approaches make tradeoffs between these properties. In some cases, it could be better to increase the security level according to the number of malicious requests detected in the environment whereas in other cases, decreasing it in order to achieve a minimum response time is preferable. From an architectural perspective, one emerging solution is to allow the architecture to be adapted with respect to the environment state. Nowadays, to describe self-adaptive applications, designers define an arbitrary number of possible architectural configurations and associate each of them to a particular state of the environment [4, 19, 14].

For instance, considering the aforementioned security level issue, the solution consists in choosing a threshold on the number of malicious requests that triggers a specific adaptation. This adaptation is often a configuration switch or a hardcoded number of configuration adaptations. Considering all the environment variables influencing the system, this can lead us to specify a large number of configurations or adaptations. To highlight this, we identified some architectural modifications linked with specific environment states in our HTTP web server:

1. `Filter` such as SQL code detectors, or undesired URL detectors can be deployed when the number of malicious requests increases, or when the kind of retrieved content change. The introduction of such `Filter` in the architecture increases the web server security level, but also increases its response time and the system used memory. Different kinds of `filters` can be identified, for instance, it exists URL `filters`, SQL `filters`, ...

2. It can be decided to use secured or unsecured `Receiver` depending on the number of malicious request. The use of secured `receiver` increases the security level but also the response time.

3. A `cache` can be deployed when the density of requests increases in order to reduce the response time. Because some request responses are taken from the `cache`, this can lead to a response which is not the freshest one. In consequence, deploying a `cache` also decreases the relevance of the response quality. Moreover, different kinds of `Caches`, that use different algorithm can be identified. For instance, one can identify the MRU (Most Recently Used) `Cache`, the LRU (Last Recently Used) `Cache` or the LFU (Last Frequently Used) `Cache` .

4. Several `StorageServers` can be deployed in the overall architecture. They contribute to improve relevance of the response quality provided by the web server and to reduce its response time. However, they negatively impact memory usage. There exist various kinds of `StorageServers` depending on the way information is stored. For instance, one can identify RDBMS (Relational DataBase Management System) based `StorageServer` or file based `StorageServer` Moreover, when various servers are deployed, different dispatching policies [17] might be applied :

   - A basic *load balancing* algorithm may be used to dispatch requests fairly among the various data servers.
   - Performance hints might also be used to dispatch requests to the best server, such as the one with the shorter response time for instance.
   - The data contained in requests may be also used as a criteria to dispatch requests to servers. A server which stores all the large video files, could receive all the request referring to video files.

From a designer point of view, we intuitively extract a primary configuration as well as a number of adaptation rules from the previously identified architectural variations. The primary configuration, suitable the majority of the time (i.e. linked to the environment state assumed to be the most frequent one) uses at the same time one unsecured receiver two `dataServers`, an URL `filter` and a MRU `cache` (see figure 1).

When the load of the server is increasing with non malicious requests beyond a specific threshold, the size of the `cache` is replaced with a bigger one and a SQL `filter`

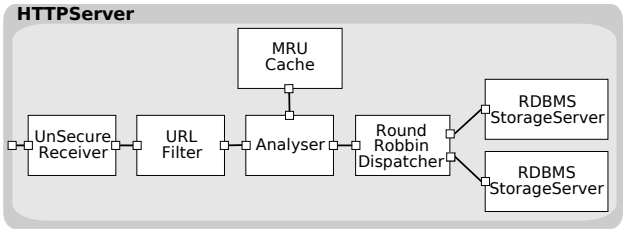

Figure 1. One possible configuration for the Web server

is added to the application. This adaptation leads to better response time. Aside, when there is not enough free memory, then the new configuration can decide to use at most one `StorageServer`.

This example highlights the fact that a designer identifies a finite number of configurations from a number of adaptation rules (or configuration switching rules). In the above scenarios, we have overviewed three configurations, each linked to a specific environment state defined through human-chosen thresholds. However, in the above example, the identified architectural variation points and their related variants define more than 72 possible architectural configurations. Naturally, if we consider an arbitrary number of servers, filters, receivers, etc, the number of induced configurations may become quickly very high. As a result, the designer may overlook one particular configuration which may be optimal for some environment state.

Our approach proposes to avoid enumerating configurations with respect to the environment states, by modeling first the variability space of a self-adaptive application. Our approach also takes into account the various architecture topologies which can be built from a given selection of components. We outline our approach in the next section.

3. Approach Overview

As explained, most of self-adaptive systems are currently designed in an ad-hoc manner using low-level and scripts or APIs. We claim that a systematic identification of variation points and of their related variants helps to cope with the complexity of self-adaptive systems. We identify three distinct spaces of variability in the development process of self-adaptive systems, which must be clearly defined to get an efficient description of self-adaptation. Figure 2 depicts these three variability spaces.

3.1. Functional Variability Space

First, the designer must identify the set of features which are used in the system and their related variants. The key point of self-adaptive systems is that they might involve several variants of the same feature and the final system must use the relevant variants with respect to changes in its environment. Organizing and reasoning about features must be achieved using features diagrams since they aim to express variability points and their related variants. Figure 2 shows a possible feature diagram for a Web server and the associated variability space (depicted as a question mark). Reasoning on those diagrams enable the computation of the *Functional variability Space* that is to say the set of possible feature configurations.

3.2. Platform Variability Space

A single feature diagram is not enough to be efficiently used to design self-adaptive software. Features are only abstractions of software pieces: A single component (i.e.. a runnable artefact) commonly implements a set of abstract features and a single feature might be implemented by several components. In Figure 2, the component which implements a feature are shown thank to an arrow which connects a feature to a component. Thus, in order to fill the gap between the abstract feature diagram and the set of legacy components, the designer must describe the relationship between each feature and the legacy components which implement it. The relationship builds the second space of variability, so-called *Platform variability Space* and is shown as the second question mark on Figure 2. Moreover, a specific execution platform cannot only be described by a set of available components. It might also entail some specific constraints on ressources: memory, CPU, bandwidth, etc. Those constraints must be taken into account to compute and define the platform variability space.

3.3. Topological Variability Space

The topological variability space relates to a set of particular configurations which might be deployed at runtime, that is to say the set of bindings between the selected components (See Figure 2). In the web server example, a SQL Filter can be inserted after the receiver, after the analyzer, after the dispatcher, etc ... In existing literature [20, 6, 7], such configurations are fixed and are not computed at runtime. Contradictorily, we do not describe any configuration. However, we offer the possibility to the self-adaptive application architect to express some constraints on the topology of the configuration. These constraints may concern the relationships between elements (i.e. a receiver should be connected to a storage server) or to enforce specific architectural patterns to take into account during design. We call *topological invariants* this kind of constraints.

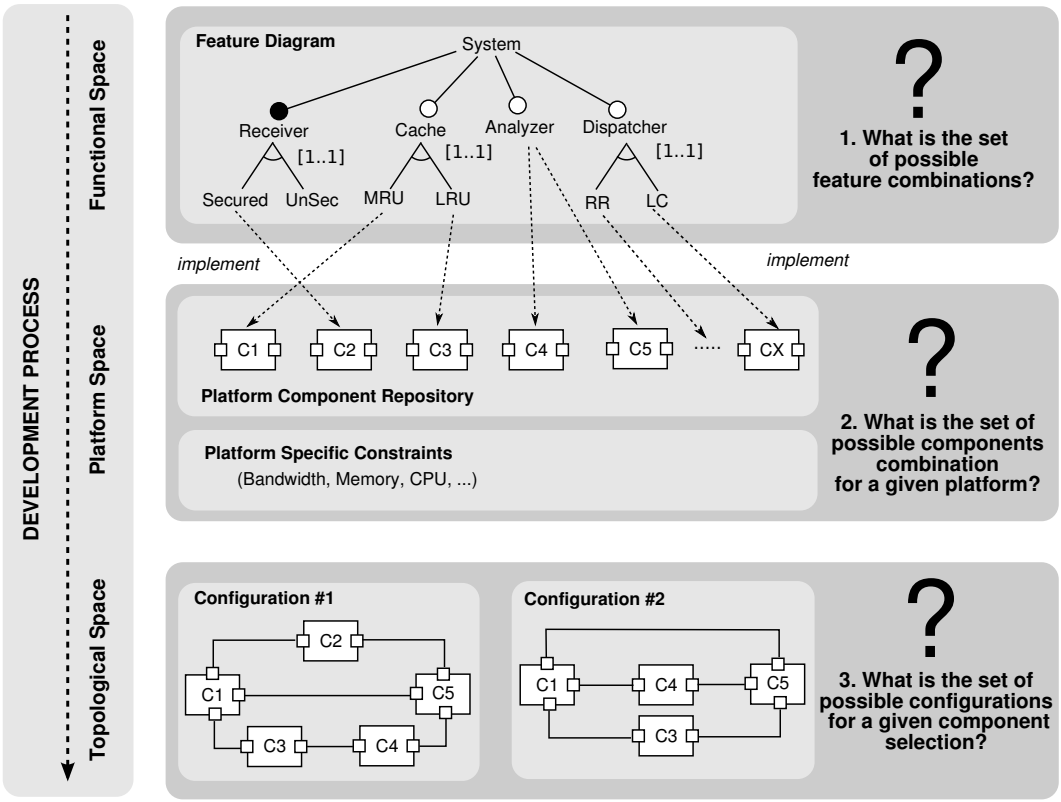

Figure 2. Defining the variability space of self-adaptive systems

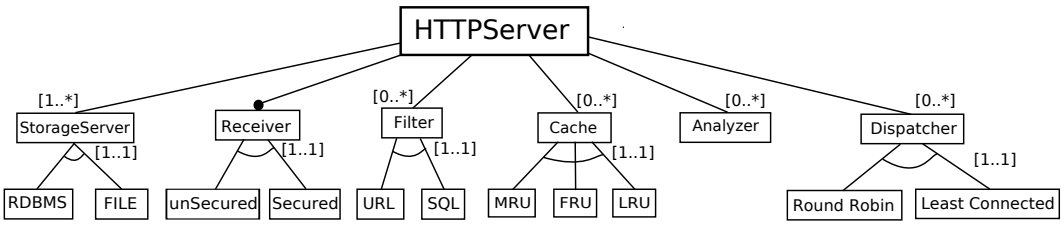

Figure 3. Feature Diagram for HTTPServer

4. Models

4.1. Overview

Functional space. To model the functional space of a self-adaptive architecture, we use a feature diagram [9] which is a popular technique to represent variability in product-lines. As noted by Schobbens et al [15], there are many variants of feature modeling notations. We chose to use the notation proposed by Czarnecki et al. [2] because it offers to model variability in terms of *cardinality* amongst features. They distinguish two kinds of cardinality; the first kind, called *feature cardinality* relates to how many instances of a feature are allowed in a product. We use feature cardinality to set how many components realizing this particular feature can be involved in a configuration. The second kind of cardinality, called *group cardinality*, enables to specify how many children can be selected for a given feature. As demonstrated by Schobbens et al [15] it is rather a concise way to model variability operators; for example the alternative (`xor`) operator is represented by group cardinality [1..1], or operator by [0..1] etc.

In this space, we mainly focus on functionality. This means that we do not model variations of feature that would actually corresponds to variations in non-functional properties of a component. For example, the size of a cache (big,

small, medium) will not be modeled but variations in their functionality (such as the algorithm they implement) have to be modeled in the feature diagram. Figure 3 shows the resulting feature diagram for `HTTPServer`. The root feature identifies the modeled application. Our application is mandatorily constituted by at least a one component implementing the `Receiver` feature which may support security or not. There is at least one storage server which can persist their data using RDBMS or files. Other features are optional.

Platform space. In order to model our components, we used an high abstraction ADL (Architecture Description Language) express in terms of components, ports and interfaces. This way, models expressed in this ADL can be easily translated in terms of UML 2.0 components [13] but also in terms of other component models. Since we are mainly concerned by the overall topology of the architecture and not internal component properties, components are considered as black-boxes. Connection between components are made through their ports. To each component is associated a sequence of costs which help the designer to determine if they can fit to a particular platform when defining the topological space. They take the form of non-functional properties covering issues such as bandwidth, memory or CPU usage. These properties are expressed in the same terms that platform-specific constraint to ease the allocation.

Topological space. Topological invariants are modeled via a UML collaboration. When N components are in a same collaboration, it signifies that there exists at least one path in the configuration that links together one instance of each of the N participants. Figure 4 depicts the topological invariant for our example. It specifies the following constraint: "For each `StorageServer` Component, there must exist a path[1] to a `Receiver` Component".

Figure 4. Topological Invariant for `HTTPServer`

Mapping features onto components. We do not seek to establish a one-to-one mapping between feature and components. This is due to the strict separation between the functional and platform spaces. Therefore several components can implement the same feature but with possibly different sequence of costs. We have defined the following rules for the mapping:

- Mappings can only be established from leaf features (i.e. features having no child) to components defined in the the platform space,
- A given component cannot implement two features. Although it is technically possible to do so, this raises the issue of assigning costs to a given component. For example, in Figure 2, considering `C3` both as an unsecured receiver and a MRU cache prevent the assignment of the memory attribute globally: is the memory value concerns one feature, all ? This would force components' designers to set those values for each feature therefore taking into account components' internals which contradicts our "black box" approach on components (see above).

Mappings are given in the form of tuples in which the first element represents the feature name and the second the set of components implementing it. For example, the Analyzer feature will be mapped as follows: `<Analyzer,{C4,C5}>`.

4.2. Formalization

In order to safely use the proposed models, it is important to ensure that the topological space is bounded. If the topological space is bounded, it signifies that the number of solutions that can be inferred from the models is finite. To prove that the topological space is bounded, a formal proof is given below.

First, we start by formally defining the notion of configuration, component and port. A component is noted $Comp$ and corresponds to a finite number of ports, each noted P (Cf definition (1).

$$Comp = \{P_1, P_2, ..., P_p\}, \forall p \in \mathbb{N}^* - \{\infty\} \quad (1)$$

To each port is associated a cardinality specifying the number of possible connectors that can be attached to it. So a port is defined in (2) by a couple of positive or null integers.

$$P = \{Cmin, Cmax\}, \forall Cmin, Cmax \in \mathbb{N} - \{\infty\} \quad (2)$$

A connector (noted Con) is a bidirectional link between two ports and is so defined in (3) as a couple of ports:

$$Con = \{P_1, P_2\} \quad (3)$$

A configuration Cf is a couple of sets. The first one is a set of components and the second one is a set of connectors.

[1] By "path", one should understand a connection with possibly intercalated Components

We define a configuration Cf as stated below: (see 4):

$$
\begin{aligned}
Cf = \{ & && \text{(4a)}\\
& \{Comp_1, Comp_2, ..., Comp_n\}, && \text{(4b)}\\
& \{Con_1, Con_2, ..., Con_q\} && \text{(4c)}\\
& \} \quad \forall n \in \mathbb{N}^{+*}, q \in \mathbb{N}^{+} && \text{(4d)}\\
& \wedge \forall i \in [1;q], Con_i = \{pi_1, pi_2\} \; where \; pi_1 \neq pi_2 && \text{(4e)}\\
& \wedge pj_1 \in Cj1 \wedge Cj1 \in \{Comp_1, ..., Comp_n\} && \text{(4f)}\\
& \wedge pj_2 \in Cj2 \wedge Cj2 \in \{Comp_1, ..., Comp_n\} && \text{(4g)}\\
& \wedge \forall j \in [1,n], k \in [1,p], l \in [1,q], && \text{(4h)}\\
& \quad \forall P_k \in Comp_j, \exists P_k \in Con_l && \text{(4i)}\\
& \wedge \forall c \in [1,p], d \in [1,q], && \text{(4j)}\\
& \quad \exists \; (\; Set(Con) \leq (Cmax \in P_c) && \text{(4k)}\\
& \quad \wedge Set(Con) \geq (Cmax \in P_c) \;) && \text{(4l)}\\
& \quad where \; Set(Con) = \sum_{d=0}^{d=q} (Con_j \wedge (P_c \in Con_d)) && \text{(4m)}\\
& \wedge \forall a, b \in [1,q], a \neq b \Rightarrow Con_a \neq Con_b && \text{(4n)}
\end{aligned}
$$

Informally, a configuration is the union of a non-empty set of components ((4b) and (4d)) and a possibly empty set of connectors (((4c) and (4d))) where each connector is a couple of nonidentical Ports ((4e)) belonging to a component of the configuration ((4f) and (4g)). Moreover, there are not two identical connectors in a configuration ((4n)) and each port of each component is connected ((4h) and (4i)). Finally, the number of connectors referencing a given port depends on the port's cardinality ((4j) to (4m)).

We now define a component repository, Rc, as a set of nonidentical components:

$$
\begin{aligned}
Rc = \{Comp_1, Comp_2, ...Comp_m\}, \forall m \in \mathbb{N}^* \\
\wedge \forall i, j \in \mathbb{N}^*, i \neq j \Rightarrow Comp_i \neq Comp_j
\end{aligned}
\tag{5}
$$

By considering our modeling of the system, a configuration is a set of components where each Component belongs to a component repository. Consequently, a set of all possible configurations noted $Set(Cf)$ is made up of components from a specific component repository. Considering a component repository noted $Rc1$, we define the set of all possible configurations as follows:

$$
\begin{aligned}
Set(Cf) = \{ & \\
& \{\alpha_1.Comp_1, \alpha_2.Comp_2, ..., \alpha_n.Comp_m\} \\
& \{Con_1, Con_2, ..., Con_q\} \\
& \} \quad \forall m \in \mathbb{N}^*, q \in \mathbb{N} \\
& \wedge \forall i \in [1;q], \exists Con_i = \{pi_1, pi_2\} \; where \; pi_1 \neq pj_2 \\
& \wedge pi_1 \in Comp_{i1} \wedge \alpha_{i1} \neq 0 \\
& \wedge pi_2 \in Comp_{i2} \wedge \alpha_{i1} \neq 0 \\
& \wedge \forall j \in [1,n], k \in [1,p], l \in [1,q], \\
& \quad \forall P_k \in Comp_j, \exists P_k \in Con_l \\
& \wedge \forall c \in [1,p], d \in [1,q], \\
& \quad \exists \; (\; Set(Con) \leq (Cmax \in P_c) \\
& \quad \wedge Set(Con) \geq (Cmax \in P_c) \;) \\
& \quad where \; Set(Con) = \sum_{d=0}^{d=q} (Con_j \wedge (P_c \in Con_d)) \\
& \wedge \forall a, b \in [1,q], a \neq b \Rightarrow Con_a \neq Con_b \\
& \wedge \alpha_i \in \mathbb{N} \forall i \in [1,m] \\
& \wedge \forall k \in [1,m], Comp_k \in Rc1
\end{aligned}
\tag{6}
$$

Informally, a possible configuration is made up with the set of all available components in the component repository $Rc1$. Each of these components can be used 0 or more times depending on the associated value of α. Moreover, a possible configuration must conform to the definition given in (4). Consequently, the set of possible configurations is the binomial combination of K possible components in a component repository that contains M components. Since $alpha$ can be greater than 1, the number of possible sets of components is a combination where repetition are allowed and order is discarded. For each set of components, there exists various possible sets of connectors. In the following definition, $NbP(Comp)$ is the number of ports of the component $Comp$. Consequently, the number of possible configurations denoted $Size(Set(Cf))$ is given by (7):

$$
\begin{aligned}
Size(Set(Cf)) =_{M+K-1} C_K * \prod_{i=1}^{NbPCf-1} (NbPCf - i)^2 \\
where \; NbPCf = \sum_{j=0}^{K} (NbP(Comp_j)) \\
\wedge \forall j, Comp_j \in Cf
\end{aligned}
\tag{7}
$$

From definition (6), it appears that the configuration can be composed of an infinite number of components if the value of a given α tends to ∞. Moreover, it can yield an infinite number of possibilities if m tends to ∞. In these cases,

the set of possible configurations can also be infinite. To avoid working on an unbounded number of configurations, we have to add a constraint such that both m and $\alpha_i \forall i \in [1, m]$ can not tend to infinity. for this purpose, we make the assumption that the size of a configuration is limited (this assumption fits a great majority of self-adaptive systems). Formally, the size of a configuration Cf is noted $Size(Cf)$ and the size of a component is noted $Size(Comp)$ and is strictly greater than 0. We consider here that a connector does not use memory space. From definition (6) we obtain:

$$Size(Cf) = \sum_{i=1}^{m} (\alpha_i.Size(Comp_i)) \\ \wedge \forall i \in [1, m], Size(Comp_i) > 0 \quad (8)$$

The constraint on the size of the configuration is then:

$$\sum_{i=1}^{m} (\alpha_i.Size(Comp_i)) \leq S \quad (9)$$

In (9), because $Size(Comp) > 0$, if $\sum_{i=1}^{m}(\alpha_i)$ increases, then $Size(Cf)$ increases. Consequently, bounding the size of the configuration bounds the number of component involved in a configuration. Regarding to the number of possible configurations defined by (7), it implies that the number of possible components noted K is bounded. For a constant K, $Size(Set(Cf))$ increase when M increase. Consequently, M must be bounded to ensure that the first term of (7), i.e. ${}_{M+K-1}C_K$, is bounded. Moreover, because the number of ports in a component is strictly finite, $NbPCf$ defined in (7) is bounded. Consequently, considering that there are not two identical connectors in a unique configuration and because all ports in the configuration must be connected, the second term of (7), i.e. $\prod_{i=0}^{NbPCf-1}(NbPCf - i)^2$ is also bounded.

As a result, this formalization ensures that, if the number of components in the component repository is bounded, if the number of ports for each component is also bounded, and if we restrict the configuration to use a finite amount of memory, then the number of possible configurations is finite.

5. Related Work

The general problem of mapping feature to the architecture have been addressed by several works. Kang et al [10] extends the FODA [9] approach with general guidelines to model the architecture. Sochos et al [16] provides a two-step approach in order to map architectural elements (implemented as plugins) with features. From one feature model, a sequence of transformations is performed in oder to ease the mapping by ordering and defining dependencies between features. Then, a one-to-one mapping between features and plugins is made. Czarnecki et al [1] propose a detailed approach to map features on various model including UML class and activity diagrams.

However, the above works were not meant to build *Dynamic Software Product Lines* (DSPLs), therefore the barely provides means to consider environmental properties and platform variability. Van Gurp et al [20] early discussed the notion of variability at various levels of abstraction (from requirements to runtime) motivating the need to define and organize variability mechanisms across these levels. However they do not provide specific modeling solutions in the case of DSPLs.

Gomaa et al [6, 7] proposed a UML-based profile and a process to model DSPLs. In particular they handle reconfiguration as a sequence of patterns on a fully designed architecture. Our approach differs in that we do not seek to determine all the possible configurations. Therefore, we need to separate variability definition from architectural models.

Trinidad et al [18] present a modeling approach to build DSPLs. They also base the variability description on a feature model which is mapped on UML 2.0 component architecture. Each feature is mapped to a component and variation points are ensured by *relationship* components whose main role is to propagate reconfiguration decisions performed by a general configurator component. However, they make the supposition that the feature model is built while thinking about the DSPL architecture (e.g. one-to-one relationship between feature and component) which is not a clear separation of concerns between functional and architectural dimensions. Furthermore, they do not provide any means to model platform/environment properties which trigger reconfiguration decisions.

Lee and Kang [11] propose a global approach for the engineering of dynamically reconfigurable products in a product-line fashion. In particular they introduce the notion of *binding unit*, a grouping of features which are used to identify architectural components. They also give several guidelines to build dynamically reconfigurable architectures, some hints about environment modeling (context) and considerations about how a configurator should work. However they remain at the general level with respect to the models employed and did not provide a formalization of the models as we did.

Hallsteinsen et al [8, 3, 5] also provide an holistic view of the engineering of DSPLs. They define variability directly in the reference architecture via a dedicated UML profile. This architecture comprises components which realize component types (variation points) via plans (variants) modeling a particular reconfiguration scenario. Reconfiguration is modeled through the composition of plans. Consequently these approaches are based on a specific topology called composition plan whereas our approach take into ac-

count the possible variability within this composition plan. It becomes possible to dynamically change the composition plan at design time.

Montero et al [12] focused on managing variability in business processes. In specific, they model how a process evolves (denoting a reconfiguration of the system) with respect to timing/scheduling constraints. We will integrate timing issues in future research while validating our approach on concrete case-studies.

6. Conclusion

Accurately modeling the variation space of self-adaptive applications requires to properly identify the dimensions (functional, platform and topological) in which this space can be described. By combining software product line techniques with separation of concerns, we are able to provide a clear separation between these dimensions, and to propose models for each of those dimensions; functional dimension is modeled via cardinality-based feature diagrams which defines constraints on the set of components which are available to form a particular self-adaptive application architectural configuration and its possible reconfigurations. The platform space concerned with the definitions of the individual components in terms of costs as well as the definition of restrictions related to the platform on which they run. Finally, the topological dimension aim at defining the possible bindings configurations for given set of components. We also formally demonstrated that our variation space is bounded which opens the way to decision algorithms able to extract intersting configurations from this variation space. Future research will concentrate on the definition of such a decision procedure as well as its validation in concrete situations. We also plan to refine our approach with respect to the complex mapping from features to components and to provide an integrated tool support for the whole approach.

References

[1] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach based on Superimposed Variants. In *4th international conference Generative programming and component engineering*, volume 3676 of *LNCS*, pages 422–437. Springer-Verlag, 2005.

[2] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.

[3] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[4] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[5] K. Geihs, M. Khan, R. Reichle, A. Solberg, S. Hallsteinsen, and S. Merral. Modeling of component-based adaptive distributed applications. *Symposium on Applied Computing*, pages 718–722, 2006.

[6] H. Gomaa. Feature Dependent Coordination and Adaptation of Component-Based Software Architectures. In C. C. J. M. Murillo and P. Poizat, editors, *ECOOP Workshop on Practical Approaches for Software Adaptation* , pages 45–52, Berlin, Germany, 2007.

[7] H. Gomaa and M. Hussein. Model-based software design and adaptation. In *SEAMS: Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Washington, DC, USA, 2007. IEEE Computer Society.

[8] S. Hallsteinsen, E. Stav, A. Solberg, J. Floch, S. ICT, and N. Trondheim. Using product line techniques to build adaptive systems. In *Software Product Line Conference*, 2006.

[9] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Nov. 1990.

[10] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[11] J. Lee and K. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Software Product Line Conference*, Aug. 2006.

[12] I. Montero, J. Pena, and A. Ruiz-Cortes. Representing Runtime Variability in Business-Driven Development Systems. *Conference on Composition-Based Software Systems (ICCBSS 2008)*, pages 228–231, 2008.

[13] OMG. Unified Modeling Language Superstructure (version 2.1.1). Technical Report formal/2007-02-03, Object Management Group, February 2007.

[14] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Syst.*, 26(2):161–197, 2004.

[15] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

[16] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In *Conference on Object-Oriented and Internet-Based Technologies*, Erfurt, Germany, 2004. Springer.

[17] Y. Teo and R. Ayani. Comparison of Load Balancing Strategies on Cluster-based Web Servers. *Transactions of the Society for Modeling and Simulation*, 77(5-6):185–195, 2001.

[18] P. Trinidad, A. Ruiz-Cortés, and J. P. na. Mapping feature models onto component models to build dynamic software product lines. In *International Workshop on Dynamic Software Product Line*, 2007.

[19] G. Valetto and G. Kaiser. A Case Study in Software Adaptation. Technical report, 2002.

[20] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *WICSA*, 2001.