Using Constraint-based Optimization and Variability to Support Continuous Self-Adaptation

Carlos Andrés Parra, Daniel Romero, Sébastien Mosser, Romain Rouvoy, Laurence Duchien, Lionel Seinturier

Using Constraint-based Optimization and Variability to Support Continuous Self-Adaptation

Carlos Parra[1], Daniel Romero[1], Sébastien Mosser[2]*, Romain Rouvoy[1], Laurence Duchien[1], Lionel Seinturier[1]†
Université de Lille 1, LIFL CNRS UMR 8022, INRIA Lille-Nord Europe, France
[1]{name.lastname}@inria.fr, [2]sebastien.mosser@sintef.no

ABSTRACT

Self-adaptation is one of the upcoming paradigms that accurately tackles nowadays systems complexity. In this context, Dynamic Software Product Lines model the intrinsic variability of a family of systems, and dynamically support their reconfiguration according to updated context. However, when several configurations are available for the same context, making a decision about the right one is a hard challenge: further dimensions such as QoS are needed to enrich the decision making process. In this paper, we propose to combine variability with Constraint-Satisfaction Problem techniques to face this challenge. The approach is illustrated and validated with a context-driven system used to support the control of a home through mobile devices.

1. INTRODUCTION

In the recent years, we have witnessed major advances in mobile computing. Modern mobile devices are equipped with sensors and network interfaces that make them versatile. This versatility leads to new requirements for software which has to be *context-aware— i.e.*, it has to monitor the events and information coming from its environment and adapt its architecture accordingly. This software adaptations may affect two phases of the software life cycle: design and runtime. During the design phase, the elements required for the adaptation are prepared in advance. At runtime, the adaptation of applications is based on the context. An approach considering both phases enables the development of context-aware applications that can be easily adapted.

Therefore, in this paper we propose an approach for building context-aware applications based on three paradigms: *(1)* variability management for the design phase, *(2)* Dynamic Software Product Lines (DSPL) [7] for the runtime phase and *(3)* Constraint Satisfaction Problem (CSP) methods applied also at runtime phase.

*SINTEF ICT
†Institut Universitaire de France

SAC'12 March 26-30, 2012, Riva del Garda, Italy.

At the design phase, adaptations can be considered as multiple configurations that an application may have. One way to deal with multiple configurations are feature models, a useful tool from SPL engineering. An adaptive application can be represented with a feature model that defines different products formed by two parts: *(1)* a *kernel* representing the commonalities (or mandatory features), and *(2) variation points* that can take different values (*i.e.*, variants) and that can be added or removed due to an adaptation. Every configuration is then formed as a set of selected variants.

For the runtime phase, DSPLs [7] enable software products to switch their configuration according to a set of events like updates in the context. For the configurations to be achievable at runtime, we specify an explicit link between every variation point in the feature model and a piece of context. However, it is possible that multiple configurations from the DSPL satisfy the context. Different analysis are required to choose the new configuration. Furthermore, the design information does not include the constraints of the specific implementations for the identified variation point.

To overcome these limitations, we propose a new decision making approach based on CSP techniques. CSP enables the decision making process to consider new elements to choose a configuration. In particular, it is possible to model the selection of a new configuration as a CSP, where we search to optimize the selection of the available variants by regarding different dimensions, *e.g.*, *Quality of Service* (QoS), resource consumption and reconfiguration cost. As CSP mechanisms may suffer from combinatorial explosion issues, our contribution includes a feature representation that bounds the exploration space during the adaptation, and consequently limits this phenomenon.

The rest of this paper is organized as follows. In Section 2, we motivate our work by using a smart-home scenario. Then, in Section 3 we introduce our proposal for developing context-aware applications. In Section 4 we present the modeling of the problem using CSP. Next, in Section 5 we validate our approach by performing different tests. Section 6 discusses some related approaches. Finally, Section 7 presents the conclusions and perspectives of our work.

2. MOTIVATION & CHALLENGES

To introduce the challenges in this paper, we present an adaptive application called MobiHome. It enables mobile devices to access different services in smarthomes. The application can be adapted using three different types of context as follows: *(1)* Internet bandwidth, *(2)* availability of services, and *(3)* battery level of the device. The architec-

ture of the application is separated in independent modules, which can be added, removed, or replaced. Figure 1 illustrates the architecture of MobiHome, using the *Service Component Architecture* (SCA) graphical notation[1].

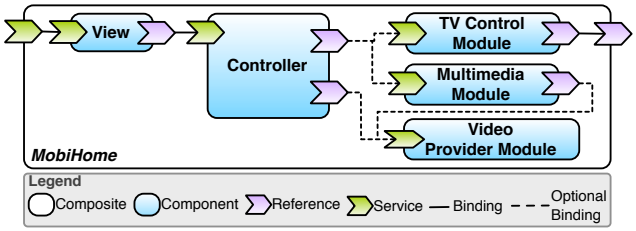

Figure 1: MobiHome Architecture.

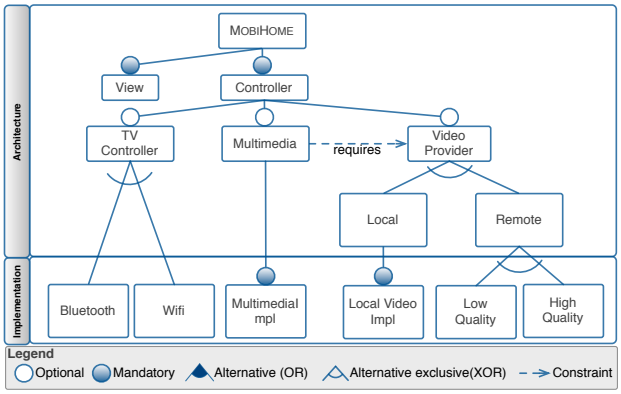

Figure 2: Feature model.

The architecture of Figure 1 includes the `View` and `Controller` components to offer a user interface. These components are always present. The architecture also has the `TV Control`, `Multimedia`, and `Video Provider` modules. The first module enables the mobile device to control a television. The second module accesses a multimedia server to offer rich content to the users. Finally, the third module is used to visualize videos on the mobile device. This module can be provided in two different ways: local or remote.

Regarding the context information used to choose the best configuration of MobiHome, the `Multimedia` module depends on the availability of a multimedia service. To choose between `Local` and `Remote` video providers, MobiHome uses two main conditions: if the battery level is under 50% the `Local` module is preferred, and if the bandwidth is over 400kb/s the `Remote` module is preferred. It is important to notice here that these conditions can be both satisfied at the same time, or none at all, depending on the context. In such cases, the decision on which provider to use will depend on other parameters such as QoS and reconfiguration cost, as we explain in the following section.

Furthermore, for each module in MobiHome, it is possible to have several implementations. In particular, the `TV Control` component has two implementations: `Wifi` and `Bluetooth`. The former one enables the mobile device to control a *Universal Plug'n Play* (UPnP)[2] television. The latter one uses bluetooth to replace a traditional remote controller. The `Multimedia` module has one implementation allowing the display of rich content such as photos, and videos. Finally, regarding the implementations for the video providers, the `Local` module has one implementation whereas the `Remote` module has two different implementations: `Low Quality` and `High Quality`. The `Low Quality` implementation reproduces videos with a resolution of 240p. `High Quality` implementation allows for resolutions of up to 480p (higher resolution supported for most current mobile devices).

To define the configurations proposed by this architecture, in Figure 2 we create a Feature Diagram (FD). A FD [9] consists of *(1)* a hierarchy of features, which may be mandatory (commonality) or optional (variability), and *(2)* a set of constraints expressing inter-feature dependencies.

In the diagram, we divide the variability in two layers: architecture and implementation. Architectural variability represents the different modules of MobiHome, whereas implementation variability represents the different implementations of each architectural feature. In the first layer, we define mandatory features for the *Kernel* components, and optional features for the modules that can be added or removed. As `Local` and `Remote` video providers are intrinsically exclusive, we define alternative features (*i.e.*, only one has to be chosen) to model this fact. In the implementation layer, we use either mandatory features; when only one implementation is available, or alternative features; to select one implementation among a list.

To obtain the final configuration of MobiHome one has to select the modules and their implementation. Since context is used for deciding the modules needed in the application, it can be understood as a way to filter out the invalid configurations (*i.e.*, the ones that do not respect the particular context situation). However, context alone is not enough to decide about the complete configuration of MobiHome. One can encounter two different situations with respect to the configurations obtained from the context information:

1. **Multiple Candidate Availability**: This situation refers to the fact that context information chooses a given module, without specifying an implementation. For example, consider the context information for service availability indicating that it is possible to control a television. In this case we need to choose between `Wifi` and `Bluetooth` implementations.
2. **Conflicting Modules Selection**: This situation takes place when there is a contradiction of the restrictions between the modules, and the context. Such situation is due to the lack of a global view of the system implementation, where different developers can implement several parts of the system in an independent way. In this case, it is necessary to choose the best implementation, and also to resolve the conflicts between module constraints and context. For example, consider the previously discussed rules when bandwidth is over 400kb/s and battery is under 50%. In this case, context is telling us to choose both, local and remote video providers, which would lead to an architectural error. Therefore, it is necessary to respect the constraints between the modules, but also to satisfy, at least partially, the modifications according to context.

[1]SCA: `http://www.oasis-opencsa.org/sca`
[2]UPnP: `http://www.upnp.org/resources/documents.asp`

Challenges.

Regarding the previously situations, we propose an approach to deal with the issues related to the selection of the new application configuration. The main contribution of this paper refers to the definition of an adaptation process based on variability that selects the optimal configuration in terms of architecture and implementation. In particular, we answer the following questions:

1. *How to link context information and architectural variability?* This challenge refers to the definition of a decision making process that considers both context and variability. For any adaptation, this process has to compute the target configuration in terms of architectural features, for a given context information.

2. *How to select one implementation for the selected architectural variability?* The second challenge refers to the selection of one implementation for a given architectural configuration. In this case, context and variability are not enough to make a decision. It is necessary to define new dimensions that allow the process of decision making to select the optimal implementation.

3. PROPOSAL

In this section we present our approach which uses context for adapting an application at runtime. Autonomic Computing Paradigm [8] aims to build self-adaptive systems by using *Feedback Control Loops* (FCLs). Our contribution focus in the planning phase of these FCLS in order to define a new approach that always selects a valid and optimal configuration according to the current user context. To do this, we divide the planning phase in two sub-phases, one phase for variability and context analysis, and a second one for optimization.

3.1 Variability and Context Analysis

The first step of our planning contribution that links context and architectural variability (cf. Section 2) is built upon DSPL. In particular, we implement a DSPL through the use of *Feature Diagrams* (FD) and a composition based on *Aspect Oriented Modeling* (AOM) [6], where every variant in the FD is implemented with a model that can be composed at design or at runtime. The models allow us to modularize the architecture of the applications to have both the kernel, and the optional elements to be incorporated. Every model includes four parts: *(1)* an architectural part with a set of elements to be added, *(2)* a definition of a place where the architectural elements are going to be added in the kernel, *(3)* a set of modifications required to incorporate the architectural elements to the kernel, and *(4)* a reference to a context element (*i.e.*, an *observable*) that is used dynamically to trigger an adaptation.

A product of the DSPL is then represented as a selection of features in the FD (*i.e.*, a configuration). For the dynamic adaptation based on context, we deal specifically with the process of selecting and/or deselecting features at runtime. For that, we assume that a product has been built and that its architecture can be expressed in terms of selected variants from the FD representing both the kernel, and all the aspects selected for the given configuration. At runtime, the dynamic changes of the different observables (*e.g.*, the available bandwidth) have an impact on the architecture of the application and trigger the adaptation process. The first phase defines an algorithm (see Algorithm 1) that finds a *target* configuration, using as input a given context situation, expressed in terms of selected variants from the FD and the *current* configuration for a particular product.

The first step of the algorithm consists in creating a target configuration with the same variants as in the current configuration. Afterwards, the algorithm iterates over the updated observables. For the observables whose aspect belongs to the current configuration, the algorithm verifies if the new observable value is false to remove its referenced aspect model from the configuration. For the observables whose aspect model does not belong to the configuration, the algorithm verifies if the new observable value is true to add its referenced aspect model to the target configuration. After the variants of the aspects to add have been selected, and the variants of the aspects to remove have been deselected, the algorithm obtains the target configuration.

Algorithm 1 Adapter algorithm

Require: A set of updated context observables $\mathcal{C}$ and the current product configuration $P_{current} = \{F_1, F_2, \ldots, F_k\}$
Ensure: A target product configuration P_{target}

```
1: P_target ← P_current
2: for all (O_n ∈ C) do
3:   if (F_On ∈ P_current) then
4:     if (O_n.value() = false) then
5:       P_target.deselect(F_On)
6:     end if
7:   else
8:     if (O_n.value() = true) then
9:       P_target.select(F_On)
10:    end if
11:  end if
12: end for
```

3.2 Optimization based on Constraints

The variability analysis generates a configuration. However, even if we have a target configuration at the architectural level (cf. Section 2), this configuration may be valid or invalid regarding the constraint analysis. In either case, further analysis are required to decide which configuration and which implementation are used for the new target configuration of the product.

To face this challenge we propose a CSP-based strategy to decide the optimal configuration and implementation for the target configuration. We model the selection of an optimal configuration as a CSP. Our goal is to find the best implementation for a given configuration issued from the previous phase. To decide which is the most suitable implementation, we provide the flexibility of choosing between different dimensions to optimize, including but not limited to the cost associated with resource consumption (*e.g.*, memory or energy), the reconfiguration cost (*e.g.*, in terms of bindings that we need to add or remove) and the offered QoS [15] (*e.g.*, response time).

To reduce the complexity of the decision making process at runtime, and make the CSP problem solvable, we divide the variability set in two different parts, as shown in the example of Section 2. Architectural variability for the design phase, and implementation variability for the runtime phase.

Table 1: Objective functions for Optimization

QS	$\max\left(\sum_{i=1}^{i=\lvert\mathcal{V}\rvert}\sum_{j=1}^{j=\lvert\mathcal{I}_i\rvert}\sum_{k=1}^{k=\lvert\mathcal{Q}\rvert} {q_Q}_k(I_j) \times si_(I_j) \times s(VP_i) \times w_q(Q_k)\right)$
Cost	$\min(\lvert C_c - C_i\rvert + \lvert C_i - C_c\rvert)$
Cons	$\min\left(\sum_{i=1}^{i=\lvert\mathcal{V}\rvert}\sum_{j=1}^{j=\lvert\mathcal{I}_i\rvert}\sum_{k=1}^{k=\lvert\mathcal{R}\rvert} {r_R}_k(I_j) \times si(I_j) \times s(VP_i) \times w_r(R_k)\right)$

The goal of this separation is to avoid the combinatorial explosion of different configurations at runtime. For this reason, implementation variability is defined with a limited set of features. Consequently, during the runtime phase, the set of available configurations that can be obtained as a result of an adaptation does not grow exponentially, and improves the performance of the CSP solving time.

We consider the two situations identified in Section 2. In the situation 1, we select the best implementation according to a specific dimension. In the situation 2, we search a valid solution optimized but that does not satisfy completely the current context. In this way, we provide adaptation considering not only the current context but also dimensions for providing an optimized application that guarantees a better user experience. The next section presents in detail the CSP modeling for finding the optimal configuration.

4. MODELING THE PROBLEM

A CSP problem consists in a set of variables and their finite domains, which are associated by constraints limiting the values that they take. Additionally, to reduce the number of solutions, it is also necessary to define objective functions to be optimized—*i.e.*, maximized or minimized. In this way, we identify two variables: *(1)* $s(VP_i)$, indicating the selection or exclusion of the VP_i variation point in the configuration and *(2)* $si(I_j)$ that expresses if the implementation I_j is part of the configuration or is not. The constraints are related to the exclude and require relationships between variation points and implementations.

The objective functions enable us to select the most suitable configuration regarding specific dimensions. In particular, we present three functions for the optimization of the configuration selection in terms of QoS, reconfiguration cost and resource consumption. The idea is to offer flexibility of using different criteria in the context-based adaptation process to select the new configuration. Nevertheless, our approach can be easily extended to include new dimensions. Table 1 summarizes the objective functions.

4.1 Optimizing the Provided QoS

One of the dimension that we use for improving the result of the decision-making process is the QoS. The usage of the QoS for selecting the new configuration is a suitable alternative since we search to improve the user experience by means of the context-aware adaptation. Therefore, it is logical that we try to maximize the value associated with the QS function in Table 1. In this function, we use the ${q_Q}_k(I_j)$ expression to estimate the value of the Q_k QoS property offered by each implementation. This value is only considered in the function evaluation if the implementation and the variation point are part of the configuration—*i.e.*, if $si(I_j)$ and $s(VP_i)$ are 1. To obtain the weight or importance given to each QoS property, we use the expression $w_q(Qk)$.

For example, we can apply the QS function in *Conflicting Modules Selection* situations such as the selection of the `Video Provider`, when the battery level indicates that the `Local` module should be chosen but the bandwidth says the `Remote` module. To do this, for the `Video Provider` variants, we define a scale of 1 (low) to 3 (high) for the QoS *video quality* and *response time*. The `Local Video Impl` implementation provides 3 in video quality and 3 in response time. The `Low Quality` implementation has 1 in video quality and 2 in response time. The `High Quality` presents 3 in video quality and 1 in response. Therefore, we will choose `Local Video Impl` even if we do not respect the context partially. The resulting configuration is optimized regarding the service offered by the component implementation.

4.2 Optimizing the Reconfiguration Cost

Another dimension that we use is the reconfiguration cost. To compute this cost, we define the $Cost$ function to search the configuration requiring the minimal number of operations to be reached. In the function, we use the set differences $C_c - C_i$ and $C_i - C_c$, which provide the implementations of the points that must be removed and added, respectively. For example, to replace $C_1 = \{Bluetooth\}$ by $C_2 = \{Bluetooth, Low\ Quality\}$ or $C_3 = \{Bluetooth, HighQuality\}$, we make the differences $C_2 - C_1 = \{LowQuality\}$ and $C_3 - C_1 = \{HighQuality\}$, which indicate what implementations we need to add in both cases. On the contrary, we do not need to remove functionality because $C_1 - C_2$ and $C_1 - C_3$ are empty. We specific an arbitrary value for the *add* and *delete* operations of components in the architecture, which can be modified at runtime if required (*e.g.*, 1 in both cases).

4.3 Optimizing the Resource Consumption

For the resources, we define the $Cons$ function in Table 1, which computes the total resource consumption for a given configuration. We employ ${r_R}_k(I_j)$ to determine the consumption associated with each point implementation, value that will be only considered if the implementation and the respective variation point make part of the application configuration (by using s and si). Additionally, we use the w_r function to include in the computation the importance of the resource. These values are extracted from user preferences stating the relevance of different resources for the user (*e.g.*, CPU and memory consumption). If a resource is no relevant in the optimization process its weight is 0.

Considering the situation that requires the selection of the `TV Controller` variant (cf. Section 2), we give constant values to the CPU consumption of x_1 and x_2 for the `Wifi` and `Bluetooth` implementations respectively, where $x_1 > x_2$. In a similar way, we estimate the memory consumption in y_1 and y_2 for the `Wifi` and `Bluetooth` implementations respectively, where $y_1 > y_2$. Additionally, we give the same importance to both resources. This means that, if we wish to reduce the resource consumption, applying $Cons$, we obtain a better performance if we select the `Bluetooth` implementation.

4.4 Constraints in the Selection Problem

In order to respect the exclude and include relations, for minimizing the *Cons*, *Cost* and *QS* functions in *Conflicting Modules Selection* situations (cf. section 2), we have to satisfy the following constraints:

CO1. $\{\forall\ (VP_h, VP_j) \in \mathcal{E} :\ (I_k \in \mathcal{I}_h \wedge I_l \in \mathcal{I}_j \wedge I_k \in$

$C_i) \Rightarrow I_l \notin C_i\}$: All the *excludes* constraints, at the architectural level, are respected.

CO2. $\{\forall\ (VP_h, VP_j) \in \mathcal{R} :\ (I_k \in \mathcal{I}_h \wedge I_l \in \mathcal{I}_j \wedge I_k \in C_i) \Rightarrow I_l \in C_i\}$: All the *requires* constraints, at the architectural level, are respected.

CO3. $\{\forall\ (I_k, I_l) \in \mathcal{EI} :\ I_k \in C_i \Rightarrow I_l \notin C_i\}$: All the *excludes* constraints, at the implementation level, are respected.

CO4. $\{\forall\ (I_k, I_l) \in \mathcal{RI} :\ I_k \in \mathcal{C}_i \Rightarrow I_l \in C_i\}$: All the *requires* constraints, at the implementation level, are respected.

CO5. $\{\forall\ VP_j \in \mathcal{VPC}_i\ : |\mathcal{IC}_i^j| = 1\}$: Each selected variation point has one and only one implementation that has been chosen.

In *Multiple Candidate Availability* situations, we only need to satisfy the $CO3$, $CO4$ and $CO5$ constraints since the first subphase already guarantee the satisfaction of the first two.

The previous constraints represent the basic constraints to find a valid configuration. Such constraints are derived from the work performed at design and implementation phases. Therefore, by respecting these constraints, we can find a configuration that is part of the product family and at the same time is the most suitable according to context and the optimized dimension. Nevertheless, new constraints could be imposed. As we illustrate in Section 5, the definition of a high number of constraints makes the resolution process more expensive. Furthermore, if there are contradictions between constraints, the problem can become inconsistent.

5. VALIDATION

To validate the approach, we implemented MobiHome and the service containing the logic of CSP with SCA components running on top of the Frascati platform [14]. We chose this platform because, in addition to its usage of different technologies and provided support for distribution, it also presents reflective capabilities. Additionally, we also implemented in Java the feature model analyzer and the adapter algorithm. Finally, to implement the CSP logic, we benefit from the JaCoP solver[3].

In order to verify the efficiency of the CSP in the decision making process, we executed several tests by varying the complexity of the application. Specifically, in different tests we changed the number of variation points, variants and *requires* constraints. In Table 2, we present for each test, the number of configurations available at both the architecture and the implementation levels.

Regarding the CSP problem, we present the optimization of the QS function using two QoS properties. Each test was executed 1,000 times. We calculate the average time of the execution by excluding the first 100, which were considered as part of the warm-up. The first three tests (a, b and c) correspond to the two situations introduced in section 2. In the other tests, we created bigger product families to vary the complexity of the CSP. Analyzing the adaptation costs provided by Table 2, we see that in test a we obtained a cost of $1.0ms$ approximately, and in test b and c, $1.5ms$ and $1.6ms$, respectively. As expected, the CSP latency increases, not only because of the application size, but also because of the number of constraints. From these results, we observe that our approach has a good performance for simple applications having a reasonable quantity of constraints, variation points, and variants. On the other hand, considering more complex applications, we see that the overhead increases in a linear way approximately comparing configurations d, e, f and g. Even if we have an application with 50 variation points and 4 implementations for each one (cf. configuration g), we are able to find a suitable configuration in less than 1 second. This is a good property of our solution, because we can still deal with context-aware adaptation and decision making by applying CSP techniques at a reasonable cost.

[3]JaCoP: `http://jacop.osolpro.com/`

Table 2: Variability and CSP results

Test	VPs	Number of Variants by VP	Number of Constraints	CSP Latency (ms)
a)	1	2	2	1.0044
b)	3	1,1,2	3	1.4822
c)	3	1,1,2	5	1.5766
d)	5	2	5	2.1311
e)	10	4	5	5.5622
f)	20	4	5	10.5055
g)	50	4	5	27.7388

6. RELATED WORK

Several works propose the use of variability and SPL for adapting applications at runtime. In [2] are proposed SPLs for adaptive systems. In their approach, a complete specification of the context and supported changes has to be provided using a state machine. Each state represents a particular variant of the system and transitions between states define dynamic adaptations that are triggered by context changes. [4] proposes a DSPL with aspect models at runtime. They use aspect models to define features and feature constraints. Their approach links what they call *dynamic features*, representing late variation points in an SPL, to dynamic aspects. In our approach we propose a similar strategy for dealing with multiple configurations at runtime and product derivation based on context information. However, we improve this process by adding an implementation level where CSP is used to choose, the optimal solution based not only in context information, but also in a set of additional dimensions. In [3], authors tackle the problem of combinatorial explosion configurations for dynamic software product lines. They present a modular approach whose decision model combines utility functions, and various optimizations of the search space. Our approach can be considered as an alternative to this approach, however, we avoid the combinatorial explosion by restricting the kind of variability for the implementations at runtime.

Regarding the optimization, several works propose the use of parameters like QoS at runtime to define the best adaptation. For example, in [10] is proposed an approach formed by two parts: an MDA transformation chain for building adaptive applications, and a middleware system to make decisions about adaptation based on QoS information. In [1] authors define an approach to automate the selection of an optimal service configuration by exploiting two different methods: *i)* a greedy method and *ii)* a constraint programming method. Unlike these two approaches, our solution uses the CSP as a complement to the configuration process of the DSPL to deal with ambiguities due to context events and to the need of choosing a specific implementation.

The work presented in [12] uses Constraint Programming

to verify DOPLER variability models associated with SPLs. The DOPLER models are converted into constraint programs to ensure the consistency of derived products of the SPL. However, this approach focuses in the design time whereas we focus on the dynamic context-based adaptations.

The *FeatUre-oriented Self-adaptatION* (FUSION) framework [5] proposes a learning-based approach to build self-adaptive software systems. In particular, FUSION enables the tuning of the adaptation logic to unanticipated conditions by using a feature model. Because in our approach we also define a FD of the system, we can complement it by applying the FUSION concepts of dynamic adaptation.

Works in [11, 13] focus on the dynamic composition of Web Services by applying CSP techniques. [11] suggests a CSP solver extended with semantic Web concepts to enable the dynamic composition of web services in Semantic Web Environments. The authors propose a CSP ontology using OWL, which provides explicit semantics to define the CSP. The resulting solver is able of dynamically combining software agents that hosts the variables and constraints that make part of the problem. In our approach we could exploit this idea to make the modification at runtime of the selection problem easier.

The approach described by [13] defines a dynamic composition of Web Services based on Fuzzy DisCSP. In this kind of CSP, different constraints have different importance levels. In [13], authors propose an algorithm modeling provider preferences as fuzzy constraints and supporting the notions of preference priorities. In our proposal, we could benefit from the fuzzy constraints in order to deal with conflict QoS properties in the optimization of the QS function (cf. 4.1).

7. CONCLUSIONS

In this paper we presented an approach to optimize context-aware adaptations divided in two contributions. In the first one, related to variability, our approach represents an adaptive application as a set of architectural features. An adaptation is a reconfiguration of the different features selected. Like this, context events are related to architectural features and can be used to trigger the adaptation.

In the second contribution, we consider the selection of the best implementation for the context-aware application. To do this, our approach adds an implementation layer to the architectural variability. Using this layer, different choices can be made for the specific implementation of architectural features. This selection is based on CSP techniques considering dimensions like QoS. In our approach, we limit the number of implementations to avoid the combinatorial explosion in the number of configurations at runtime.

In terms of perspectives, we intend to consolidate the results obtained so far, following two directions. First, we will explore the reusability of our approach in terms of finer-granularity implementations. Thus far, each implementation is defined independently. However, several generic elements of such implementations could be created separately and used across different features. We believe that this could be also an important factor when evaluating the reconfiguration cost of each adaptation. Secondly, currently in our approach we only use context to decide which architectural features to choose. However, there may exist several dependencies between context events and one or more implementations. We would like to extend our approach to include this kind of dependencies as part of the CSP problem.

8. REFERENCES

[1] M. Beauvois, D. Belaïd, and G. Bernard. A planning framework for dynamic configuration in mobile environments. In *GIIS'07: 1st Int. Workshop on Seamless Services Mobility (SSMO)*, 2007.

[2] 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 Int. Workshop on Dynamic Software Product Lines (DSPL 2008)*, 2008.

[3] G. Brataas, S. O. Hallsteinsen, R. Rouvoy, and F. Eliassen. Scalability of decision models for dynamic product lines. In *SPLC (2)*, pages 23–32, 2007.

[4] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A Dynamic Software Product-Line Approach using Aspect Models at Runtime. In *5th Domain-Specific Aspect Languages Workshop*, 2010.

[5] A. Elkhodary, N. Esfahani, and S. Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proceedings of the 18th ACM SIGSOFT int. symposium on Foundations of software engineering*, FSE '10, pages 7–16, New York, NY, USA, 2010. ACM.

[6] R. France and J.-M. Jézéquel. Editorial for the special issue on aspects and model-driven engineering. *Transactions on Aspect-Oriented Software Development*, 2009.

[7] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[8] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu. The autonomic computing paradigm. *Cluster Computing*, 9(1):5–17, 2006.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[10] S. A. Lundesgaard, A. Solberg, J. Oldevik, R. France, J. O. Aagedal, and F. Eliassen. Construction and execution of adaptable applications using an aspect-oriented and model driven approach. In *Proceedings of the 7th IFIP WG 6.1 Int. Conference on Distributed applications and interoperable systems*, DAIS'07, pages 76–89, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] D. Maruyama, I. Paik, and M. Shinozawa. A flexible and dynamic csp solver for web service composition in the semantic web environment. In *Proceedings of the Sixth IEEE Int. Conference on Computer and Information Technology*, CIT '06, pages 43–, Washington, DC, USA, 2006. IEEE Computer Society.

[12] R. Mazo, P. Grünbacher, W. Heider, R. Rabiser, C. Salinesi, and D. Diaz. Using constraint programming to verify dopler variability models. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 97–103, New York, NY, USA, 2011. ACM.

[13] X. T. Nguyen, R. Kowalczyk, and M. T. Phan. Modelling and solving qos composition problem using fuzzy discsp. In *Proceedings of the IEEE Int. Conference on Web Services*, pages 55–62, Washington, DC, USA, 2006.

[14] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.

[15] X. Xiao. *Technical, Commercial and Regulatory Challenges of QoS: An Internet Service Model Perspective*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.