Run-Time Adaptation of Mobile Applications using Genetic Algorithms

Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes
University of Málaga, CAOSD Group, Málaga, Spain
{gustavo,pinto,lff}@lcc.uma.es

***Abstract*—Mobile applications run in environments where the context is continuously changing. Therefore, it is necessary to provide support for the run-time adaptation of these applications. This support is usually achieved by middleware platforms that offer a context-aware dynamic reconfiguration service. However, the main shortcoming of existing approaches is that both the list of possible configurations and the plans to adapt the application to a new configuration are usually specified at design-time. In this paper we present an approach that allows the automatic generation at run-time of application configurations and of reconfiguration plans. Moreover, the generated configurations are optimal regarding the provided functionality and, more importantly, without exceeding the available resources (e.g. battery). This is performed by: (1) having the information about the application variability available at runtime using feature models, and (2) using a genetic algorithm that allows generating an optimal configuration at runtime. We have specified a case study and evaluated our approach, and the results show that it is efficient enough as to be used on mobile devices without introducing an excessive overhead.**

***Index Terms*—Dynamic Reconfiguration, Context, Middleware, Feature Models, Genetic Algorithms, Autonomic Computing**

I. Introduction

Modern mobile applications demand services that support runtime reconfiguration, in order to adapt their behaviour to the continuous contextual changes that occur in their runtime environment. Although there can be several reasons to reconfigure an application, in the case of mobile applications, what normally drives their runtime adaptation is the necessity of optimizing their functionality to the availability of certain resources (e.g. battery, memory, CPU, etc.). Ideally, mobile applications should be able to manage such optimization autonomously, that is, they should be self-managed or self-adapted. In this sense, the main goal of the Autonomic Computing (AC) paradigm [1] is to endow distributed systems with self-management capacities.

According to the principals of AC, runtime reconfiguration of mobile applications involves (1) *monitoring* the runtime environment; (2) *analysing* the monitored information; (3) generating the reconfiguration *Plan* and (4) *executing* the plan.

All of these stages are driven by a *Knowledge* base, and all together are known as the MAPE-K loop. However, how to implement each of the MAPE-K loop functions is an open issue. In our proposal, we define a *Context Monitoring* Service (CMS), which is in charge of *monitoring* the environment and providing this information to a *Dynamic Reconfiguration Service* (DRS), which covers the *analysis* of the monitored information and the *generation* and *execution* of the reconfiguration *plans*. Both services are designed to be integrated in a middleware for adaptive applications development [2], although in this paper we mainly focus on presenting the details of how the DRS accomplishes the runtime reconfiguration of mobile applications.

Typically, after monitoring the contextual information, the analysis phase in the MAPE-K loop consists on deciding if the detected changes are significant enough as to initiate a reconfiguration of the application. If yes, a plan has to be selected/generated to switch from the running application configuration to a new one that fits the current available resources. Although there are other approaches that provide dynamic reconfiguration of applications at runtime, many of these approaches specify both the list of possible configurations and the reconfiguration plans at design-time [3], [4], [5], [6], [7], [8], [9]. This shortcoming limits the number of possible configurations and avoid generating the optimal ones.

Contrarily to these approaches, the DRS presented in this paper generates the application configurations and the reconfiguration plans automatically at runtime. Moreover, the generated configurations are optimal regarding the provided functionality and, more importantly, without exceeding the available resources (e.g. battery).

On the one hand, our DRS follows a Dynamic Software Product Line (DSPL) approach. DSLPs produce software capable of adapting to changes in user needs and resource constraints [10]. Concretely, the variation points of the application are specified using a Feature Model (FM) [11], which in our approach is available at runtime as part of the knowledge used by the DRS to perform the reconfiguration. Since the reconfiguration is defined in terms of variations on the application software architecture, our approach defines a special kind of FM that we name Architectural Feature Models (AFM). An AFM is a FM where features represent architectural elements – i.e. components and connectors that specify the interconnections among them.

On the other hand, when the availability of certain resources decreases or increases significantly, the DRS has to decide which architectural configuration provides the best functionality, while not exceeding the available resources. Since this can be formulated as an optimization problem, our approach proposes the use of a genetic algorithm (GA) to optimize the selection of architectural elements that will conform to the new configuration. In [12] the same kind of GA is used to

978-1-4673-4401-2/13/$31.00 © 2013 IEEE

optimize the selection of features of a SPL at design time. We use a similar GA, but executing it at runtime as part of our DSPL approach.

Finally, since our DRS is installed inside a mobile phone, we present some evaluation results showing that our approach is feasible and efficient for being executed with the fairly limited resources of a mobile phone, resulting in good response times and nearly-optimal architectural configurations. The DRS have been implemented in Android and tested on a Nexus 7 device.

The rest of the paper is organized as follows. The motivation of our approach, an overview of the main contributions and the case study used throughout the paper are presented in Section II. Then, the approach is further described in Sections III and IV. The results of our evaluation are presented in Section V, the related work discussed in Section VI and finally our conclusions and on-going work are described in Section VII.

II. Motivation and Approach Overview

In this section we show the motivation for our work by discussing a list of challenges that have to be taken into account for specifying the reconfiguration service. The basics of feature modelling, an overview of our approach and a case study are also presented.

A. Feature Modelling

A very popular technique for managing variability in SPLs is feature modelling. Although they are typically used to model the variability in the requirements specification phase, some authors [13], [14] have shown that feature models (FMs) can be successfully applied to manage variability in any phase of the software development life cycle, including the specification of the software architecture. FMs are organised in a hierarchical structure (Figure 1), where each feature is decomposed into children features, which can be connected to their parent individually using optional/mandatory connectors (if the children feature is optional/mandatory) or in groups (an OR group if some children features can be selected or a XOR group if only one children feature can be selected). Selecting a feature means that its parent is automatically selected too.

For instance, in Figure 1 FeatureA is decomposed in FeatureB and FeatureC. While FeatureB is mandatory and thus has to be present in all the generated configurations, FeatureC is optional – i.e. it is a variation point. Also, FeatureD and FeatureE are part of an OR group meaning that one or more features can be selected simultaneosly, while features FeatureF, FeatureG and FeatureH forms a XOR group and thus only one of them can be part of a particular configuration.

Finally, it is also possible to specify *cross-tree constraints* between features, which allows the definition of constraints which span the feature tree independently of the parent-child relationships. These constraints are difficult to manage visually and are usually specified using a textual notation. For instance, the FM in Figure 1 contains 2 cross-tree constraints. The first one means that, in case FeatureD is selected in a configuration, FeatureF should be selected too. On the other hand, the second

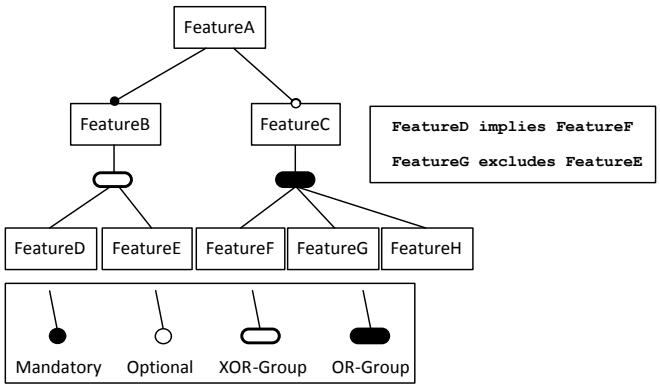

Fig. 1. Feature Model Example

one states that if FeatureG is selected, the feature FeatureE can not be included in the same configuration.

The use of FMs allows us to take advantage of their wide support ([15], [16], [17], [18]) and the existing tools (FAMA [19], Hydra [20], S.P.L.O.T. [21] or FeatureIDE [22]). Moreover, FMs are specified using formal languages, as for instance CSP (Communicating Sequential Process) [23]. This means that the visual representations are only for the purpose of facilitating the writing and understanding of the FMs, but then the tools automatically map this graphical representation into a CSP specification. This allows reasoning about variability, as well as other capacities of FMs such as the generation of valid product configurations, the quantification of the number of possible configurations, etc.

B. Challenges

In order to achieve our goal of building a DRS that reacts to the runtime contextual changes by optimizing the configurations according to the availability of certain resources (e.g. battery, memory, CPU), we have identified a list of challenges that must be taken into account:

Challenge 1: Modelling the system variability. In order to generate different configurations of a system it is necessary to model its dynamic variability at the appropriate abstraction level. Concretely, in the case of runtime reconfigurations we need to manage architectural configurations. In other words, a reconfiguration is normally specified in terms of changes on the application software architecture – i.e. on their components and connections. So, the challenge we address is to model variability at the architectural level, expressing the commonalities and variabilities of the application as part of the specification of its software architecture (see Section III).

Challenge 2: Reasoning about variability at runtime. Modelling the variability at the architectural level, as discussed in *Challenge 1*, implies a new challenge. As far as we know, the existing approaches to incorporate variability in software architectures, such as [24], [25], [26], or [27], lack the support provided by FMs to reason about the correctness of the variability and to generate a particular product configuration. In our approach we avoid this shortcoming by using an Architectural FM (AFM). In our proposal, an AFM contains

features that refer to components and connections of the software architecture, being able to ensure that any architectural configuration generated from the AFM is correct, in terms of the set of components and their connections. Furthermore, this AFM is generated off-line, which avoids introducing additional overhead. The way the AFM is generated (manual or automatic) is out of the scope of this paper and is not relevant for understanding our dynamic reconfiguration approach.

Challenge 3: Optimizing the architectural configuration. Mobile applications have scarce resources, so the challenge is to generate optimal configurations at runtime. We use an optimization algorithm that is able to find a nearly-optimal configuration taking into account the resource usage of the valid architectural configurations [1]. Concretely, the algorithm optimizes a *utility function* that quantify the architectural variation points according to a criterion specified by the Software Architecture (SA). This utility function typically refers to the general user satisfaction, although our approach is independent of the chosen utility function. For instance, the criterion can be the *precision* in the case of a component that is focused on providing location information, or the *quality* in the case of a component for video streaming. Because of its ability to fit well with optimization problems based on variability, the concept of utility function has been applied before in other proposals, such as MUSIC [7] and [6].

Challenge 4: Generating the reconfiguration plan at run-time. In our approach this challenge is straightforwardly satisfied. Since a configuration is specified as an array of bits (the output of the optimization algorithm), the reconfiguration plan to go from the running configuration to a new optimized one can be generated at runtime just by applying an XOR operation between the arrays of bits representing the source and target configurations (see Section IV).

Challenge 5: Executing the service in mobile environments. An important challenge of any service executing on a mobile environment is to minimize the resources (time, memory, CPU, battery) consumed by the service itself. In particular, for a reconfiguration service, the time is critical since, in order to be useful, applications must be reconfigured without appreciating the extra time employed for the reconfiguration process. Regarding this, in Section V we demonstrate that our DRS is fast enough to avoid harming the user response time or the performance of the system.

C. Our Approach

All the above challenges have been addressed in our approach, which is summarized in Figure 2. We propose a middleware in which the CMS and the DRS provide support for deploying adaptive applications by covering all the steps of the MAPE-K loop.

Knowledge. As shown in Figure 2, in our approach the knowledge is represented by (1) the AFM; (2) the current AFM configuration; (3) the software architecture; (4) the resource and utility information, and (5) the reconfiguration policy. The SA specifies the architectural variation points (*Challenge 1*) using an AFM (*Challenge 2*), as well as an estimation of the resource usage and the utility provided by the components of the architecture. This information provides an optimization criterion for run-time reconfiguration and, therefore, using it we can generate different configurations at run-time which maximize the utility of the application without exceeding the availability of a concrete resource, addressing the *Challenge 3*. The AFM can be generated manually or using automatic methods such as mapping algorithms or model transformations (see Section VII).

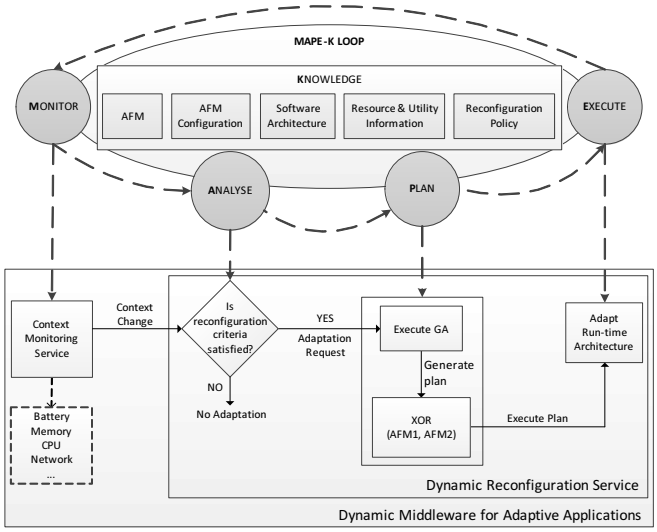

Fig. 2. Approach Overview

Monitor. The CMS provides the DRS with information about the evolution of the availability of a certain resource, such as the battery level or the memory. When a change is detected, the DRS is notified.

Analyse. When a Context Change event is received, the DRS analyses if the change is significant enough to trigger the adaptation process –i.e. if the reconfiguration criteria is satisfied. There can be several criteria for measuring the significance of a context change. For instance, a change in the battery level can be significant if it has changed more than a 5% since the last measurement or more than 10% per hour. Therefore, several reconfiguration policies can be defined, and the policy applied is part of the *Knowledge* base.

Plan. In case the analyser decides that the application needs to be adapted, the GA is executed in order to find a nearly-optimal configuration according to the current context. Then, the differences between the current AFM configuration and the new one are calculated, generating a plan for switching between them (*Challenge 4*). As it has been explained in Section II-B, calculating the difference between two configurations is quite straightforward since it is directly obtained by performing an XOR operation between both configurations.

Execute. Finally, the plan is executed in order to adapt the running architecture of the application.

[1]An exact algorithm cannot be used because the problem to be solved is NP-hard (non-deterministic polynomial-time hard)

D. Case Study

In the following sections we use a case study that consists of an application that assists attendees of international congresses, keeping them up to date with the latest news and providing several social facilities. The application provides the following variable set of services:

1) Information about events, stands, and the latest news about the congress.
2) Receive a video stream of keynotes or conferences in the mobile phone. The quality of the received video is variable (high, medium, low).
3) Check-in in the stands/events to track your activity. The technology used is variable and either NFC or Bluetooth may be used.
4) Information about your friends: location, visited events and stands, agenda. The location is obtained using GPS or WLAN, and the measuring rate is variable (high, low).
5) Exchange public messages or with your friends using a message board.

This application can be adapted according to user preferences (e.g. high quality of video is preferred), to the availability of the resources (e.g. WLAN is used because GPS is not available) or to the amount of consumed resources (e.g. use low quality of video because the mobile battery is low). In this paper we focus on this last kind of reconfiguration.

Figure 3 shows a component-and-connector view of the software architecture of our case study – i.e. components model the basic behaviour of the application and communicate with each other using connectors. All the connectors, except AuthorizationConn have been omitted from the figure for legibility reasons. For modelling the variability information we use the variability taxonomy of the Common Variability Language (CVL) [28], a domain-independent language for specifying and resolving variability, submitted to the Object Management Group, Inc. (OMG) for being considered as a standard language by IBM, Thales, Fraunhofer FOKUS and TCS. CVL allows the specification of variability over any model which has been defined using a MOF-based metamodel by means of *variation points*, which are bound to the elements of the base model. Basically, in CVL, a variability model and a realization model are defined. The variability model allows the specification of the variability related to the base model, while the realization model contains the information that is necessary to decide how the variability is resolved, resulting in a fully specified product without variability when it is executed. However, in our approach we only use the variability model, since the deployed configurations are chosen by our DRS using the genetic algorithm. The different kinds of variation points used in our case study are described in Table I. For instance, using CVL we define *optional* components (ObjectExistence variation point), different variants for a component (ObjectSubstitution variation point), parametrizable components (SlotValueAssignment variation point) and optional links between elements (LinkExistence variation point).

The main component of the architectural model is the CongressAssistant. On the one hand, it communicates with the DataAggregator component for accessing information about events, stands, news or for receiving a video stream of a conference. On the other hand, it communicates with the SocialManager component in order to take advantage of the social facilities of the application. The Location component is responsible for providing the location of the owner of the mobile device for tracking his/her position, and can be realized either by the Location_GPS or the Location_WLAN variants. The GPS variant measurements are more precise but it is also much more expensive regarding battery consumption. On the other hand, the CheckIn component can also be realized by CheckIn_NFC and CheckIn_Bluetooth components. As we can see in the figure, this is specified in the architectural model by applying the ObjectSubstitution variation points to the components and realizations.

On the other hand, the components Location_GPS and Location_WLAN have a configurable parameter, frequency, which defines the measuring rate. Another parameterizable component is VideoReceiver, in which the quality of the video stream can be set to different values which allows to find a balance between quality and resource consumption. To this end, the SlotValueAssignment variation point has been applied to the parameters of the components.

The architectural elements with an ObjectExistence variation point associated can be removed from the configuration. For instance, if the battery level is low, the Location component could be removed from the configuration. Then, the links between the SocialManager and Location components, which are not shown in detail in the figure, should be removed too. Our DRS detects when a connector or a component is not necessary and removes it automatically in order to ensure that the resulting configuration is always consistent. This is also the case of the Authorization component, which is not mandatory. Therefore, an ObjectExistence variation point is associated to the component. Furthermore, we can see that the LinkExistence stereotype has been associated to the links which connect the Authorization and the SocialRemoteAccess components because they are removed in case the connector is deleted from the architectural model.

As previously described, our approach is based on the optimization of an AFM. To this end, the SA can either manually specify the AFM, or map the information that is contained in the specification of the software architecture with variability into an AFM. The validity of our approach is independent from the way in which the AFM is generated.

III. Knowledge Base for Dynamic Reconfiguration

As described in Section I, all the stages of the MAPE-K loop are driven by a knowledge base. The knowledge that is managed in our approach is shown in Figure 2, and was discussed in Section 2. In this section, we focus on the AFM, which is one of the differences of our approach in comparison with other similar approaches.

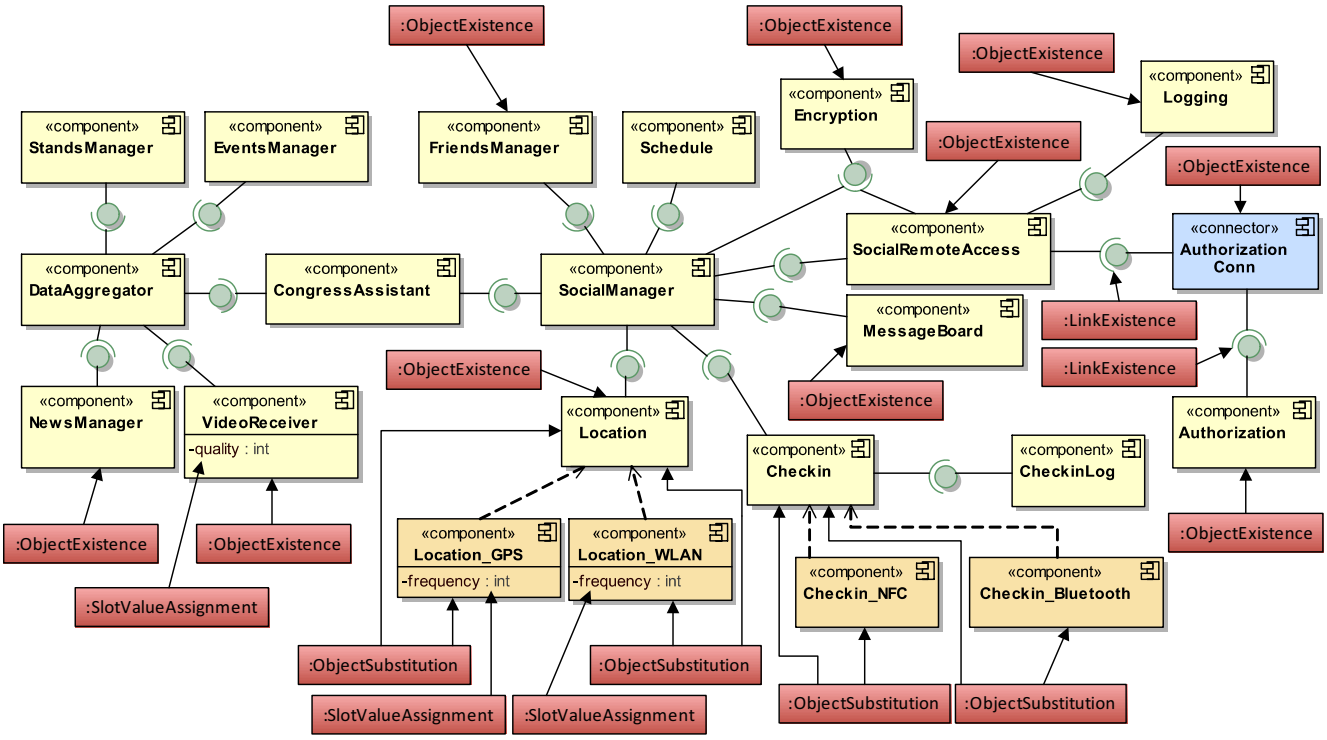

Fig. 3. Architectural Model of the case study

TABLE I
CVL VARIATION POINTS TAXONOMY SUMMARY

Variation point	Description
ObjectExistence	When it is applied to an element from the base model, indicates whether the element is included (positive application) or deleted (negative application). Therefore, it is used to define *optional* model elements.
ObjectSubstitution	Allows the substitution of an element of the base model for another element. It can be used for specifying alternative realizations of components of the base model.
SlotValueAssignment	Inserts a value into a slot of a base model element.
LinkExistence	Similar to ObjectExistence variation point, but applied to links among elements of the base model.

Our approach uses an AFM that contains a formal representation of the software architecture and its variation points. This AFM is one of the inputs of our GA. As discussed in Section VII, we are working on designing a mapping algorithm to go from the software architecture with variability specified in Figure 3 to the AFM. However, this AFM may also be manually specified. Since the reconfigurability approach presented in this paper is independent from the mechanism used to generate the AFM we omit these details in the paper.

Figure 4 shows an excerpt of the AFM that formalizes the variability of the case study presented in Section II-D. All the components are modelled as optional or mandatory children of the Component feature, and all the interconnections among them are modelled as children of the Connector feature. The variants of a component are also modelled as children of the component they realize, and grouped under an OR group or an XOR group (e.g. V_Location_GPS and V_Location_WLAN features). Finally, the variability of the component's parameters is also specified as part of the AFM. We can see, for instance, that it is mandatory to select the feature related to the quality parameter of the VideoReceiver component (P_VideoReceiver_quality) because, in other case, that parameter's value would be undefined. Furthermore, exactly one value for this parameter can be selected simultaneously, since the features are in an XOR group.

Moreover, there are constraints between features of the AFM whose relationship is not parent-child. Therefore, several cross-tree constraints are introduced in order to ensure that each valid configuration of the AFM corresponds to a valid and consistent system architecture. These constraints also allow the detection of some architectural inconsistencies that can be refined by the SA. For instance, Constraints 1-3 ensure that if the source or target components of the respective connectors are not selected, then the connector has to be excluded.

The information about the resource usage and utility is provided as a table in which each entry specifies the resource usage and utility of different features of the AFM. These features can be related to components, component variants or different values of parameters related to these components or variants. This information, together with the AFM, are the input for the GA which is executed by the DRS in order to find a configuration of the application that fits the current context. In this case, the resource we are restricting is the battery usage. Some of these values are shown in Table II.

IV. DYNAMIC RECONFIGURATION SERVICE

As previously described, the DRS is responsible for adapting the applications at runtime according to the current context,

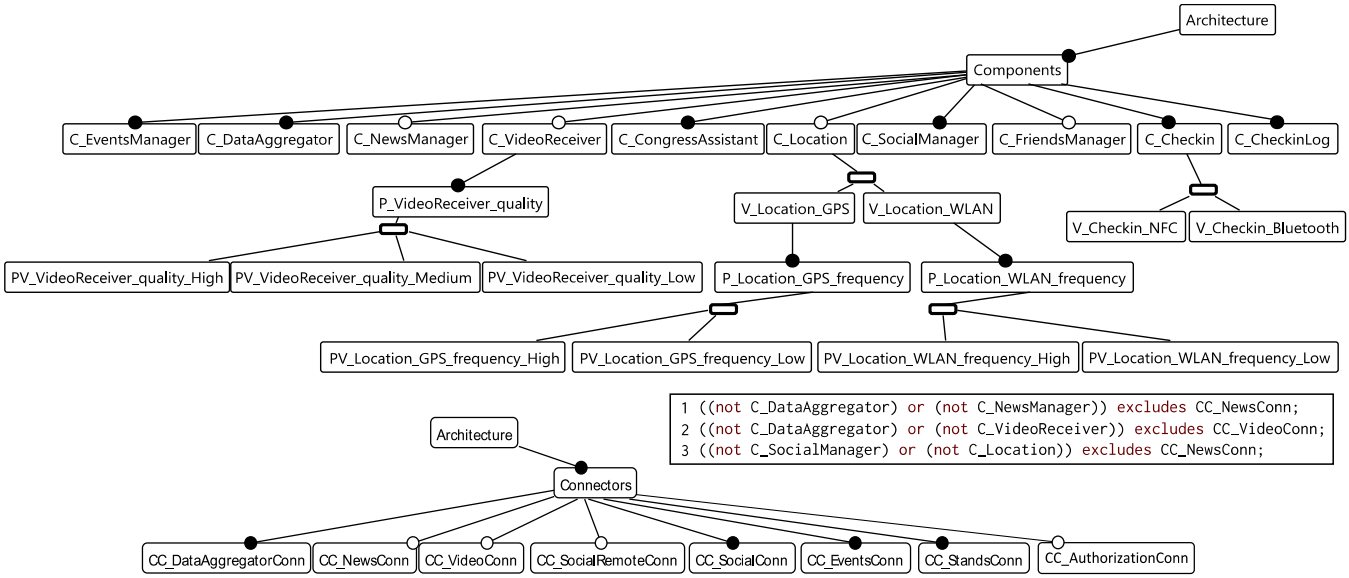

Fig. 4. Excerpt of the AFM, including cross-tree constraints

TABLE II
RESOURCE USAGE AND UTILITY INFORMATION TABLE

Element	Battery	Utility
Location_GPS	60	35
Location_WLAN	30	15
Checkin_Bluetooth	50	50
CheckinLog	15	30
Location_WLAN.frequency.High	10	7
Location_WLAN.frequency.Low	5	4

while the CMS provides the DRS with context information.

In this section we mainly focus on the plan stage of the MAPE-K loop (Plan Generator), which is part of the DRS and uses the AFM, the context information and the utility and resources information. As Brataas et al. show in [29], the reconfiguration time is divided in three different tasks: (1) analyse the context data; (2) plan (decide) the new configuration and (3) execute the plan in order to deploy the new configuration. They prove that the cost of the first and third tasks can be considered fixed, while it is critical to make the plan task as efficient as possible because it depends on the number of configuration variants. Therefore, the challenge is finding the set of features that defines the optimal configuration (the one that provides the highest utility while not exceeding the resources limitations) in a very efficient way. However, optimizing an FM with resource constraints is an NP-hard problem [17] and, therefore, it is impossible to use exact techniques to solve this optimization problem for our purpose. Concretely, as shown in [12], exact techniques can only be applied to small FMs at the cost of a very high execution time. Nevertheless, artificial intelligence algorithms can find nearly-optimal solutions in an efficient and scalable way. In this paper, we use the genetic algorithm of Guo et al. [12], which focus on optimizing FM configurations, for optimizing the AFM, since it has been proven to be efficient and produces nearly-optimal results. Concretely, this algorithm is able to generate FM configurations with about 90% of optimality, which means that the utility of the solutions obtained using this algorithm is approximately the 90% of the utility of the optimal configuration that would be obtained using an exact algorithm. Although the algorithm by Guo et al. is not focused on a DSPL approach, we show in this paper that their algorithm is applicable to the DSPL domain. Furthermore, thanks to the great improvement in the processing and memory capacities of smartphones, using artificial intelligence algorithms in mobile devices is feasible and efficient, as it is proven in this paper.

Therefore, the plan generator of the DRS relies on a genetic algorithm to decide which configuration should be deployed according to the current context. In genetic algorithms, solutions are modelled as chromosomes. In our case, a chromosome consists of a sequence of genes where each gene is a boolean value that indicates whether a concrete feature is selected or unselected. The steps taken during the execution of the algorithm are as follows:

1. *Population initialization*. A set of initial chromosomes (configurations) is randomly generated. Therefore, it is necessary to transform each one to get a valid solution from each randomly-generated one. The transformation process performs the necessary additions and exclusions of features from the randomly generated one, returning a chromosome which represents a valid configuration as a result which, in addition, does not exceed the available resources.

2. *Evolution through generations*. Once an initial population of valid configurations has been generated, the next step is evolving the population through generations in order to find better configurations, which provide a higher utility. In each generation, two chromosomes randomly chosen from the population are crossed. The resulting chromosome is transformed to get a a valid solution, and the worst chromosome of the population is replaced with the new one. This process is

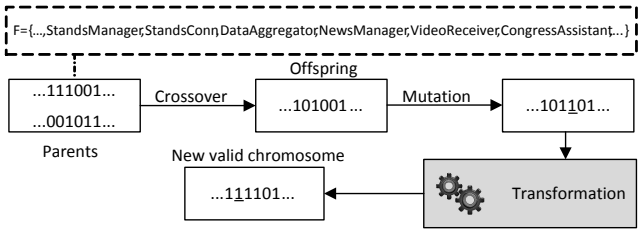

Fig. 5. Applying the genetic algorithm in the Dynamic Reconfiguration Service

repeated until a stopping condition is reached. For instance, the evolution can be stopped once a maximum number of generations is reached or when the population has not evolved after a certain number of consecutive generations. In our case, we use both conditions, stopping the evolution when the first one is reached.

3. *Return the best chromosome*. The best chromosome, which represents the configuration which provides the highest utility, is returned as the solution to the optimization problem.

In the rest of this section, this approach is applied to our case study, as illustrated by Figure 5. First, before the application is started, it is necessary to deploy the initial AFM configuration. An initial population of chromosomes that represent valid configurations and fit the resource constraints is generated. Since our AFM is composed of 81 features, each chromosome contains 81 genes but, due to the lack of space, we only show a reduced set (StandsManager, StandsConn, DataAggregator, NewsManager, VideoReceiver, CongressAssistant in Figure 5). Then, in every generation, two chromosomes are randomly selected for performing a *crossover*. A *crossover* between the two selected parents (111001 and 001011) is performed taking genes randomly from both parents, and the resulting *offspring* (101001) is *mutated* by changing the value of one of its genes (101**1**01). However, the *offspring* will probably be an invalid chromosome because it does not fit the constraints of the AFM. For instance, in our example, the *offspring* has the StandsManager component selected (i.e. the first bit is 1) while its connector, StandsConn, is not selected (i.e. the second bit is 0). Therefore, it is necessary to apply a *transformation* to the *offspring*, which adds the missing features. The transformation mechanism adds the missing features, and its output is a valid configuration where the StandsConn feature is also selected (1**1**1101). Then, this new chromosome replaces the chromosome with lowest value of the population, and this process is repeated until the stopping condition is reached.

Once the solution has been found, the DRS generates the reconfiguration plan from the differences between the previous configuration and the new one. Since each configuration is represented as a sequence of bits, the reconfiguration plan is obtained with a XOR operation between both configurations. In this way, the components that are no longer available in the new configuration are removed, the new components are instantiated and the parameters of the component are reconfigured. An important issue to highlight here is that the reconfiguration plans in our approach are generated in such a straightforward way because of the use of our AFM, where the features of the FM directly represent elements of the application software architecture. Otherwise, if our approach would use traditional FMs instead of AFM, there would be a gap between the output of the GA, where features would represent high-level concerns, and the elements in the software architecture that need to be added/removed. In other approaches this gap needs to be solved by defining a mapping between the elements of the FM and the elements of the software architecture. Moreover, since the mapping needs to be done at runtime the reconfiguration time will be penalized. In our approach there is also a mapping between the software architecture and the AFM, but it is done only once and, more importantly, off-line.

V. Evaluation

In this section we evaluate the ability of the optimization algorithm to find nearly-optimal configurations according to the available resources. Furthermore, since the resources of mobile devices are very limited, it is very important to verify the efficiency of the algorithm. Concretely, the time elapsed by the algorithm during the optimization process has been measured. To this end, the optimization algorithm has been applied to our case study using an ASUS Nexus 7 device running Android 4.2.1.

The case study has been previously mapped to an AFM which consists of 81 features (an excerpt can be seen in Figure 4), resulting in 2^{81} possible combinations, of which only 2400 represent valid configurations that satisfy all the constraints. This small ratio makes it more difficult to reach valid solutions using artificial intelligence algorithms, which makes it even more important to evaluate the effectiveness of the genetic algorithm applied in our approach. Figure 6 shows how these configurations are distributed according to their resource usage. Concretely, we can see that there is a peak in the distribution of configurations at around 500 units of resource usage. Therefore, we can expect a significant decrease in the execution time of the algorithm as the available resources increase and get closer to 500 units because it is increasingly easier to find a valid configuration. On the other hand, once the peak is exceeded, the number of new valid configurations decreases fast. Therefore, we can expect a nearly-constant execution time despite the increase in the available resources.

All the experiments have been repeated 100 times and the mean value and standard deviation (both for utility and time) has been calculated. The size of the population is 30, while the maximum number of generations for each repetition of the experiment is 20, stopping the algorithm if no better solutions are found after 3 consecutive generations. The algorithm execution is stopped after 60 unsuccesful retries for generating the initial population, although this point was not reached during the evaluation of our case study. These settings have been proven to provide good results, although an exhaustive optimization of them, which will be addressed in future work, has not been performed.

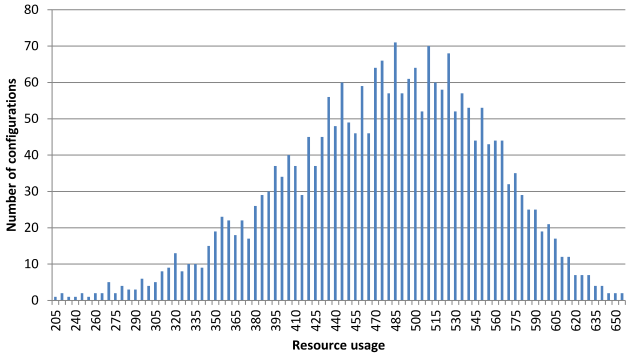

Fig. 6. Case Study AFM configurations distribution

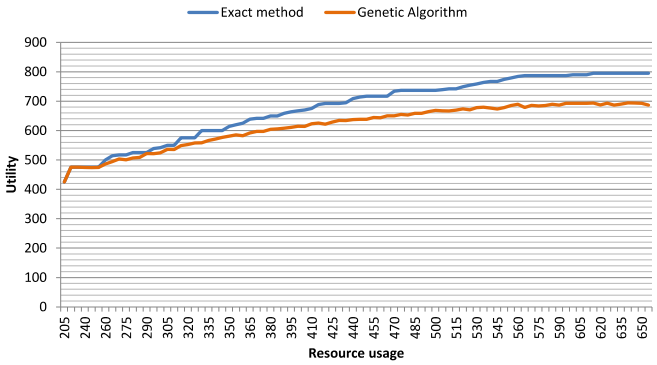

Fig. 7. Optimality Evaluation

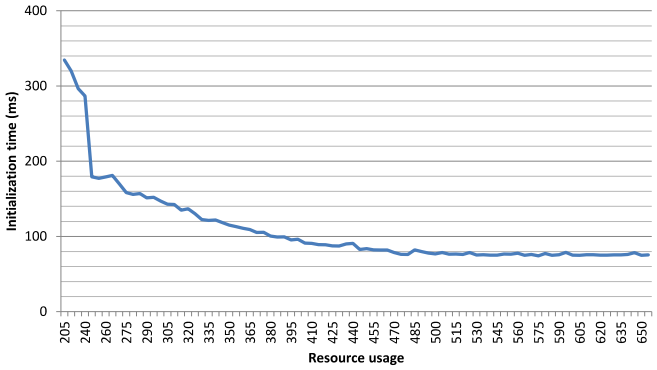

Fig. 8. Initialization Time Evaluation

For the evaluation of the effectiveness of the algorithm we have compared the solutions obtained using the genetic algorithm with the optimal solutions. In order to find the optimal solutions we have obtained a list of all the valid configurations of the AFM using FAMA Tool Suite [19], calculating then the resource usage and the utility of each one of them. This step (obtaining the optimal solutions) have been executed in a desktop computer since it is too expensive to be run in a mobile device.

The results are shown in Figure 7 and summarized in Table III. If we use the concept of optimality presented in [12], defined as the ratio between the utility of the solution obtained using the genetic algorithm and the one obtained using the exact method, the results show that the degree of optimality of the solutions obtained is always over 87%. The optimality slightly decreases as the available resources increase because there are much more valid configurations whose utility is much lower than the optimal one. However, even in the worst case the degree of optimality is very high, specially taking into account that the optimization problem is NP-hard.

On the other hand, we have evaluated the time elapsed in the execution of the algorithm. We distinguish between the initialization time, which is the time needed to generate the initial population, and the analysis time, elapsed iterating over the successive generations. The results for the initialization time are shown in Figure 8, and some of the measurements are detailed in Table III. As it is expected, when the restrictions are harder (less resources are available) it is more difficult to obtain valid solutions. Therefore, the time elapsed in the generation of the initial population is higher. In the worst case, the initialization time is 334.584 ms. However, as the available resources are higher, it becomes much easier to find valid solutions and the initialization time drops significantly, falling below 100 ms when the available resources are higher than 380 units. Further optimizations can be introduced in the algorithm in order to minimize the initialization time. For instance, those elements of the population that remain valid can be reused along different executions of the optimization algorithm. However, it has not been still evaluated and will be addressed in future work. Regarding the analysis time, we can see that it is very low compared with the initialization time. Although its value does not vary significantly with respect to the available resources, we can see that it increases slightly as the number of available resources increase. This behaviour can be explained because, when there are less available resources, the algorithm usually stops before reaching 20 generations because no better solutions are found.

Taking into account the results obtained, we consider that our approach is suitable for providing support for dynamic reconfiguration on mobiles devices, generating nearly-optimal configurations without introducing an excessive overhead.

VI. Related Work

In this section we discuss those approaches that are comparable to the work presented in this paper. On the one hand, our approach is driven by the MAPE-K loop on which AC rely, providing the applications for mobile devices with the ability to reconfigure their architecture in an autonomic and optimal way according to the available resources. We can find several approaches in the literature which also rely on the same principals. For instance, Gamez et al. [8] propose a reconfiguration mechanism that switches among different architectural configurations at run-time. The valid configurations are manually specified and represented using feature models, while the reconfiguration plans are automatically generated from the differences among them. Therefore, both are specified at design-time, which leads to the deployment of sub-optimal configurations at run-time. Trumler et al. [30] also propose an autonomic middleware, which is focused exclusively on

TABLE III
EVALUATION RESULTS SUMMARY

Resource limit	Optimal Utility	Obtained utility	Optimality	Initialization time	Analysis time
205	425	425 ($\sigma = 0$)	100%	334.584 ms ($\sigma = 55.207$ ms)	2.416 ms ($\sigma = 0.995$ ms)
255	475	474.62 ($\sigma = 1.886$)	99.92%	177.312 ms ($\sigma = 29.056$ ms)	3.224 ms ($\sigma = 2.697$ ms)
300	542	524.59 ($\sigma = 10.755$)	96.79%	147.137 ms ($\sigma = 22$ ms)	4.055 ms ($\sigma = 2.169$ ms)
350	614	580.9 ($\sigma = 17.514$)	94.61%	115.03 ms ($\sigma = 17.483$ ms)	4.321 ms ($\sigma = 2.419$ ms)
400	667	614.52 ($\sigma = 19.865$)	92.13%	96.291 ms ($\sigma = 11.877$ ms)	5.03 ms ($\sigma = 2.195$ ms)
450	717	641.635 ($\sigma = 20.333$)	89.49%	81.319 ms ($\sigma = 8.577$ ms)	6.055 ms ($\sigma = 3.738$ ms)
500	734	665.075 ($\sigma = 23.652$)	90.24%	76.067 ms ($\sigma = 7.128$ ms)	7.128 ms ($\sigma = 4.577$ ms)
550	774	680.445 ($\sigma = 27.414$)	87.91%	74.043 ms ($\sigma = 6.316$ ms)	8.013 ms ($\sigma = 5.52$ ms)
600	790	692.66 ($\sigma = 32.322$)	87.68%	75.165 ms ($\sigma = 9.921$ ms)	9.484 ms ($\sigma = 6.762$ ms)
655	795	691.904 ($\sigma = 32.656$)	87.03%	73.682 ms ($\sigma = 6.091$ ms)	8.83 ms ($\sigma = 6.381$ ms)

the Smart Doorplate Project and do not provide details about which different kinds of reconfigurations they support and how they are performed.

There are also many work that do not exactly follow the principals of AC but provide support for reconfiguration at the application level [3], [9], or also at the middleware layer [4], [5], [6], [7]. However, they are not usually available for evaluation or they are not runnable on resource-constrained devices. MUSIC [7] is a OSGi-based middleware for developing context-aware reconfigurable applications. It is a component based and service oriented approach which mainly consists of two different parts: the context and the adaptation middlewares. The adaptation middleware is responsible for adapting the applications, deploying the configuration that best fits the current context. This middleware is equivalent to our DRS and uses a model of the application, in which each configuration (known as a plan) consists of a set of component types and the connections between them. When context changes, a utility function is evaluated for each configuration, and the one with the highest utility is deployed. The main difference between MUSIC (as well as the other existing approaches) and our approach is that they require having available at runtime all the valid application configurations, while in our approach they are generated on demand using the optimization algorithm.

On the other hand, we use FMs to derive correct architectural configurations. We have found other work in which FMs are used beyond their usual purpose. For instance, in [31] FMs are introduced as an additional step in a system's architectural recovery process, by providing a mapping between source code and features. This approach differs from ours in that the origin of their mapping process is the source code, while in our case is a specification of the SA. In [13] FMs are used to reverse engineer the variability of an existing system by recovering an FM from the actual architecture, but with important differences to our work. Mainly, variability is not explicitly represented in their software architectures, and thus their main motivation is to define a process to be able to "capture" and model that variability using an FM.

Finally, we use an optimization algorithm to select a nearly-optimal configuration that satisfies the resource constraints and maximizes a utility function. In this sense, there are algorithms for feature selection optimization that take a FM as input and generate a product configuration by selecting a highly optimal set of features that adheres to a set of resource constraints. In [17], an FM is transformed into a Multi-dimensional Multiple-choice Knapsack Problem that allows nearly-optimal FM configurations in polynomial-time to be found. This is also the objective of [12], but using genetic algorithms, being even faster than the previous one. On the other hand, the proposal of Benavides et al. [32] always finds the optimal configuration using Constraint Satisfaction Problems with exponential-time complexity, making it unsuitable for runtime optimization.

The main difference with our approach is that all these algorithms have been used in static SPLs, while we use it in DSPLs. In a static SPL a product configuration is generated during the design time in order to deploy one particular product from the family of products. This means that the algorithm is applied only once at design time. We use the algorithm to implement a DSPL, meaning that the optimization algorithm is used at runtime by the DRS in order to adapt the product. The most similar approach to ours is the work presented in [33], where an optimization algorithm is also used to improve user interface adaptation at runtime. An important difference is that their work is specific to a user interface architectural model, while our approach is more general because it can be applied to the architectural model of any kind of applications They use a different optimization algorithm although, as in our case, their approach does not depend on a particular optimization algorithm and is designed to work with other algorithms. Finally, the average adaptation time of our approach is considerable lower than the one reported in [33].

VII. CONCLUSIONS AND ON-GOING WORK

In this paper we have presented a novel approach that provides support for the dynamic reconfiguration of mobile applications, optimizing the configuration of the system at runtime according to the available resources. In order to do that we model the variability of the application architectural model into an AFM, which is a variant of a FM where features are architectural elements. In this way, we take advantage of the tools and algorithms that are available for FMs in order to analyse architectural variability. Concretely, the use

of a genetic algorithm has been proposed that allows nearly-optimal configurations at runtime to be obtained using the AFM, the context information and the resource and utility information as input. In order to describe and evaluate our approach we have applied our approach to a case study. A set of experiments have been defined to evaluate the efficiency of the optimization algorithm applied to our case study in order to verify that it is suitable for resource-constrained devices. The results obtained show that it is efficient and can be used to provide dynamic reconfiguration in mobile devices without introducing an excessive overhead.

As part of our on-going work we are designing and implementing a mapping algorithm that will be able to generate the AFM directly from the models of the software architecture with variability. The use of this mapping algorithm will improve our proposal specially on two aspects: (1) SAs will focus on modelling the software architecture using architectural models, which are closer to their discipline. This means that they will never have to work directly with the AFM, and (2) It will be easier to have a complete and correct AFM. This is especially important for effectively representing all the cross-tree constraints between features, which is not a trivial task. Using our mapping algorithm most of the cross-tree constraints that need to be considered will be automatically generated.

ACKNOWLEDGEMENTS

This work is supported by Projects TIN2008-01942, P09-TIC-5231 and INTER-TRUST FP7-317731.

REFERENCES

[1] IBM, *Autonomic Computing White Paper — An Architectural Blueprint for Autonomic Computing*. IBM Corp., 2005.

[2] G. G. Pascual, L. Fuentes, and M. Pinto, "Aspect-oriented reconfigurable middleware for pervasive systems," in *Proceedings of the CAiSE Doctoral Consortium*, vol. 731. CEUR-WS, 2011.

[3] A. Chan and S. Chuang, "MobiPADS: a reflective middleware for context-aware mobile computing," *IEEE Transactions on Software Engineering*, pp. 1072–1085, 2003.

[4] T. Gu, H. Pung, and D. Zhang, "A service-oriented middleware for building context-aware services," *Journal of Network and Computer Applications*, vol. 28, no. 1, pp. 1–18, 2005.

[5] A. Janik and K. Zielinski, "AAOP-based dynamically reconfigurable monitoring system," *Information and Software Technology*, vol. 52, no. 4, pp. 380–396, 2010.

[6] N. Paspallis, "Middleware-based development of context-aware applications with reusable components," *University of Cyprus*, 2009.

[7] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, "Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments," *Software Engineering for Self-Adaptive Systems*, pp. 164–182, 2009.

[8] N. Gamez, L. Fuentes, and M. Aragüez, "Autonomic computing driven by feature models and architecture in famiware," *Software Architecture*, pp. 164–179, 2011.

[9] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[10] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93 –95, april 2008.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," DTIC Document, Tech. Rep., 1990.

[12] J. Guo, J. White, G. Wang, J. Li, and Y. Wang, "A genetic algorithm for optimized feature selection with resource constraints in software product lines," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2208 – 2221, 2011.

[13] M. Acher, A. Cleve, P. Collet, P. Merle, L. Duchien, and P. Lahire, "Reverse engineering architectural feature models," in *Software Architecture*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, vol. 6903, pp. 220–235.

[14] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J.-M. Jézéquel, "Weaving variability into domain metamodels," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 690–705.

[15] M. Acher, P. Collet, P. Lahire, and R. France, "Comparing approaches to implement feature model composition," in *Modelling Foundations and Applications*, ser. LNCS. Springer Berlin, 2010, vol. 6138, pp. 3–19.

[16] M. Matinlassi, "Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 127–136.

[17] J. W. et al., "Selecting highly optimal architectural feature sets with filtered cartesian flattening," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1268 – 1284, 2009.

[18] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[19] FaMa Tool Suite. [Online]. Available: http://www.isa.us.es/fama/

[20] Hydra project. [Online]. Available: http://caosd.lcc.uma.es/spl/hydra/

[21] S. P. L. O. T.: Software product lines online tools. [Online]. Available: http://www.splot-research.org/

[22] C. Kastner, T. Thum, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel, "FeatureIDE: A tool framework for feature-oriented software development," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. Ieee, 2009, pp. 611–614.

[23] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: http://doi.acm.org/10.1145/359576.359585

[24] Y. Choi, G. Shin, Y. Yang, and C. Park, "An approach to extension of uml 2.0 for representing variabilities," in *Computer and Information Science, 2005. Fourth Annual ACIS International Conference on*, 2005, pp. 258 – 261.

[25] E. A. Barbosa, T. Batista, A. Garcia, and E. Silva, "Pl-aspectualacme: an aspect-oriented architectural description language for software product lines," in *Proceedings of the 5th European conference on Software architecture*, ser. ECSA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 139–146.

[26] S. Robak, B. Franczyk, and K. Politowicz, "Extending the UML for modelling variability for system families," *International Journal of Applied Mathematics and Computer Science*, vol. 12, no. 2, pp. 285–298, 2002.

[27] T. Ziadi and J.-M. Jézéquel, "Product Line Engineering with the UML: Deriving Products," in *Software Product Lines*, K. Pohl, Ed. Springer Verlag, 2006, pp. 557–586.

[28] Common Variability Language (CVL). [Online]. Available: http://www.omgwiki.org/variability/doku.php

[29] G. Brataas, S. Hallsteinsen, R. Rouvoy, and F. Eliassen, "Scalability of Decision Models for Dynamic Product Lines," in *Proceedings of the International Workshop on Dynamic Software Product Line (DSPL)*, Sep. 2007, pp. 23–32.

[30] W. Trumler, J. Petzold, F. Bagci, and T. Ungerer, "Amun: an autonomic middleware for the smart doorplate project," *Personal Ubiquitous Comput.*, vol. 10, no. 1, pp. 7–11, Dec. 2005. [Online]. Available: http://dx.doi.org/10.1007/s00779-005-0029-4

[31] I. Pashov and M. Riebisch, "Using feature modeling for program comprehension and software architecture recovery," in *Engineering of Computer-Based Systems, 2004.*, may 2004, pp. 406 – 417.

[32] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Advanced Information Systems Engineering*. Springer, 2005, pp. 381–390.

[33] A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, and J.-M. Jézéquel, "Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation," in *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, Pise, Italy, Jun. 2011, pp. 85–94.