Run-time Support to Manage Architectural Variability Specified with CVL *

Gustavo G. Pascual, Mónica Pinto, and Lidia Fuentes

Departamento de Lenguajes y Ciencias de la Computación
University of Málaga, Málaga (SPAIN)
CAOSD group (http:/caosd.lcc.uma.es)
{gustavo,pinto,lff}@lcc.uma.es

Abstract. The execution context in which pervasive systems or mobile computing run changes continuously. Hence, applications for these systems should be adapted at run-time according to the current context. In order to implement a context-aware dynamic reconfiguration service, most approaches usually require to model at design-time both the list of all possible configurations and the plans to switch among them. In this paper we present an alternative approach for the automatic run-time generation of application configurations and the reconfiguration plans. The generated configurations are optimal regarding different criteria, such as functionality or resource consumption (e.g. battery or memory). This is achieved by: (1) modelling architectural variability at design-time using Common Variability Language (CVL), and (2) using a genetic algorithm that finds at run-time nearly-optimal configurations using the information provided by the variability model. We also specify a case study and we use it to evaluate our approach, showing that it is efficient and suitable for devices with scarce resources.

Keywords: Architectural Variability, CVL, Dynamic Reconfiguration, Genetic Algorithm, Context, Pervasive Systems

1 Introduction

Mobile applications demand runtime reconfiguration services that make it possible to adapt their behaviour to the continuous contextual changes that occur in their environment. One accepted approach to manage the runtime variability of applications is the Dynamic Software Product Line (DSPL) approach. DSPLs produce software capable of adapting to changes, by means of binding the variation points at runtime [15]. This means that the variants of the DSPL are generated at runtime.

Moreover, mobile applications run of lightweight devices with scarce resources (e.g. battery, memory, CPU, etc.), so they have the necessity of optimizing their functionality to the continuous resource variations, and also to the user

* Work supported by Projects TIN2008-01942, P09-TIC-5231 and INTER-TRUST FP7-317731.

needs. Ideally, such optimization should be managed autonomously by the application, which should be self-adapted. In this sense, it is widely accepted by the distributed systems community the use of the Autonomic Computing (AC) paradigm [16] to endow distributed systems with self-management capacities.

Combining the ideas of DSPL with AC, the development of a software system with self-adaptation capacities implies the following steps: (1) modelling as part of the software architecture (SA) the variation points that the designer foresees that may change at runtime; (2) the runtime environment needs to be monitored to listen for contextual changes that may affect the variation points; (3) when a contextual change occurs, the system must analyse how the change affects the variation points, and if a reconfiguration is needed; (4) if so, a plan defined as the set of changes that need to be performed in the current configuration over the set of variation points must be generated, ideally at runtime, and finally (5) the architectural variation points that are affected by the reconfiguration must be modified according to the plan generated in the previous step.

For the first step, a language to model the system variability is needed. Variability is modeled at different abstraction levels, mostly using feature models (FM) [10] at the requirements level and UML profiles or Architecture Description Languages (ADLs) [14, 2, 11] at the architectural level. In our approach, we model variability at the architectural level using the Common Variability Language [9](CVL). The reasons for choosing CVL are twofold. First, it is a MOF-based variability language and this means that any MOF-based application model can be easily extended with variability information using CVL; second, it has been submitted to the OMG for its standardization and it is expected to be accepted soon as the standard for modelling and resolving variability.

For the rest of steps, we follow the typical MAPE-K loop of the AC paradigm, where "MAPE"' stands for Monitoring-Analysis-Plan-Execution and 'K' stands for Knowledge. Existing approaches [7, 12, 17, 19, 20, 10, 8] mainly consists on doing at design time the analysis of the contextual changes and the generation of the reconfiguration plans to meet the new environmental conditions. Then, the set of valid configurations are pre-calculated, as well as the differences between pairs of configurations and the conditions to adapt the system from one configuration to another one, loading them into the device as part of the knowledge base. This is a shortcoming which limits the number of possible configurations and avoid generating the optimal ones. The alternative of using models@runtime approaches [21, 1] has also limitations in mobile environments since these approaches normally demand high computing resources. Thus, one of the contributions of our approach is the generation of the application configurations and the reconfiguration plans automatically at runtime.

Moreover, most DSPL approaches do not consider the optimization of the used resources at runtime. However, when the availability of certain resources decreases or increases significantly, the ideal situation would be to be able to decide which architectural configuration provides the best functionality, while not exceeding the available resources. Thus, fast algorithms to calculate the optimum configuration at runtime are desirable. Since this can be formulated as

an optimization problem, genetic algorithms (GAs) can be used to optimize the selection of architectural variation points that will conform to the new configuration. In this sense, a second contribution of our approach is the optimization of the used resources using genetic algorithms.

Specifically, our approach defines a *Context Monitoring* Service (CMS) for *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 [18], although in this paper we focus on presenting the details of how the DRS accomplishes the runtime reconfiguration of mobile applications. On the one hand, our DRS has the SA with variability specified using CVL available at runtime as part of the knowledge base, using it to perform reconfiguration. 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. For this we use a GA [13] which have already been used in static SPL – i.e. the optimization is performed at design-time. Since our DRS is installed inside a mobile device, we present some evaluation results showing that our approach is feasible and efficient for being executed with the fairly limited resources of a mobile device, resulting in good response times and nearly-optimal architectural configurations.

The rest of the paper is organized as follows. The motivation of our approach, the main contributions and the case study used throughout the paper are presented in Section 2. Then, the approach is further described in Section 3. Evaluation results are presented in Section 4, related work discussed in Section 5 and finally our conclusions and on-going work are described in Section 6.

2 Motivation and Approach Overview

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

2.1 Common Variability Language (CVL)

CVL is a domain-independent language for specifying and resolving variability that allows the specification of variability over any model which has been defined using a MOF-based metamodel. The approach proposed by CVL can be seen in Figure 1. The *base model* of an application does not contain any information about variability. Instead, the variability information is separately specified in a *variability model*, according to the CVL metamodel. One of the main advantages of CVL is that it is *executable*, meaning that it is possible to automatically generate *resolved models*. To this end, *resolution models* are specified to decide the choices in the variability models that are selected in order to automatically generate a fully specified product (i.e. without variability). In CVL, a variability model consists of three main parts:

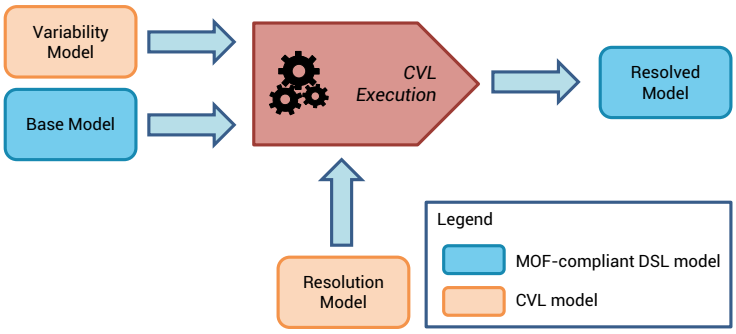


Fig. 1. CVL Approach

1. **Variation points**. Define the points of the base model that are variable and can be modified during the CVL execution. For instance, some of the variation points supported by CVL are the existence of elements of the base model or the links among them, or the assignment of an attribute's value.
2. **Variability Specification Tree (VSpec tree)**. Tree structures whose elements represent choices bound to variation points. These choices are resolved by a resolution model and propagated to variation points and the base model, generating the resolved model without variability. As it is explained in Section 2.4, VSpec trees show many commonalities with respect to FMs.
3. **OCL Constraints**. CVL supports the definition of OCL constraints among elements of a *VSpec tree*, providing a highly flexible mechanism for delimiting the bounds of variability, being able to discard invalid configurations.

2.2 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: Optimizing the architectural configuration. Mobile devices 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 an *utility function* that quantify the architectural variation points according to a criterion specified by the 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

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

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 [20] and [19].

Challenge 2: Generating the reconfiguration plan at runtime. 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 3).

Challenge 3: Executing the service in mobile environments. An important challenge of any service executing on a mobile environment is to reduce to the minimum 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 4 we demonstrate that our DRS is fast enough to avoid harming the user response time or the performance of the system.

2.3 Our Approach

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

Knowledge. As shown in Figure 2, in our approach the knowledge is represented by (1) the variation points; (2) the VSpecs tree; (3) the OCL constraints; (4) the software architecture; (5) the resource and utility information, and (6) the reconfiguration policy. The SA specifies the variability model in CVL, containing the variation points, the VSpecs tree and the OCL constraints, 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 1.*

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 if it changes 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 realization model

and the new one are calculated, generating a plan for switching between them (*Challenge 2*). As it has been explained in Section 2.2, 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.

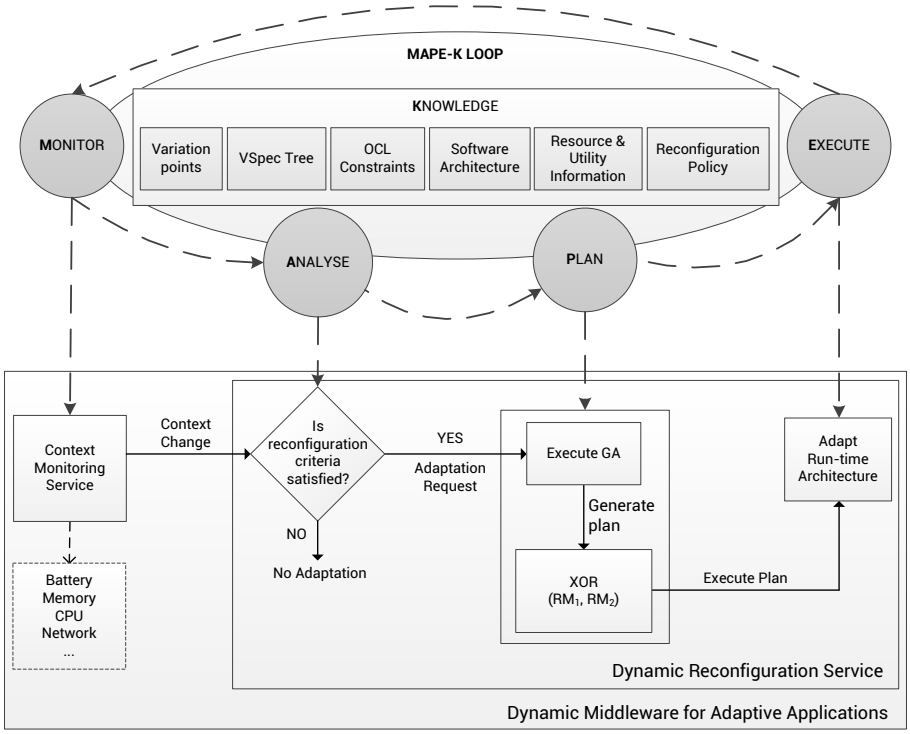


Fig. 2. Approach Overview

2.4 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. Access to information about the events, stands and 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. Access information about your friends: location, visited events and stands, agenda. Location is obtained using GPS or WLAN, and the measuring rate is variable (high, medium or low).
5. Exchange public messages or with your friends using a message board.

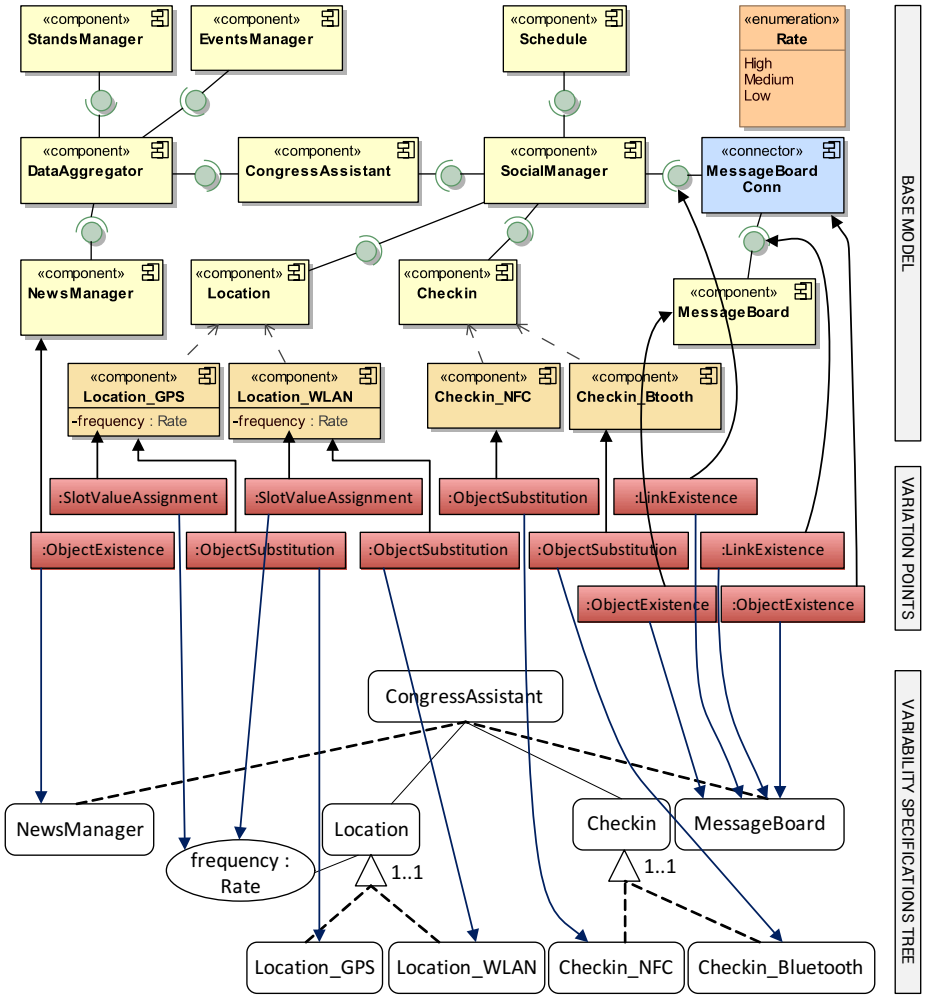


Fig. 3. Case study (Base model and variability model)

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 an excerpt of the 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 MessageBoardConn have been omitted from the figure for legibility reasons. The variability model is also shown in the Figure, including both the *variation points* and the *variability specifications tree*. 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_Btooth 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. To this end, the SlotValueAssignment variation point has been applied to the parameters of the components.

The architectural elements with an ObjectExistence variation point can be removed from the configuration. For instance, the Location component if the battery level is low. 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. We can see that a LinkExistence variation point has been associated to the links which connect the SocialManager and the MessageBoard components because they are removed in case the connector is deleted from the architectural model.

Each variations point has to be bound to a VSpec of the VSpec tree. We use two different kinds of VSpecs: *choices* and *variables*. Choices, which are shown in the figure as rectangles with rounded corners, are evaluated to *true* or *false*. On the other hand, *variables* can be evaluated to values of different types. For instance, if the VSpec *NewsManager* is decided false, the linked ObjectExistence variation point is disabled and the NewsManager component is removed from the architectural configuration. On the other hand, the value provided to the variable frequency is propagated to the frequency attribute of the Location_WLAN and Location_GPS components because they are bound to this variable through SlotValueAssignment variation points. An Vspec can be bound to its parent by a

solid or a dotted line. In the first case, it means that in case the parent has been decided true, a value has to be decided for that VSpec too. Then, a dotted line means that if the parent has been evaluated false, it is not necessary to decide a value for this VSpec. For instance, if the Location Vspec is decided false, it is not necessary to decide a value for Loaction_GPS or Location_WLAN.

The information about resource usage and utility is provided as a table in which each entry specifies the resource usage and the utility of different elements of the architectural model (components, variants or parameters). This information, together with the VSpec tree, 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 1.

Table 1. Resource usage and utility information table

Element	**Battery**	**Utility**
Location_GPS	60	35
Location_WLAN	30	15
Location_WLAN.frequency.High	15	9
Location_WLAN.frequency.Medium	10	7
Location_WLAN.frequency.Low	5	4

3 Dynamic Reconfiguration Service

As previously described, the DRS is responsible for adapting the applications at runtime according to the current context, 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 variability model, the context information and the utility and resources information.

As Brataas et al. show in [5], 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 choices for the VSpecs Tree (i.e. the resolution model) that defines the optimal configuration (the one that provides the highest utility while not exceeding the resources limitations) in a very efficient way. However, it is an NP-hard problem [22] and, therefore, it is impossible to use exact techniques to solve this optimization problem for our purpose. Concretely, as shown in [13], exact techniques can only be applied to small cases 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 a genetic algorithm based on the algorithm of Guo et al. [13], which focus on optimizing feature models configurations, for optimizing the Vspecs Tree, since it has been proven to be efficient and produces nearly-optimal results. Concretely, this algorithm

is able to generate 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. A chromosome consist of a sequence of genes, where each gene is a boolean value. In our case, VSpecs are mapped to genes in this way: (1) VSpec tree is traversed in a concrete order, which can be either breadth-first or depth-first; (2) each *choice* VSpec is modelled as a gen. In case the gen is evaluated as *true*, the VSpec is also decided true and (3) each *variable* VSpec is modelled as a set of genes. Concretely, a gen is added for each possible value of the VSpec. Only one of these genes can be evaluated as true simultaneously. Then, the gen whose value is true provides the value for the VSpec.

The steps taken during the execution of the algorithm are as follows:

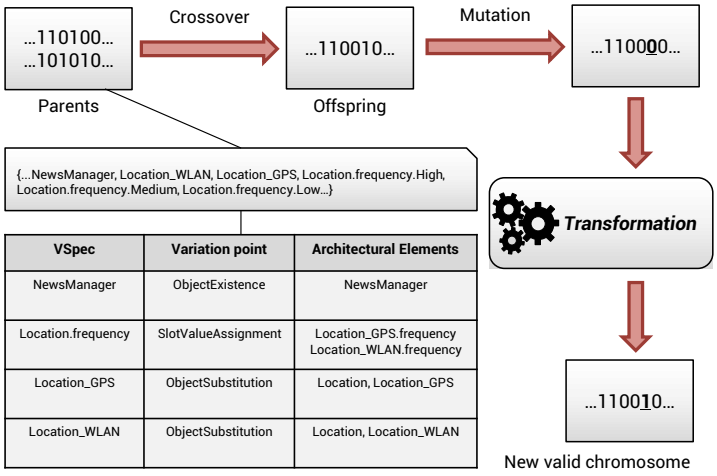
1. *Population initialization.* A set of initial chromosomes (configurations) is generated. They are generated randomly, and therefore it is necessary to transform each one to get a valid solution from each randomly-generated one. The transformation process performs the necessary additions and exclusions of Vspecs 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 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 4. First, before the application is started, it is necessary to deploy the initial configuration. An initial population of chromosomes that represent valid configurations and fit the resource constraints is generated. Our VSpec

Tree is mapped to a chromosome that contains 81 genes but, due to the lack of space, we only show a reduced set (NewsManager, Location_WLAN, Location_GPS, Location.frequency.High, Location.frequency.Medium, Location.frequency.Low in Figure 4). Then, in every generation, two chromosomes are randomly selected for performing a *crossover*. A *crossover* between the two selected parents (...110100... and ...101010...) is performed taking genes randomly from both parents, and the resulting *offspring* (...110010...) is *mutated* by changing the value of one of its genes (...111010...). However, the *offspring* will probably be an invalid chromosome because it does not fit the constraints of the VSpec Tree. For instance, in our example, the *offspring* has the Location_GPS component selected (i.e. its bit is 1), but no location frequency is specified. Therefore, it is necessary to apply a *transformation* to the *offspring*, which adds all the missing decisions. The transformation mechanism adds them, and its output is a valid configuration where, in this case, the Location.frequency VSpec is set to *medium* (...110010...). Then, this new chromosome replaces the chromosome with lowest value of the population, and this process is repeated until the stopping condition is reached.



VSpec	Variation point	Architectural Elements
NewsManager	ObjectExistence	NewsManager
Location.frequency	SlotValueAssignment	Location_GPS.frequency Location_WLAN.frequency
Location_GPS	ObjectSubstitution	Location, Location_GPS
Location_WLAN	ObjectSubstitution	Location, Location_WLAN

Fig. 4. Applying the genetic algorithm in the Dynamic Reconfiguration Service

4 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 VSpec tree defined for the variability specification of our case study contain 2400 valid configurations that fulfill all the constraints. Figure 5 shows

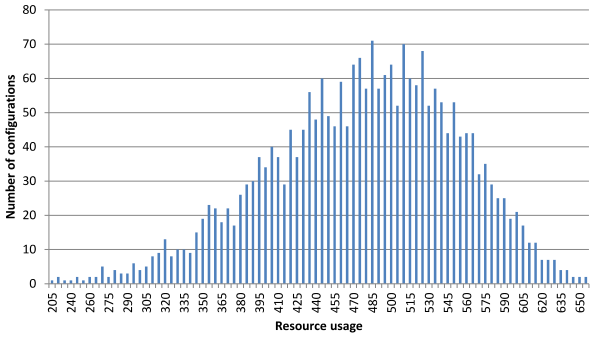


Fig. 5. Case Study configurations distribution

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. 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. 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 generated a list of all the valid configurations, 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.

Results are shown in Figure 6 and summarized in Table 2. If we use the concept of optimality presented in [13], which can be 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 Table 2. 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.

According to 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.

Table 2. Evaluation Results Summary

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

5 Related Work

In this section we discuss those approaches 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. [10] propose a reconfiguration mechanism that switches among different architectural configurations at run-time. The valid configurations are manually specified and represented using

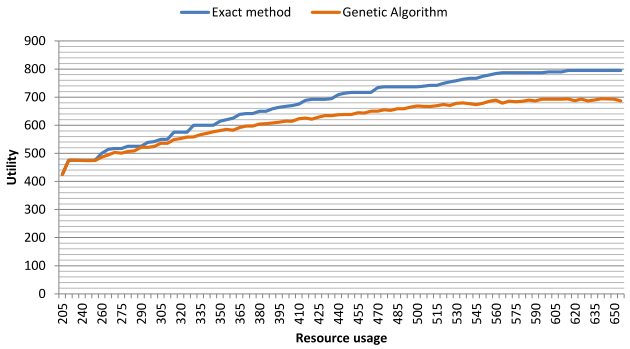


Fig. 6. Optimality Evaluation

FMs, 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.

There are also many work that do not exactly follow the principals of AC but provide support for reconfiguration at the application level [7, 8], or also at the middleware layer [12, 17, 19, 20]. However, they are not usually available for evaluation or they are not runnable on mobile devices. MUSIC [20] is an OSGi-based middleware for developing context-aware adaptive 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. 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 configurations of an application, while in our approach this configuration is generated on demand using the optimization algorithm.

Other work use CVL to manage variability and provide reconfiguration support. For instance, Ayora et al. [1] propose a mechanism for managing variability in business processes. At design time, variability is modelled using CVL. Then, process variants are adapted following a *models@runtime* approach, which is not suitable for devices with scarce resources. In [6], Cetina et al. also model variability using CVL, applying it to smart-homes environments. Concretely, several mechanisms for applying the necessary model transformations are evaluated. However, as in the previous approach, it is not applicable to mobile devices.

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 similar algorithms that allow the automatic generation of a *resolution model* according to different criteria. However, they are applied to (1) variability modelling techniques different than CVL VSpec trees, such as FMs, and (2) to static SPLs. In [22], 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 [13], but using

genetic algorithms, being even faster than the previous one. On the other hand, the proposal of Benavides et al. [3] 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 [4], 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 [4].

6 Conclusions

In this paper we have presented a novel approach that provides support for the dynamic reconfiguration of mobile applications, optimizing the system configuration according to the available resources. In order to do that we model the variability of the application architectural model using CVL. In this way, we take advantage of available algorithms to optimize the variability resolution. Concretely, the use of a GA has been proposed to obtain nearly-optimal configurations at runtime using the VSpec tree, the context information and the resource and utility information as input. In order to describe and evaluate our approach we have applied it 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.

References

1. Ayora, C., Torres, V., Pelechano, V., Alférez, G.H.: Applying CVL to business process variability management. In: Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone. pp. 26–31. VARY '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2425415.2425421
2. Barbosa, E.A., Batista, T., Garcia, A., Silva, E.: Pl-aspectualacme: an aspect-oriented architectural description language for software product lines. In: Proceedings of the 5th European conference on Software architecture. pp. 139–146. ECSA'11, Springer-Verlag, Berlin, Heidelberg (2011)

3. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: Advanced Information Systems Engineering. pp. 381–390. Springer (2005)
4. Blouin, A., et al: Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation. In: ACM SIGCHI Symposium on Engineering Interactive Computing Systems. pp. 85–94. Pise, Italy (Jun 2011)
5. Brataas, G., et al: Scalability of decision models for dynamic product lines (2007)
6. Cetina, C., Haugen, O., Zhang, X., Fleurey, F., Pelechano, V.: Strategies for variability transformation at run-time. In: Proceedings of the 13th International SPLC. pp. 61–70. SPLC '09, Carnegie Mellon University, Pittsburgh, PA, USA (2009), http://dl.acm.org/citation.cfm?id=1753235.1753245
7. Chan, A., et al: MobiPADS: a reflective middleware for context-aware mobile computing. IEEE Transactions on Software Engineering pp. 1072–1085 (2003)
8. Cuervo, E., Balasubramanian, A., Cho, D., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: Maui: Making smartphones last longer with code offload. In: Proceedings of the 8th international conference on Mobile systems, applications, and services. pp. 49–62. ACM (2010)
9. CVL: Common Variability Language. http://www.omgwiki.org/variability/
10. Gamez, N., Fuentes, L., Aragüez, M.: Autonomic computing driven by feature models and architecture in famiware. Software Architecture pp. 164–179 (2011)
11. Gomaa, H.: Designing software product lines with uml 2.0: From use cases to pattern-based software architectures. Reuse of Off-the-Shelf Components pp. 440–440 (2006)
12. Gu, T., et al: A service-oriented middleware for building context-aware services. Journal of Network and Computer Applications 28(1), 1–18 (2005)
13. Guo, J., White, J., Wang, G., Li, J., Wang, Y.: A genetic algorithm for optimized feature selection with resource constraints in software product lines. Journal of Systems and Software 84(12), 2208 – 2221 (2011)
14. Haber, A., Kutz, T., Rendel, H., Rumpe, B., Schaefer, I.: Delta-oriented architectural variability using monticore. In: Proceedings of the 5th European Conference on Software Architecture: Companion Volume. pp. 6:1–6:10. ECSA '11, ACM, New York, NY, USA (2011)
15. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41(4), 93 –95 (april 2008)
16. IBM: Autonomic Computing White Paper — An Architectural Blueprint for Autonomic Computing. IBM Corp. (2005)
17. Janik, A., Zielinski, K.: AAOP-based dynamically reconfigurable monitoring system. Information and Software Technology 52(4), 380–396 (2010)
18. Pascual, G.: Aspect-oriented reconfigurable middleware for pervasive systems. In: Proceedings of the CAiSE Doctoral Consortium. vol. 731. CEUR-WS (2011)
19. Paspallis, N.: Middleware-based development of context-aware applications with reusable components. University of Cyprus (2009)
20. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. Software Engineering for Self-Adaptive Systems pp. 164–182 (2009)
21. Welsh, K., Bencomo, N.: Run-time model evaluation for requirements model-driven self-adaptation. In: Requirements Engineering Conference (RE), 2012 20th IEEE International. pp. 329–330. IEEE (2012)
22. White, J., et al: Selecting highly optimal architectural feature sets with filtered cartesian flattening. Journal of Systems and Software 82(8), 1268 – 1284 (2009)