Future Generation Computer Systems 47 (2015) 127–144

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Self-adaptation of mobile systems driven by the Common Variability Language

Gustavo G. Pascual *, Mónica Pinto, Lidia Fuentes

Universidad de Málaga, Andalucía Tech, Spain

HIGHLIGHTS

- We specify an approach for the dynamic reconfiguration of mobile applications.
- We model a mobile application with variability which can be reconfigured at runtime.
- We simulate the execution of the mobile application when our dynamic reconfiguration service is applied and not applied, respectively.
- We measure the battery life as well as the overall utility of the application perceived by the user.
- Applying our dynamic reconfiguration, the battery life is incremented by 45.9% and the utility is incremented by 10.31%.

ARTICLE INFO

Article history:
Received 20 November 2013
Received in revised form
12 June 2014
Accepted 26 August 2014
Available online 6 September 2014

Keywords:
Architectural variability
CVL
Dynamic reconfiguration
Genetic algorithm
Context
Pervasive systems

ABSTRACT

The execution context in which pervasive systems or mobile computing run changes continually. Hence, applications for these systems require support for self-adaptation to the continual context changes. Most of the approaches for self-adaptive systems implement a reconfiguration service that receives as input the list of all possible configurations and the plans to switch between them. In this paper we present an alternative approach for the automatic generation of application configurations and the reconfiguration plans at runtime. With our approach, the generated configurations are optimal as regards 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 the Common Variability Language (CVL), and (2) using a genetic algorithm that finds nearly-optimal configurations at run-time 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.

1. Introduction

Mobile applications demand runtime reconfiguration services that make it possible for them to self-adapt their behaviour to the continual contextual changes that occur in their environment. Such reconfiguration services have to deal with the high variability of possible configurations that fit the different dynamic contexts. 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 [1]. This means that we have to model the elements that could be adapted dynamically as *dynamic variation points* and generate, at runtime, the different variants of the DSPL.

On the other hand, mobile applications run on lightweight devices with scarce resources (e.g. battery, memory, CPU, etc.), so they have to adapt their functionality to the continual resource variations, and also to the user's needs. Ideally, such optimization should be managed autonomously by the application itself, which should be able to self-optimize its functioning. For this purpose, widely accepted by the distributed systems community, is the use of the Autonomic Computing (AC) paradigm [2] to endow distributed systems with self-management capacities, such as self-adaptation and self-optimizing.

Combining the ideas of DSPL with AC, the development of a software system with self-adaptation capacities implies the following steps: (1) the variation points that the designer foresees that may change at runtime (i.e. the *dynamic variation points*) have to be modelled as part of the software architecture (SA); (2) the runtime environment needs to be monitored to listen for contextual changes that may affect the dynamic variation points; (3) when a contextual change occurs, the system must analyse the relationships between this change and the dynamic variation points, and

* Corresponding author. Tel.: +34 952132846.
E-mail addresses: gustavo@lcc.uma.es (G.G. Pascual), pinto@lcc.uma.es (M. Pinto), lff@lcc.uma.es (L. Fuentes).

http://dx.doi.org/10.1016/j.future.2014.08.015

whether or not 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 dynamic 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 modelled at different abstraction levels, mostly using feature models (FM) [3] at the requirements level and UML profiles or Architecture Description Languages (ADLs) [4–6] at the architectural level. In our approach, we model variability at the architectural level using the Common Variability Language [7] (CVL). The reason for choosing CVL is twofold. First, it is an 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 widely known MAPE-K loop [8] of the AC paradigm, where 'MAPE' stands for Monitoring, Analysis, Plan and Execution and 'K' stands for Knowledge. Existing approaches [3,9–14] mainly consist of analysing, at design time, the contextual changes and the generation of the reconfiguration plans to fit 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. All this previously calculated information is loaded into the device as part of the knowledge base of the MAPE-K loop. This is a shortcoming which limits the number of possible configurations and prevents the generation of the optimal ones. Other existing approaches that generate the configurations at runtime [15–20] also have limitations in mobile environments as usually most of them 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 and efficiently, so that it can be used in devices with few resources.

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 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 [21], 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 called DAGAME [22], optimized to be executed at runtime with scarce resources. As our DRS is installed inside a mobile device, we present some evaluation results showing that our approach is feasible and efficient enough to be 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 backgrounds to CVL and genetic algorithms are presented in Section 2. After this, the motivation of our approach, the main contributions and the case study used throughout the paper are presented in Section 3. Then, the approach is described further in Section 4. Evaluation results are presented in Section 5, related work is discussed in Section 6 and finally our conclusions and on-going work are described in Section 7.

2. Background

In this section we provide a background to DSPLs. Furthermore, we show the basics of CVL and genetic algorithms, which are used in our approach to model the architectural variability and generate the reconfiguration plan, respectively.

2.1. Dynamic software product lines

An SPL is "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way".[1] DSPLs move existing SPL engineering processes to runtime, ensuring that system adaptations lead the system to a valid state. Therefore, while in SPLs the engineering processes generate several systems of the same family at design time, a DSPL is a single system which is able to adapt its behaviour at runtime.

Variability modelling, which consists in specifying the commonalities and variabilities, is the central activity of both SPLs and DSPLs. The engineering processes of SPLs generate products by selecting specific values for the variable characteristics specified in the variability model. Therefore, the SPL engineer binds the variation points at design time considering the requirements of the intended product. In contrast, in DSPLs the variability model describes the potential range of variations that can be produced at runtime for a single product, i.e. *the dynamic variation points*, which must refer to the system architectural components. Therefore, in DSPLs the system architecture supports all possible adaptations defined by the set of dynamic variation points [1].

Then, as part of a DSPL definition the engineer must define:

1. The range of potential adaptations supported by the system in terms of architectural components.
2. An explicit representation of the valid configuration space of the system.
3. The context changes that may trigger an adaptation, i.e. the criteria to initiate a reconfiguration or decision making process.
4. The set of possible reactions to context changes that should be supported the system.

However, the way these aspects are implemented may differ greatly, as will be shown in Section 6.

As for the majority of DSPLs the decision to initiate a reconfiguration is made autonomously by the system (not by a human), they are considered a good technology for developing self-adapting systems such as mobile applications. In this context, most DSPL approaches share some common properties with the Autonomic Computing paradigm [2] (AC) such as the monitoring of the environment and the generation of successive configurations.

[1] http://www.sei.cmu.edu/productlines/.

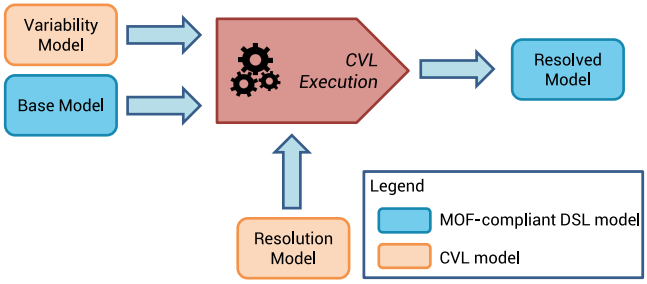

Fig. 1. CVL approach.

2.2. Common Variability Language (CVL)

CVL is a domain-independent language for both specifying and resolving variability. Its prime advantage is that it allows the specification of variability over any model which has been defined using a MOF-based metamodel. An overview of the approach proposed by CVL can be seen in Fig. 1. By one side, the software architect specifies the *base model* of the application, which does not contain any information about variability. On the other side, the variability information is separately specified in a *variability model*, according to the CVL metamodel. In order to generate the configuration of a specific product, the SPL engineer selects a set of options in the variability model. This set of options makes it possible to bind the variation points to concrete values, and this is what is called a *resolution model* of the variability in CVL. CVL is *executable*, meaning that it is possible to automatically generate *resolved models*, which are full product models (i.e. without variability). The advantage of CVL is that these resolved models are fully specified in the base language, making it possible for them to be processed with regular base language tools. So, it is easier to adopt than other SPL approaches, since the software architect does not have to change either the architectural language, or the design tool that is normally used.

In CVL, a variability model consists of three main parts:

1. **Variation points**. Define the points of the base model that are variable and can be modified during the execution of CVL. For instance, some of the variation points supported by CVL are the *existence* of elements of the base model or the links between them, or the *value assignment* of an attribute.
2. **Variability Specification tree (VSpec tree)**. Tree structures the elements of which (i.e. VSpec) are similar to features in feature modelling, representing choices bound to variation points. There are four types of VSpec: (1) *Choice* requires a binary decision (i.e. yes/no); (2) *variable* allows providing a value of a certain type; (3) *VClassifier* is a VSpec that can be instantiated zero or more times, and that generates a sub-tree for each instance. Each VClassifier has an instance multiplicity which indicates how many instances of it may be created (it is similar to the cardinality-based feature models); and (4) *composite VSpecs* (CVSpec) are used for modularity purposes. They are VSpecs that encapsulate other VSpec trees. A VSpec specification is resolved by a resolution model and propagated to the variation points and to the base model, generating the resolved model without variability. As it is explained in Section 3.3, VSpec trees present many commonalities with respect to FMs.
3. **OCL constraints**. CVL supports the definition of OCL constraints between elements of a *VSpec tree*, providing a highly flexible mechanism for delimiting the bounds of variability. These constraints are primarily used to discard invalid configurations.

In this paper we use CVL to specify the dynamic variation points of mobile applications.

2.3. Genetic algorithms

GAs are a search heuristic, inspired by the process of evolution, which are typically used to find solutions for optimization problems. Using GAs it is possible to find nearly-optimal solutions for optimization problems without having to explore the whole solutions space. As stated by Guo et al. [23], applying GAs can be highly appropriate when the solutions space is very wide and it is not affordable to evaluate all of them due to a lack of resources and time.

In genetic algorithms, candidates to be returned as the solution to the optimization problem are known as *chromosomes*, making up a *population*. Chromosomes are typically modelled as a list of binary variables known as *genes*, each one modelling a property of the solution, although different encodings can be defined. The objective function of the optimization problem is typically used as a *fitness* function for measuring the quality of each solution.

Three different stages can be identified during the execution of genetic algorithms:

Generation of the initial population. An initial set of solutions are generated to fill the population. The size of the population is a configurable parameter, and choosing the most appropriate size depends on the optimization problem that we are trying to solve.

Evolution through generations. The initial population generated in the previous step is evolved in order to find better solutions with a higher fitness value. Concretely, in each generation, two chromosomes are chosen (typically the two best chromosomes). A *crossover* operation is performed between them, obtaining a new solution that takes genes from both chromosomes. Then, a *mutation* may be introduced in the resulting chromosome changing the value of one or more of its genes. Mutations are useful for improving the diversity of the population, but a very high mutation probability can take the optimization problem closer to a random-based search, which leads to a loss in population fitness. At the end of each generation, the new chromosome replaces the worst one in the population, thereby improving the overall fitness of the population.

Returning the solution. After the last generation, the solution with the highest fitness value is returned as the solution to the optimization problem. Different criteria can be defined for stopping the evolution process. For instance, a maximum number of generations can be specified as a configurable parameter. Moreover, it is also possible to stop the evolution if the algorithm is unable to improve the population fitness during a concrete number of consecutive generations.

As part of our approach we defined DAGAME, a GA that automatically generates, at run-time, a quasi-optimal application configuration that fits the current resources of the mobile application.

3. Motivation and approach overview

In this section we present the motivation of our work. We discuss the challenges that have to be overcome for the automatic generation of quasi-optimal application configurations and reconfiguration plans at run-time, for mobile applications. An overview of our approach and a case study are also presented.

3.1. Challenges

From the literature it is possible to identify the differences between existing DSPL approaches. The main ones are [24]: (1) how they model the dynamic variation points; (2) whether the successive configurations generated by the approach are optimal regarding some criteria or not; (3) if the reconfiguration plan is generated at design or runtime; (4) the decision making process

used to trigger a reconfiguration, and (5) if they can be used to develop applications for resource constraint devices, or not. According to this, we have identified the following challenges that we have to face in the building of a *Dynamic Reconfiguration Service* (DRS) service (the core part of our approach). The goal of this DRS service is to react to the run-time contextual changes by optimizing the configurations according to the availability of certain resources (e.g. battery, memory, CPU).

Challenge 1: *Modelling dynamic variation points*. The first task of a DSPL approach is to appropriately model the dynamic variation points, that is, the elements that could be adapted dynamically. But, these dynamic variation points must be available at runtime, in order to generate the different variants of the DSPL. So, once the variation points have been specified in a variability language (e.g. Feature Models), the challenge is how they can be made available at run-time in order to generate the successive runtime configurations. Most DSPLs approaches apply model driven development technologies, which generate runtime models or code from a variability model, for instance, a feature model. We propose an architecture-centric approach in which the dynamic variation points instead of being defined in terms of the application features (as usual in feature models), they are defined in terms of the application architecture. So, at run-time we generate the successive software architecture configurations by binding the architectural variation points specified in a variability language [25]. Regarding the variability language used, we propose to use CVL as it eases the generation of the architectural variation points compared with, for instance, feature models [25]. In CVL the VSpecs tree is linked with the base model (i.e. the software architecture model) so, the architectural variation points are obtained almost directly, as we will show later. Also, the choice of CVL has other advantages, already given in the introduction, such as it being a proposed standard, and that it facilitates the adoption of our approach since it does not impose the use of a new language to describe the application architecture.

Challenge 2: *Optimizing the architectural configuration*. Any DSPL approach ensures that the successive configurations that are instantiated at runtime are valid regarding the variability model. But, sometimes this is not enough, in addition, the DSPL process must also ensure that runtime configurations are also optimal in regard to some specific criteria (e.g. user preferences, quality of service, amount of resources, etc.). In our case, the primary aim of our research is to provide dynamic adaptability of applications running on mobile devices, with resource constraints. So, we need to consider not only the valid, but also the optimal architectural configurations regarding the usage of the device's resources (e.g. battery, memory, etc.). So, our main criteria to initiate a reconfiguration will be the optimization of the usage of the available resources. In this paper we use an optimization algorithm (called DAGAME) that is able to find a nearly-optimal configuration taking into account the resource usage of the valid architectural configurations. Note, that an exact algorithm cannot be used because the problem to be solved has been proven to be NP-hard (non-deterministic polynomial-time hard) [26].

Challenge 3: *Generating the reconfiguration plan at runtime*. Most DSPL approaches generate, at design time, the configurations that will be deployed at runtime [13,27–29]. However, the potential number of configurations normally grows exponentially with the number of dynamic variation points. In order to cope with this serious problem, some approaches consider only a subset of the valid configurations at runtime (e.g. the most probable ones), which are pre-loaded in the system. However, this is an important drawback, especially in our case. It would be very difficult to ensure at design time, that the list of loaded configurations includes the optimal ones according to the resources that are available at any point of the application's execution. So, it is preferable to automatically generate all the potential configurations at runtime, there by making it possible to choose the optimal one taking into account a given context change. Concretely, in our approach the different architectural configurations are generated on demand using the DAGAME optimization algorithm that is loaded in the mobile device. The reconfiguration plan is easily calculated as the difference between the running and the new configuration generated by the optimization algorithm, as we will show in Section 4.

Challenge 4: *A scalable decision making process*. Those DSPL approaches that perform the analysis and derivation of reconfiguration plans at design time are usually based on the definition of a set of event–condition–action (ECA) rules [29,20]. An ECA rule includes the event that will trigger a reconfiguration, a condition about the system state that must be evaluated as true, and the reconfiguration plan or actions that have to be executed. The main problem with this approach is that the number of rules could become untreatable, especially if the number of potential configurations is very high. Goal-based approaches overcome this problem since they do not need to enumerate all the "context change–product configuration" pairs at design time, but at a cost of more runtime overhead. In our approach, we use the algorithm DAGAME, which optimizes a *utility function* that quantifies the quality of the generated product configurations. Although our approach is independent of the chosen utility function, the notion of utility typically refers to the expected user's overall satisfaction. For instance, the criterion used to determine the utility of a component could be the *precision* and the *measuring rate* in the case of a component that provides 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 [13,12].

Challenge 5: *Executing the service with scarce resources*. An important challenge of any service executing in a mobile environment is to reduce by as much as possible 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 the extra time employed for the reconfiguration process being noted. In this regard, in Section 5 we demonstrate that our DRS is fast enough to avoid harming the user response time or the performance of the system.

3.2. Our approach

All these challenges have been addressed in our approach, summarized in Fig. 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 Fig. 2, in our approach the knowledge is represented by (1) the dynamic variation points; (2) the VSpecs tree; (3) the OCL constraints; (4) the software architecture (i.e. the base model); (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, addressing *Challenge* 1. Also defined as part of the SA is an estimation of the resource usage and the utility provided by the components of the architecture. This information provides an optimization criterion (addresses *Challenge* 2) for runtime reconfiguration and, therefore, using it we can generate different configurations at runtime (addresses *Challenge* 3) which maximize the utility of the application without exceeding the availability of a given resource.

Monitor. The CMS provides the DRS with information about the evolution of the availability of a certain resource, such as the

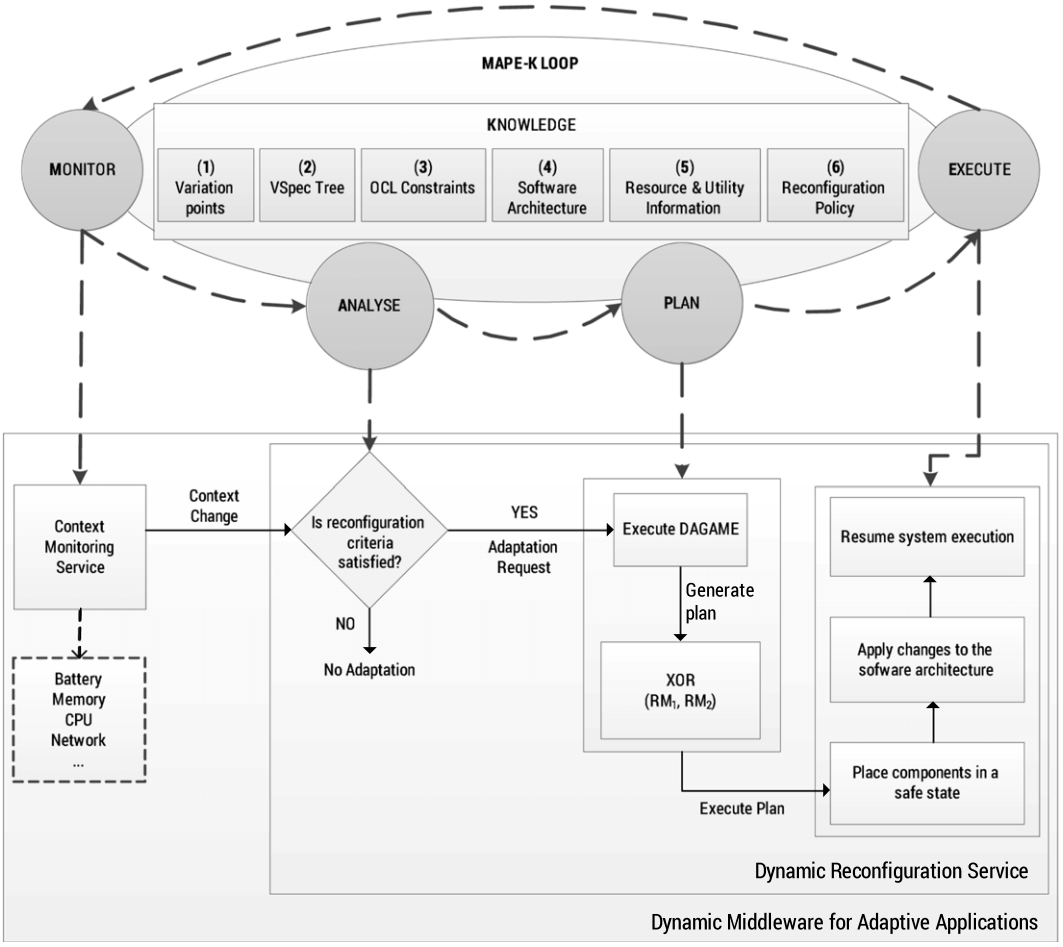

Fig. 2. Approach overview.

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 whether 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 by more than 5% since the last measurement was taken, or if it changes by more than 10% per hour. Therefore, several reconfiguration policies can be defined, and the policy applied is part of the *Knowledge* base (addresses *Challenge* 2).

Plan. In the case that the analyser decides that the application needs to be adapted, the DAGAME algorithm is executed in order to find a nearly-optimal configuration according to the current context (addresses *Challenge* 4). Then, the differences between the current realization model and the new one are calculated, generating a plan for switching between them. Calculating the difference between two configurations is quite straightforward since it is directly obtained by performing an XOR operation between both configurations. Since the resolution models in CVL representing the architectural configurations are encoded as a sequence of ones and zeros, the generation of the software architectural model is very efficient, addressing *Challenge* 5.

Execute. Finally, the reconfiguration plan is executed in order to adapt the running architecture of the application, which implies removing the components that are no longer needed, adding the new components, connecting them and also reconfiguring the modified parameters. As explained in Section 4, to ensure that this process is performed flawlessly, all the components are placed in a safe state before they are reconfigured. Then, once the reconfiguration plan has been executed, the components are activated and the system execution resumes.

In order to address *Challenge* 5, we have implemented all the functions very efficiently, reducing by as much as possible both the resources consumed and the response time.

3.3. 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 functionalities. 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 on the mobile phone. The quality of the received video is variable (high, medium, low).
3. Check-in at the stands/events to track your activity. The technology used is variable and either NFC or Bluetooth can 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.

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

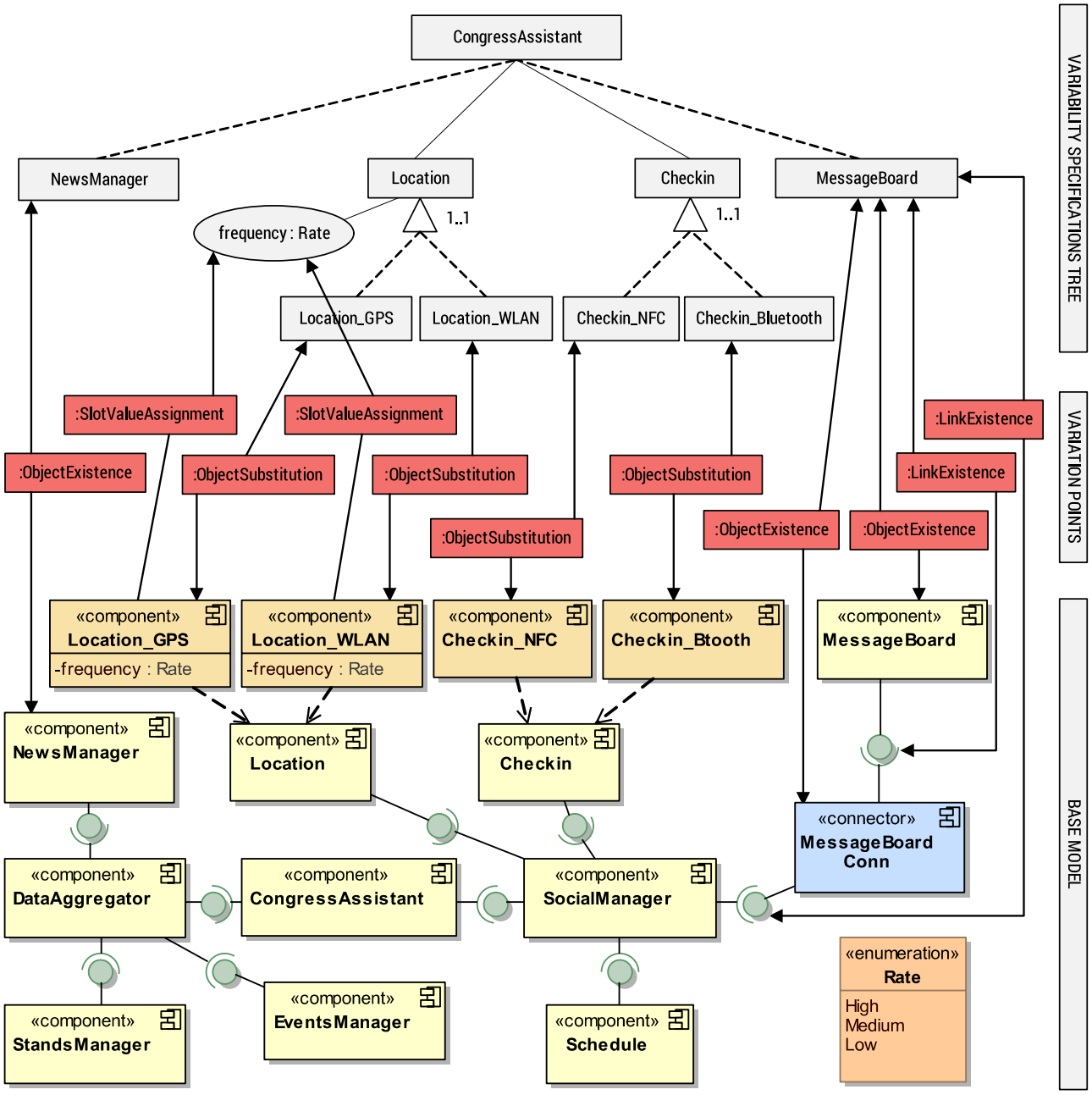

Fig. 3. Case study (excerpt of the base model and the variability model).

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.

Fig. 3 shows the variability model of our case study, which is specified using CVL, as well as a component-and-connector view of its software architecture—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 reasons of legibility. The VSpec tree determines the set of valid configurations of the application. We use two different kinds of VSpecs: *choices* and *variables*. Choices, which are shown in the figure as rectangles, are evaluated as *true* or *false*. On the other hand, *variables*, which are shown as ovals, can be evaluated as values of different types. A VSpec can be bound to its parent by a solid or a dotted line. In the first case, it means that in the case that the parent has been decided to be true, a value has to be decided for that VSpec too. Then, a dotted line means that if the parent has been evaluated as false, it is not necessary to decide a value for this VSpec. For instance, if the `Location` VSpec is decided to be false, it is not necessary to decide a value for `Location_GPS` or `Location_WLAN`.

VSpecs are bound to variation points, which specify how the elements in the base model (i.e. the software architecture) are changed according to the decisions taken in the VSpec tree. For instance, using `ObjectExistence` variation points, we can define *optional* components. In particular, an `ObjectExistence` variation point is bound to the `NewsManager` VSpec and the `NewsManager` component, which means that, in the case that the `NewsManager` VSpec is decided to be false, the `NewsManager` component is removed from the resulting configuration. On the other hand, `ObjectSubstitution` variation points allow the specification of different variants of the same component. For instance, the `Location` component is responsible for providing the location of the owner of the mobile device

Table 1
Excerpt of the OCL constraints of our case study.

Number	Constraint
#1	`VideoReceiver implies EventsManager`
#2	`VideoReceiver.quality='High' implies VideoDecoder.HardwareDecoder`

Table 2
Resource usage and utility information table.

Element	Battery consumption (mA)	Utility
`Location_GPS`	75	100
`Location_WLAN`	20	60
`Location_WLAN.frequency.High`	30	30
`Location_WLAN.frequency.Medium`	20	20
`Location_WLAN.frequency.Low`	10	15

for tracking his/her position, and can be realized either by the `Location_GPS` or the `Location_WLAN` variants, the GPS variant measurements being more precise but also much more costly with regard to battery consumption. As we can see in the figure, `ObjectSubstitution` variation points are bound to the `Location_GPS` and `Location_WLAN` components, as well as their corresponding VSpecs. Therefore, in the case that the `Location_GPS` VSpec is decided to be true, the `Location_GPS` variant is included in the configuration, being the `Location_WLAN` variant excluded.

We can also define parameterizable components using `SlotValueAssignment` variation points. For instance, 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 these components, and is bound to the `frequency` VSpec in the VSpec tree. Therefore, when a value is assigned to this VSpec, it is also assigned to the value of the bound components.

Finally, `LinkExistence` variation points enable the definition of optional links between elements.

As seen in Section 2.2, CVL allows the specification of OCL constraints in order to delimit the degree of variability. These constraints are defined as relationships between different VSpecs of the VSpec tree. Table 1 shows some constraints of our case study that cannot be included in Fig. 3 for reasons of legibility (it is not possible to show the complete VSpec tree of our case study). For instance, since video streams are related to events, we include constraint #1, where both `VideoReceiver` and `EventsManager` are VSpecs of the full VSpec tree, and states that if the `VideoReceiver` VSpec is included in the configuration, it is mandatory to also enable the `EventsManager` VSpec. It is also possible to specify constraints involving the values of variable VSpecs. For instance, we include constraint #2 which states that, for playing high-quality video, it is mandatory to use a hardware video decoder.

The information about resource usage and utility is provided as a table in which each entry specifies the resource usage and the utility of each VSpec in the VSpec tree. This information, together with the VSpec tree, is the input for DAGAME, 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 2. We can see that locating a user using a GPS consumes more battery than using a WLAN and thus, from the point of view of the consumption of resources, the `Location_WLAN` option would be better. However, the precision in the localization is relevant to the user and for this reason the utility of the `Location_GPS` element is higher that the utility of the `Location_WLAN`. In the same way the measuring rate used to locate the user can be said to be similar. A higher value consumes more resources but it is more useful. Thus, the software architect should assign a higher utility value to those elements that are more useful for the user. Measuring the usefulness of each element is beyond the scope of this paper and can be established following different criteria, as stated previously.

4. Dynamic reconfiguration service

As described previously, the DRS is responsible for adapting the applications at runtime according to the current context, while the CMS provides the DRS with context information. Therefore, in this section we focus on: (1) the plan stage of the MAPE-K loop (plan generation), which is part of the DRS and uses as inputs the variability model, the context information and the utility and resources information, and (2) the execution stage (plan execution), in which the application is finally reconfigured taking into account the plan generated in the previous stage.

4.1. Generation of the reconfiguration plan

As Brataas et al. show in [30], the reconfiguration time is divided into 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 to find 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 highly efficient way. However, this is an NP-hard problem [26] and, therefore, it is impossible to use exact techniques to solve this optimization problem for our purpose. Concretely, as shown in [23], 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 the DAGAME algorithm that focuses on optimizing feature models configurations, to optimize the VSpecs tree, as it has proven to be efficient and produces nearly-optimal results [22]. Concretely, this algorithm is able to generate configurations with about 90% optimality, which means that the utility of the solutions obtained using this algorithm is approximately 90% of the utility of the optimal configuration that would be obtained using an exact algorithm. Furthermore, thanks to the huge improvement in the processing and memory capacities of smartphones, using artificial intelligence algorithms in mobile devices is feasible and efficient, as is proven in this paper (see Section 5).

4.1.1. Modelling the optimization problem

As has been explained in Section 2, when using GAs the solution candidates for the optimization problem are known as *chromosomes*, which are part of a *population*. Each chromosome consists of a set of binary variables known as *genes*, each one modelling a property of the solution. In this section we show how we model the problem of optimizing VSpecs trees configurations. Concretely, in our case, VSpecs are mapped to genes in this way:

1. The VSpec tree is traversed either breadth-first or depth-first. Choosing any order criterion in particular does not affect the result, but this criterion should be maintained throughout the execution of the reconfiguration process.
2. Each *choice* VSpec is modelled as a gene. In the case that the gene is evaluated as *true*, the VSpec is also decided to be true.

Table 3
VSpecs tree chromosome modelling.

Index	VSpec	Variation point	Architectural element
0	CongressAssistant	–	–
1	NewsManager	ObjectExistence	NewsManager
2	Location	–	–
3	Location_GPS	ObjectSubstitution	Location, Location_GPS
4	Location_GPS.frequency=High	SlotValueAssignment	Location_GPS.frequency
5	Location_GPS.frequency=Medium	SlotValueAssignment	Location_GPS.frequency
6	Location_GPS.frequency=Low	SlotValueAssignment	Location_GPS.frequency
7	Location_WLAN	ObjectSubstitution	Location, Location_WLAN
8	Location_WLAN.frequency=High	SlotValueAssignment	Location_WLAN.frequency
9	Location_WLAN.frequency=Medium	SlotValueAssignment	Location_WLAN.frequency
10	Location_WLAN.frequency=Low	SlotValueAssignment	Location_WLAN.frequency
11	Checkin	ObjectExistence	Checkin
12	Checkin_NFC	ObjectSubstitution	Checkin_NFC
13	Checkin_Bluetooth	ObjectSubstitution	Checkin_Bluetooth
14	MessageBoard	ObjectExistence	MessageBoard

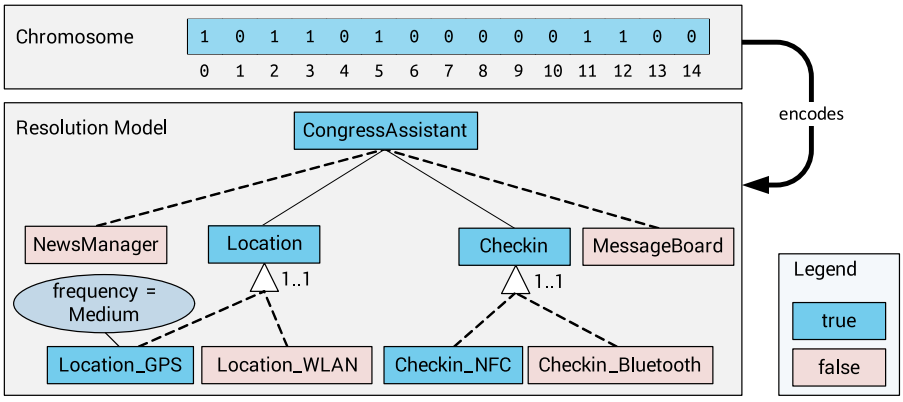

Fig. 4. Example of a chromosome encoding a valid resolution model.

3. Each *variable* VSpec is modelled as a set of genes. Concretely, a gene is added for each possible value of the VSpec. Only one of these genes can be evaluated as true simultaneously. So, the gene whose value is true provides the value for this *variable* VSpec.

Therefore, a CVL *resolution model* is obtained from a set of *genes* the values of which have been already decided. Note that, unlike other optimization problems, in this case not all combinations of gene values represent valid solutions. In order for a solution to be valid and then encode a resolution model for our VSpecs tree, it has to satisfy the VSpecs tree constraints (VSpecs connections) and other constraints between VSpecs and should not exceed the available resources.

Table 3 shows how the VSpecs tree of the excerpt of our case study (see Fig. 3) is modelled in a chromosome, where the column *index* shows the position of the VSpec in the chromosome. An example of a valid configuration can be found in Fig. 4. In this example, the optional VSpecs NewsManager and MessageBoard are evaluated as *false* and, therefore, the genes 1 and 14 have the value 0. Location_GPS variant with a *medium* frequency is enabled (genes 3 and 5), as well as the Checkin_NFC variant (gene 12).

Next, the utility function needs to be defined. In GAs the utility function is typically used to evaluate the quality of each chromosome, allowing the comparison of the chromosomes in the population. In the context of a DSPL with CVL, the utility function has to be seen as a function that calculates the utility of each generated configuration, based on the utility of each element of the VSpec tree. Assuming that the VSpec tree is modelled in chromosomes the length of which is n genes, we define the utility of a configuration (i.e. the utility function) as:

$$U_F(C) = \sum u_i v_i, \quad 1 \le i \le n$$

where:

$$U = \{u_i\}, \quad 1 \le i \le n$$

is a vector that defines the utility of each gene in a chromosome, in the case that it is included in the configuration, and

$$V = \{v_i\}, \quad 1 \le i \le n, \ v_i \in \{0, 1\}$$

is a vector that indicates, for each gene in the chromosome, whether it is included in the configuration.

Therefore, the utility function is the summatory of the utility of the elements that are part of the configuration (i.e. the summatory of the utility of the genes in the chromosome the value of which is 1).

4.1.2. Execution of the genetic algorithm

When the context changes and the DRS decides that it is necessary or it is worth reconfiguring the application, the genetic algorithm is executed in order to generate a reconfiguration plan and then deploy a new architectural configuration tailored to the current execution context. In Section 2 it is shown that three different stages are distinguished during the execution of genetic algorithms: (1) generation of the initial population; (2) evolution through generations, and (3) returning the solution. In this section we cover these stages in more detail, showing how they are performed for our optimization problem in particular and illustrating them with an example.

Population initialization. In this stage, the genetic algorithm generates a set of initial chromosomes, which represent valid solutions and therefore resolution models for our VSpecs tree. Furthermore, these resolution models do not exceed the available resources. Since the values for these genes are randomly selected,

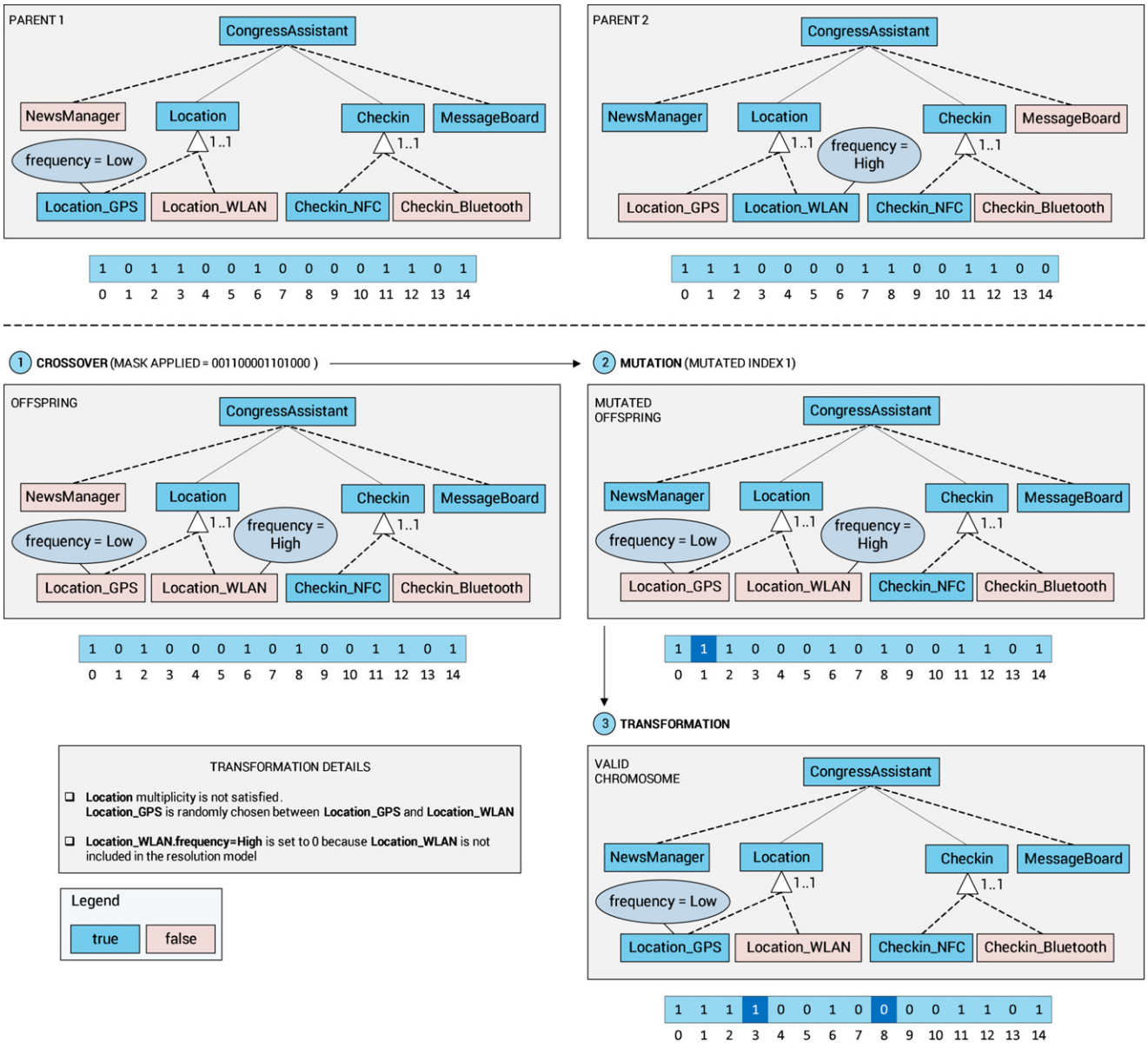

Fig. 5. Applying the DAGAME algorithm in the Dynamic Reconfiguration Service.

it is necessary to apply some transformations to each one to get a valid solution from each randomly-generated one (see details below). As it is shown in Table 3, the VSpecs tree of the excerpt of our case study is modelled as a chromosome containing 15 genes.

Evolution through generations. Once an initial population of valid resolution models has been generated, the next step is to evolve the population through generations in order to find resolution models leading to architectural configurations which better fit the current context. Fig. 5 shows an example of this evolution process. In each generation, a new chromosome is generated and introduced in the population by applying several *operators*:

1. `Selection`. Two chromosomes of the population are selected. In our algorithm we select the two best solutions in the population, which are represented by `Parent 1` and `Parent 2` at the top of Fig. 5.
2. `Crossover`. The genes of both chromosomes are combined by selecting genes from both of them, providing as a result an offspring chromosome. The genes of the offspring are taken from both parents applying a mask that is randomly generated. Concretely, for the example shown in the figure, the random mask applied is `001100001101000`, where 0 and 1 imply that the gene value is taken from `Parent 1` and `Parent 2` respectively.
3. `Mutation`. Once the offspring has been generated, a mutation is introduced in the offspring by randomly choosing a gene of the chromosome and switching its value. Concretely, in this case, the value of gene 1 has been changed from 0 to 1, which implies that the `NewsManager` VSpec has been added to the resolution model. However, the current resolution model is invalid and needs to be modified in order to get a valid one.
4. `Trasformation`. The resulting chromosome is invalid and needs to be modified in order to get a valid one. Although this paper does not cover our genetic algorithm in detail, the transformation process is basically as follows:
   (a) For each VSpec evaluated as true, the algorithm makes sure that its VSpec parent and mandatory children are also evaluated as true, changing their value if necessary,
   (b) Descendant VSpecs groups multiplicity is checked to ensure that it is satisfied and
   (c) Each time a VSpec is evaluated as true its resource usage is taken into account, avoiding the enabling of VSpecs which leads to an excessive resource usage.

The mutated offspring generated after applying the `mutation` operator is then transformed, obtaining a new chromosome which represents a valid resolution model. Concretely, we can see that in the mutated offspring in Fig. 5 that the multiplicity of the `Location` VSpec is not satisfied. This is solved by randomly choosing one between `Location_GPS` and `Location_WLAN` and adding it to the configuration (in this case, `Location_GPS` has been added). Furthermore, since `Location_WLAN` is not included in the configuration, the value of its `frequency` parameter is also removed. As a consequence, the values of the genes 3 and 8 of the chromosome have been changed.

The resulting chromosome is then included in the population, replacing the chromosome with the lowest utility value, thereby improving the overall fitness of the population.

Return the best chromosome. The evolution process is then repeated until a stopping condition is reached. For instance, when a maximum number of generations is reached or when the population has not successfully evolved after a certain number of consecutive generations. In our case, we apply both conditions, stopping the evolution when one of them is reached. At the end, the chromosome with highest utility is returned as the solution for the optimization problem. This chromosome represents the resolution model that leads to the best architectural configuration found for the current execution context.

4.1.3. Calculating the differences between configurations

Once a new configuration, which fits the current context, has been found, we need to generate a reconfiguration plan that allows us to deploy the new configuration. Therefore, first we need to know the differences between the previous configuration and the next one, which has been obtained as a result of the execution of the genetic algorithm.

Fig. 6 shows an example in which the differences between two different configurations are calculated. Initially, differences are calculated at the chromosomes level. Then, these differences are propagated to the VSpecs tree and, thanks to CVL and its formal definition, we can propagate the differences in the VSpecs tree to the variation points and finally the software architectural model. As seen in Section 3.2, finding the differences between the previous and the new configuration is quite straightforward because the resolution models are encoded as a sequence of ones and zeros. Concretely, the following steps are involved in the process of finding the differences between configurations:

Find the genes with different values. The differences between the two chromosomes, which represent the previous and the new configuration, are found by applying an XOR operation. As a result, we know which genes have changed between them. In the example shown in Fig. 6, genes 1, 4, 6, 12 and 13 are different.

Propagate differences to the resolution model. The VSpecs in the resolution model which are affected by the genes whose value have been modified are determined. In our example, gene 1 has changed its value from 1 to 0 which means that, in the CVL resolution model, according to the modelling criterion specified in Table 3, the VSpec `NewsManager` which was previously evaluated as *true* is now evaluated as *false* (difference called `VS1` in Fig. 6).

Identify variation points. It is necessary to determine the variation points bound to the VSpecs identified in the previous step. Depending on the kind of variation points, the procedure differs:

1. `ObjectExistence`, `LinkExistence` and `ObjectSubstitution` variation points. The linked architectural elements need to be removed or added from the configuration if the variation points are enabled or disabled in the new configuration, respectively.
2. `SlotValueAssignment` variation points. In the case of disabled variation points, no further actions are necessary. On the other hand, if the variation point is enabled, the associated value is assigned to the linked component parameter.

In our example, the VSpec from `VS1` is linked to an `ObjectExistence` variation point (which we call `VP1`). Since the VSpec value has changed from *true* to *false*, the variation point's new state is *disabled*. On the other hand, the variation point `VP5`, which is linked to `VS4`, is now *enabled* because `VS4` has changed its value from `false` to `true`.

Identify the differences in the architectural model. The last step is to propagate the different variation points to the architectural model. In our example, `VP1` is linked to the `NewsManager` component. Since it is an `ObjectExistence` variation point and its status is *disabled*, this component is removed from the new configuration. On the other hand, since the `ObjectSubstitution` variation points `VP4` and `VP5` are now disabled and enabled, respectively, the `Checkin_NFC` component is removed from the software architecture, being replaced by the `Checkin_Bluetooth` component as the chosen variant for `Checkin`.

4.2. Execution of the reconfiguration plan

In the rest of this section, this reconfiguration plan is executed in order to deploy the new architectural configuration. The reconfiguration plan allows us to know which components and connectors should be added and removed, and also which parameters' values should be updated.

4.2.1. Safe architectural reconfiguration

In order to guarantee a successful reconfiguration process, it is necessary to place the system in a consistent state before the reconfiguration plan is executed. Therefore, currently running transactions should be paused or neatly finished and the current components' state should be saved and restored as necessary.

Safe adaptation is typically addressed by ensuring the property of *quiescence* [31]. According to Kramer et al. [31] the components of the software architecture can be in one of the following states:

1. *Active*. The component can accept, initiate and execute transactions.
2. *Passive*. The component should accept transactions and execute them but (1) it should not initiate new transactions and (2) it is not involved in a transaction it initiated.

However, a *passive* state is not a guarantee of consistency because passive components may still have to accept new transactions or finish them. To this end, the property of *quiescence* is defined, which ensures that the system consistency is preserved when a component of the software architecture is reconfigured. According to the definition provided in [31], an architectural element is *quiescent* if the following criteria are satisfied:

1. It is in a *passive* state.
2. It is not currently executing a transaction.
3. No transactions initiated by other elements will require it.

Therefore, when a component is *quiescent*, we can safely remove or modify it. Typically, the transitions from each one of these states is described using a state machine [31,32], as shown in Fig. 7.

A weaker condition than the *quiescence* property is the *tranquillity* [33] property. *Tranquillity* is less disruptive in the application running, but is still a sufficient condition to place an application in a consistent state before and after the runtime changes. Note that although both concepts can be considered in our approach, our current implementation is based on the *quiescence* property.

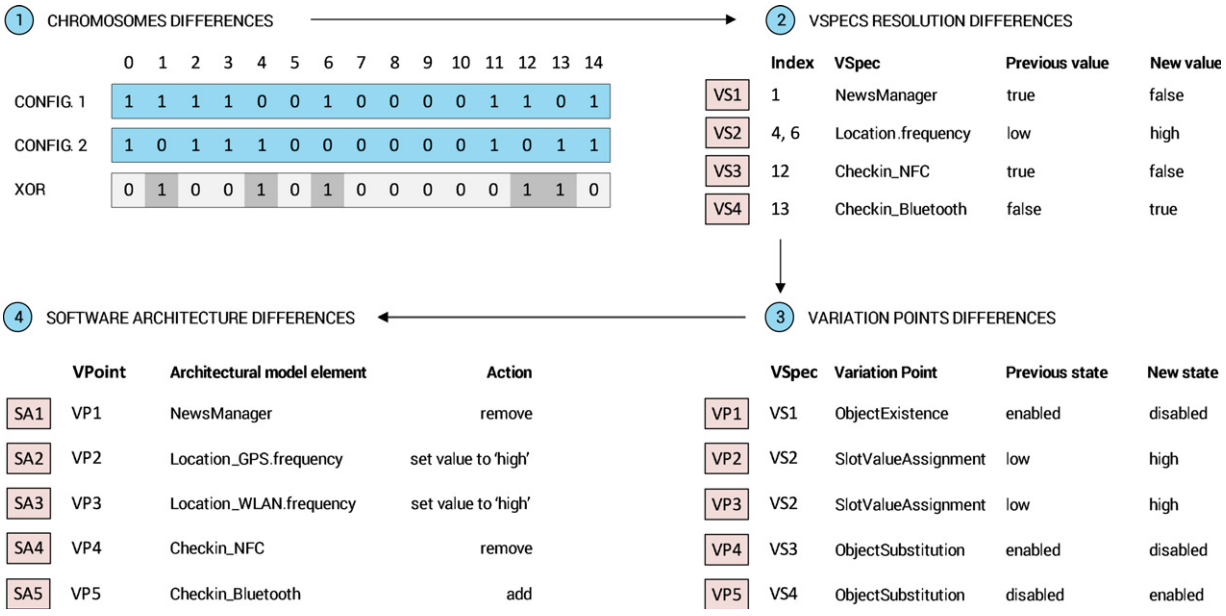

Fig. 6. Calculating the differences between two architectural configurations.

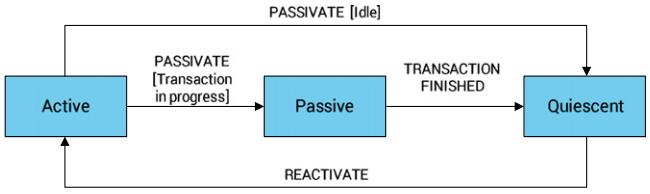

Fig. 7. Adaptation state machine.

4.2.2. Software architecture reconfiguration management

In our approach, the execution of the reconfiguration plan is initiated by the DRS, but the responsibility of transiting to the *quiescent* state rests with each one of the components involved. In particular, once the reconfiguration plan has been calculated and the set of components that are being removed, added or whose parameters' values are being updated is known, the DRS asks the following components to transit to *quiescence*:

1. Components which are being removed.
2. Components containing a parameter the value of which is being updated.
3. Components which require another component that is being added to the software architecture.

Then, according to the adaptation state machine shown in Fig. 7, when a component receives a *passivate* message, it transits to *passive* or *quiescent* state, depending on the transactions it is involved with. Furthermore, it also sends a *passivate* message to all the components meeting any of the following rules, which will react identically:

1. Components requiring its interface.
2. Components currently involved in a transaction with it.

Then, when all the components involved are in a *quiescent* state, the DRS proceeds to:

1. Update the values of the parameters which have been modified in the new configuration.
2. Remove all the components which are not included in the new configuration, as well as the connectors linked to them.
3. Add the new components and the connectors linked to them.
4. Activate all the components in the software architecture.

Fig. 8 shows an extract of a sequence diagram illustrating the execution of a reconfiguration plan. Note that, for reasons of legibility, only the most relevant components are shown. Initially, the DRS sends a *passivate* message to the components that are being removed (`MessageBoard`), to those requiring a new component (`DataAggregator`) and to those containing a parameter the value of which is being modified (`Location_GPS`). Then, following the criteria previously specified, these components send *passivate* messages to the ones they are linked to, in case they are not already in a *passive* or *quiescent* state. For instance, since `SocialManager` requires `MessageBoard` and, furthermore, they are involved in a transaction, `MessageBoard` sends a *passivate* message to `SocialManager`. In like manner, `DataAggregator` asks `CongressAssistant` to transit to *passive* state. When a component receives the *passivate* message, it transits to `quiescent` state (see `Location_GPS`) in the case that it is not involved in any transaction or to `passive` state, neatly finishing all the pending transactions it initiated (as in the case of `CongressAssistant` and `SocialManager`). Finally, when all the transactions involving a component have finished, it transits from *passive* to *quiescent* state.

5. Evaluation

In this section we evaluate our approach, showing that the DRS is efficient enough to be executed on mobile devices (≈21.17 ms of execution time) and that it is able to generate nearly-optimal architectural configurations at run-time tailored to the current execution context (>87.4% of optimality in comparison with the optimal solution). Furthermore, we also demonstrate the benefits of our dynamic reconfiguration approach by showing how, in our case study, the duration of the battery and the overall utility provided to the user by the application are improved when DRS is used (+45.9% in battery duration and +10.31% in provided utility).

5.1. Execution time and optimality

In [22] we evaluated the DAGAME algorithm by applying it to a set of randomly-generated FMs. In this section, we demonstrate that DAGAME is capable of finding nearly optimal configurations, and in an efficient way, when it is applied to our case study.

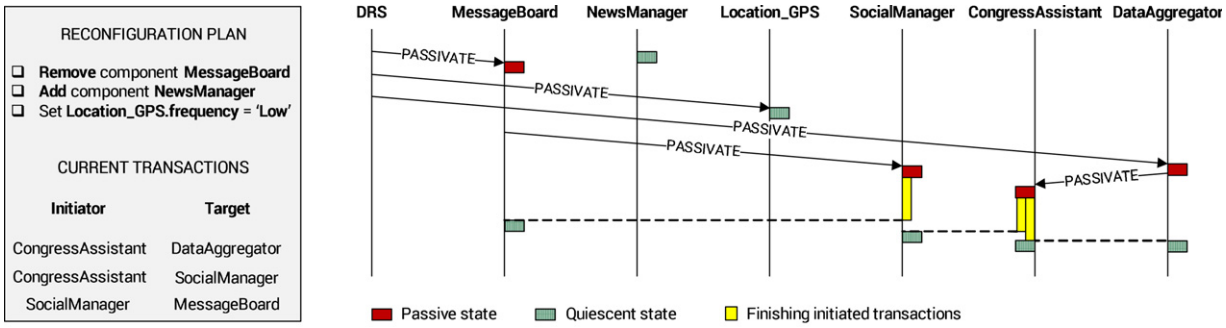

Fig. 8. Partial sequence diagram of a reconfiguration plan execution.

The VSpecs tree that models the variability of our case study contains 16 128 configurations that satisfy all the constraints, and for each one of the VSpecs of this variability model, we have modelled the estimated battery consumption (in mA) and the utility that provides to the user.

For the evaluation of DAGAME when it is applied to our case study, we have compared the solutions obtained using DAGAME with the optimal solutions. In order to find the optimal solutions we have generated a list of all the valid configurations, then calculated the battery consumption and the utility of each of them. The search of the optimal solutions was done on a desktop computer as it is too expensive to run it on a mobile device.

We have executed DAGAME on a LG Nexus 4 smartphone running Android 4.3, applying it to our case study and calculating the execution time and the utility obtained for each possible value of the application's battery consumption, ranging from 50 to 385. For each one of these values, the experiments have been repeated 1000 times. We have configured DAGAME with a population size of 25 chromosomes and a maximum of 20 generations, stopping the algorithm if no better configurations are found after 5 consecutive generations. An exhaustive optimization of these parameters has not been performed because the selected settings have been proven to provide good results.

Fig. 9 shows the utility of the solutions found by DAGAME and compare it with the optimal utility. It can be seen that the solutions obtained by DAGAME are very close to the optimal ones. If we apply the concept of *optimality* [23], which is defined by Guo et al. as the ratio between the utility of the solution obtained using DAGAME and the utility of the optimal solution, obtained using the exact method, we can see that, in the worst case, the optimality of the solutions obtained using DAGAME is higher than 87.4%. This is a very high degree of optimality, particularly taking into account that the optimization problem is NP-hard.

Regarding the execution time, we have measured the mean execution time of DAGAME. The results show that the mean execution time is 21.17 ms, with an execution time in the worst case of 98.15 ms. Therefore, we can consider that DAGAME is efficient enough to be executed on a mobile device without introducing an excessive overhead.

5.2. Benefits of using the DRS

In the rest of this section we focus on showing how the user benefits from the use of our DRS. To this end, we have simulated the execution of our case study in two different scenarios: (1) when the DRS is available and the application is reconfigured and (2) when the DRS is not applied and the application is not reconfigured. In this simulation, the objective of the DRS is to provide the best utility to the user while trying to extend the battery life. To this end, the DRS will try to deploy configurations of the application the battery consumption of which (in mA) does not exceed

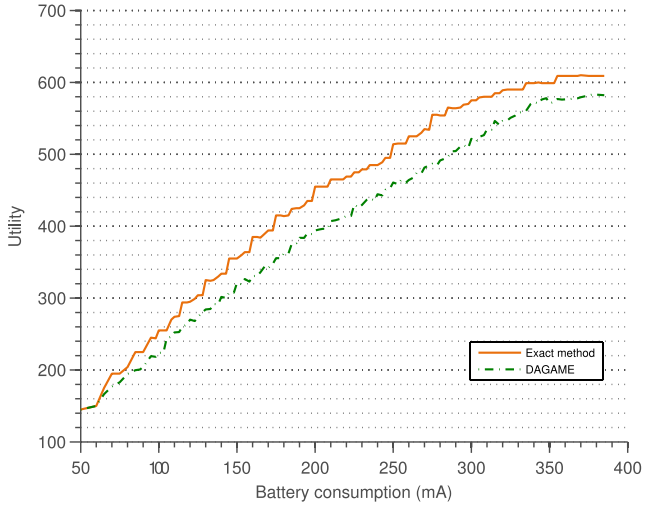

Fig. 9. Utility of the architectural configurations generated by DAGAME.

the remaining battery level (in mAh). Therefore, it generates, as needed, new architectural configurations that provide a nearly optimal utility while ensuring that the battery life is extended.

Figs. 10 and 11 show the results of the simulation when the DRS is enabled and disabled, respectively. We have measured and compared the battery life achieved in both cases, as well as the *accumulated utility*, which is the area under the utility curve in both figures and quantifies the overall utility perceived by the user during the execution of the application from the start until the battery is depleted.

In both cases, the initial battery level is 1200 mAh. As it can be seen in Fig. 10, initially the DRS tries to find a configuration with a high utility since the battery level is very high. The configuration generated consumes 375 mA and, therefore, the battery level starts decreasing until it reaches 375 mAh after 2 h and 12 min. Then, according to the reconfiguration policy that we previously defined, it is necessary to reconfigure the application. The DRS generates a new configuration with a lower battery consumption (335 mA) and, successively, each time the reconfiguration condition is met, a new architectural configuration is generated. It can be seen that, as the new generated configurations consume less current from the battery, the gradient of the battery level curve gets lower as the new configurations are deployed. Consequently, the utility provided by the application is also reduced when less functional configurations are deployed. After 3 h of simulation time, the mobile device is connected to a battery charger, which provides a current of 800 mA to the battery when it is connected. Therefore, since the battery consumption of the application is lower than the current provided by the charger, the battery level rises. As it can be seen, the DRS detects that the battery level is increasing

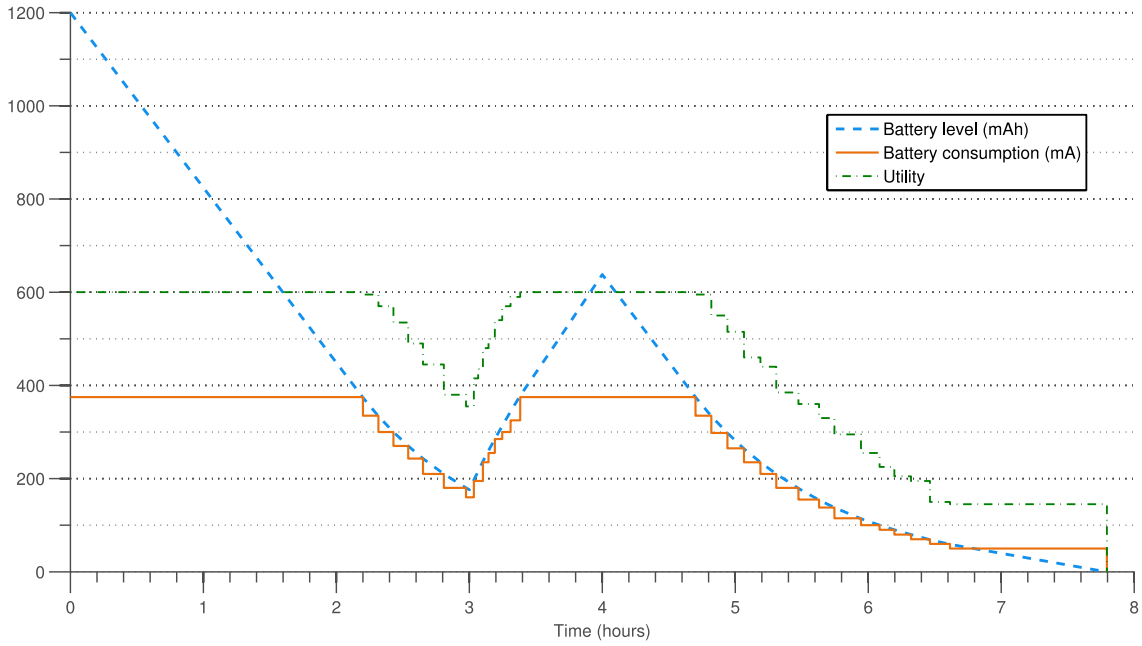

Fig. 10. Simulation of the execution of the case study with DRS enabled.

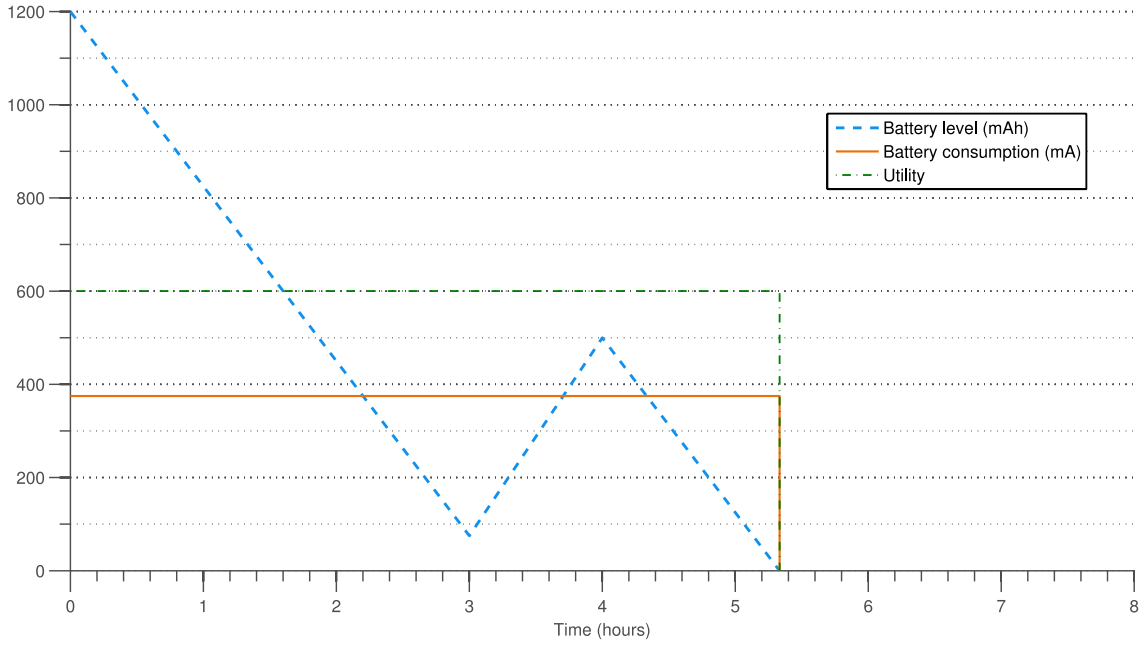

Fig. 11. Simulation of the execution of the case study with DRS disabled.

and, therefore, it generates new architectural configurations that provide higher utility values until no configurations with higher utility values can be found. When the simulation time reaches 4 h, the battery charger is disconnected and the battery level starts decreasing. As expected, the DRS reconfigures the application as necessary trying to extend the battery life. When the simulation time is 6 h and 47 min, the battery level is 50 mAh. However, since it is not possible to find configurations with a battery consumption of lower than 50 mA, the current configuration is maintained and the battery is depleted 1 h later.

On the other hand, when the DRS is not enabled, the initial configuration of the application is executed until the battery is depleted, as it can be seen in Fig. 11. Therefore, the application's battery consumption is constant (375 mA) and, when the battery charger is connected 3 h after starting the simulation, the battery level is only 75 mAh. Since the battery charger introduces 800 mA, the battery is charged with a current of 425 mA for 1 h. At that point, the battery level is 500 mAh, and the battery is finally depleted after 5 h and 20 min of simulation time.

Fig. 12 compares the accumulated utility obtained in both cases. As expected, they are equivalent until the DRS reconfigures the application when the simulation time is at 2 h and 12 min. Then, since the DRS starts deploying architectural configurations with lower battery consumption to extend the battery life, the accumulated utility increases more slowly than when the DRS is disabled. However, if the application is not reconfigured, the battery is depleted after 5 h and 20 min and, therefore, the accumulated utility no longer increases. Finally, before reaching 6 h

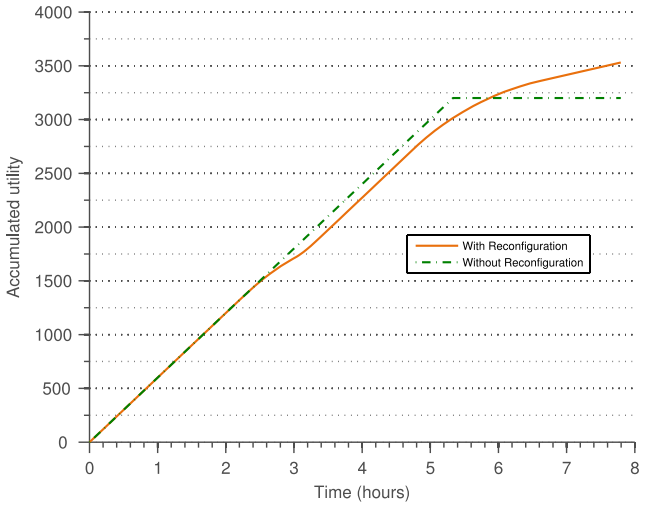

Fig. 12. Accumulated utility during the simulation.

Table 4
Comparison of utility and battery life.

	Battery life	Accumulated utility
DRS enabled	7 h 47 m	3530
DRS disabled	5 h 20 m	3200
Benefit of DRS	**2** h **27** m (+45.9%)	**330** (+10.31%)

of execution time the accumulated utility when the DRS is enabled surpasses the accumulated utility obtained when the application is not reconfigured.

The results obtained in both simulations regarding battery life and accumulated utility are summarized in Table 4. As can be seen, when the DRS is enabled, the battery life is incremented by 45.9% (2 h and 27 min longer). Furthermore, the overall utility perceived by the user once the execution has finished is also higher when the DRS is enabled (an increase of 10.31%, 3530 *versus* 3200 when it is disabled).

According to the results obtained, we consider that, on the one hand, our approach is suitable for providing support for dynamic reconfiguration on mobiles devices without introducing an excessive overhead and, on the other hand, it can generate nearly-optimal architectural configurations that can improve the utility of the applications as well as extend the availability of the resources in the mobile device.

5.3. Discussion

In the rest of this section we discuss some issues of our approach that can be further improved upon, as well as the main threats to validity.

Modelling the resource usage and utility of the application. The optimality of the architectural configurations generated by the DRS relies on the accuracy in the definition of the resource usage and utility of each feature of the application. Therefore, depending on the generated values, the obtained optimality may be slightly different. Further research is necessary to provide mechanisms that can successfully lead this process.

Multiple resources. In our approach we are currently taking into account the availability of only one resource (the battery level in our case study). However, a more general approach would be to prevent the application from exceeding the availability of several resources simultaneously (e.g. memory, network bandwidth, etc.). This limitation is not strictly related to our approach, but to the optimization algorithm. However, as shown in [22], it is very straightforward to modify DAGAME in order to include the ability to take into account the usage of distinct resources. In fact, this extension of DAGAME is part of our future work.

Multiple objectives. In our approach, the DRS optimizes only one objective function, modelled as a utility function. However, under some circumstances, it may be preferable to find a balance between several objectives instead of optimizing just one. Providing this functionality requires a multi-objective optimization algorithm, and it would be necessary to evaluate the suitability of solving multi-objective problems at run-time in constrained devices such as mobile phones, which are our main target. This is also part of our future work.

Evolution of the DSPL. The explicit modelling of variability in DSPL approaches inherently constrains the solution space. In order to cope with this we need to consider that both the variation points of the DSPL and its architectural elements can evolve over time. Moreover, this evolution needs to be managed at runtime in order to reconfigure an application where the old and the new configurations conform to different VSpec trees. As part of our future work we will complete our approach providing the support required to allow the evolution of the DSPL.

Scalability. Brataas et al. have proved that, in the dynamic reconfiguration process, the most relevant task regarding execution time is the decision of the new configuration [30]. They show that the cost of analysing the context data as well as executing the reconfiguration plan can be considered fixed and, therefore, it is very important to make the decision plan task as efficient as possible. In [22] we demonstrated that DAGAME is scalable by applying it to complex FMs of different sizes and, consequently, we can consider that our approach is also scalable.

Perception by 'real' users. In this paper we have evaluated our approach by simulating the execution of an application in a mobile device, a well accepted mechanism for evaluating how an approach would then work in real-world situations. As a threat to validity it could be considered that the results shown in this section have not been complemented with a study on the perception that 'real' users have of the benefits of our approach when they run our application in their mobile devices with/without the DRS service.

Efficiency of the DRS service. In this paper we have compared the execution of our case study when the DRS is available and when it is not applied, and thus the application is not reconfigured. We have demonstrated that using our DRS service the battery life, which is the resource that we focus on, in this paper, is considerably improved. However, as part of our future work we need to complete this evaluation by considering the penalty that the use of the DRS service introduces into the consumption of resources both during the generation of the new configuration and during the reconfiguration of the system.

6. Related work

In this section we discuss how the most relevant approaches in the related work address the main issues identified in the development of self-adaptive systems. Furthermore, as we use an optimization algorithm as part of our approach, we also describe some optimization algorithms that can be found in the literature.

6.1. Self-adaptation approaches

In Section 3.1 we discussed some challenges that DSPL approaches should address and that allow us to identify the main differences between them. In the rest of this section we discuss how these challenges are addressed by the approaches compared here. The results of this discussion are summarized in Table 5, where

Table 5
Self-adaptation approaches.

Approach	Generation of config.	Dynamic var. points modelling	Decision making	Optimization mechanism	Mobile devices
Gamez [3]	Design time	Feature mapping	ECA rules	None	Yes
Vassev [34]	Design time	Custom model	ECA rules	None	N/A
Shen [29]	Design time	Custom model	ECA rules	None	No
Rouvoy [13]	Design time	Custom model	Utility function	Brute force/heuristics	Yes
Brataas [35]	Design time	Custom model	SP model	Brute force	N/A
Rosenmuller [28]	Partially runtime	Feature-oriented programming	Manual / ECA rules	None	No
Trinidad [15]	Runtime	Feature mapping	Manual	None	No
Ayora [16]	Runtime	CVL	ECA rules	None	No
Cetina [20]	Runtime	Custom model	ECA rules	None	N/A
Cheng [17]	Runtime	Strategies	Predefined strategies	None	N/A
Gomaa [18]	Runtime	Feature mapping	Utility function	Heuristics	No
Blouin [19]	Runtime	Context-actions mapping (specific for user interfaces)	Utility function	Genetic algorithm	N/A
Our approach	*Runtime*	*CVL*	*Utility function*	*Genetic algorithm*	*Yes*

there is a column for each challenge. In those approaches where it has been not possible to determine whether they are suitable for mobile devices, the value N/A (i.e. not available) is shown.

Firstly, we can distinguish between those approaches that generate the valid configurations at design time [3,34,29,13,35] and those that generate them at runtime [28,15,16,20,17–19]. We can see that, among the first ones, there are important differences in how they address the challenges under discussion.

In particular, Gamez et al. [3] propose a reconfiguration mechanism, also driven by the MAPE-K loop, that switches between different architectural configurations at runtime. The valid configurations are manually specified and represented using FMs, and a mapping is defined between the features in the FM and the system architecture. The reconfiguration plans are automatically generated from the differences between the configurations. Therefore, both are specified at design-time, which leads to the deployment of sub-optimal configurations at run-time. Their approach has been implemented in several languages, including Java for Android. Therefore, it can be used to develop adaptive mobile applications.

Vassev et al. [34] propose ASCENS, a framework for the representation and reasoning of knowledge, which is defined as a specific interpretation of the context data. In this framework, which enables awareness and self-adaptation, knowledge is specified using KnowLang, a language based on ontologies and Bayesian networks. The description of the system, as well as the reconfiguration polices, are specified in a single model, known as *Knowledge Representation and Reasoning (KR&R) model*. The decision making process is led by a set of predefined policies based on the execution context. Although they propose a case study which involves robots and sensors, as far as we know, their approach has not been evaluated and, therefore, it is not possible to assess its suitability in the case of mobile applications.

Shen et al. [29] propose a dynamic reconfiguration approach based on dynamic aspect weaving where the set of valid configurations is also generated at design time and the reconfiguration process is triggered by ECA rules. The variability is specified using FMs, and they propose a meta-model for specifying *role models*, which are used to bind features to elements of the software architecture. However, their approach relies on the *JBoss-AOP* framework, which is not available in mobile devices. The reason for this limitation is that *JBoss-AOP* relies on *cglib*, a Java library for runtime bytecode generation which is not available in Java virtual machines for mobile devices such as Dalvik, the virtual machine used in the Android operating system.

MUSIC [13] is an OSGi-based middleware for developing context-aware adaptive applications. It is a component based and service oriented approach which principally 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 by evaluating a utility function specified by the software architect. In order to take advantage of MUSIC, the architecture and the variability of the applications are specified together in the same model, according to the meta-model proposed. The main difference between MUSIC (as well as 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. Although the MUSIC middleware does not focus on mobile devices in particular, it is possible to execute it on mobile devices supporting an implementation of the OSGi platform (e.g. Android devices).

Brataas et al. [35] propose a mechanism for extending MUSIC with support for specifying the requirements and the utility of the components of a software architecture. They estimate how many hardware operations are generated by each user action and the application response time. To this end, a structure and performance (SP) model is defined, which allows them to evaluate, at runtime, the resource usage of each configuration and therefore decide whether a configuration is appropriate for the current context. We have identified several drawbacks to this approach: (1) the authors state that it is necessary to evaluate each variant of the application at runtime, which can lead to scalability issues; (2) as far as we know, their approach has not been evaluated on mobile devices and, therefore, it is not possible to determine its suitability for reconfiguring mobile applications and (3) some profiling tools that are necessary to obtain the resources' usage information are not available on mobile devices.

Summarizing, we can see that, on the one hand, in the majority of these approaches, the decision making process is triggered by ECA rules, and therefore they do not provide any optimization mechanism. As stated before, this leads to sub-optimal configurations at run-time. On the other hand, MUSIC is the only one of these approaches that is suitable and available for mobile devices.

Regarding those proposals which generate valid configurations at runtime, we can also find notable differences. For instance, the majority of them do not provide an optimization mechanism [28,15,16,20,17]. Only those whose decision making process is lead by utility function [18,19] use an optimization mechanism (either heuristics or a genetic algorithm).

Rosenmüller et al. [28] present a DSPL approach which partially generates the configurations at runtime. Firstly, part of the variability of the SPL is reduced at design time, generating several DSPLs which are subsets of the complete SPL. So, an adaptation mechanism is included in each DSPL that is capable of generating different configurations of that DSPL at runtime. A SAT solver is used to reconfigure the application at runtime and, since it is demonstrated that SAT problems are NP-complete, it is not appropriate for mobile devices.

Trinidad et al. [15] propose a DSPL approach where each feature in the FM is mapped to a component in the software architecture that can be activated or deactivated. Therefore, using a CSP solver, they perform a real-time analysis of the FM, generating valid configurations at runtime. In this approach, the user manually proposes new configurations of the application, and the CSP solver is used to reason about their validity, among other operations.

Ayora et al. [16] propose a mechanism for managing variability in business processes. At design time, variability is modelled using CVL. Then, following the MAPE-K loop, process variants are adapted using *MoRE-BP*, a reconfiguration engine for web services. Instead of providing an optimization mechanism, the reconfiguration process is triggered by ECA rules. Furthermore, their approach solely focus on web services, making it unsuitable for the development of mobile applications.

In [20] Cetina et al. present an approach for the design of pervasive SPLs which are reconfigured according to changes in the environment. The pervasive system is modelled using the Pervasive Modeling Language (PervML), which includes the ECA rules that trigger the reconfiguration process. The variability of the system is specified using FMs, and both models (PervML and FMs) are related using a realization model, which is an extension that they have incorporated in Atlas Model Weaving (AMW) [36], an Eclipse plugin for model weaving. In order to cope with the complexity of the variability, the SPL is pruned at design time, removing those model elements which are related to scenarios that are supposed to be not useful and therefore limit the configuration space. They have evaluated their approach by applying it to a smart home case study. However, as far as we know, they do not provide details on the execution time and the scalability of their approach. Therefore, we cannot determine whether it is appropriate for mobile devices.

Cheng et al. [17] propose a predictive, instead of reactive, adaptation approach, trying to foresee changes in the availability of resources and lower the disruption to the quality of service provided to the user. To this end, they extend the Rainbow framework [37] with a mechanism for predicting resource availability based on data gathered in the past. The adaptation process is based on predefined strategies, which specify the changes that need to be applied to the system under certain conditions and contexts in particular. Furthermore, this approach has been evaluated by applying it to a web service and, therefore, it is not possible to assess the suitability of their approach in the case of mobile applications.

In [18], Gomaa et al. propose a DSPL approach which enables the dynamic adaptation of software architectures, but it is exclusively focused on service-oriented architectures. The variability is modelled using FMs, the features of which are mapped to the artefacts of the software architecture. However, it does not provide details on how a configuration of the FM is selected at runtime, although it states that this process is usually human assisted. Moreover, this approach has been built using the Self-Architecting Software Systems framework (SASSY), a model-driven framework for service-oriented software systems which is implemented on top of Eclipse Swordfish. Therefore, this work is not suitable for mobile applications.

The most similar approach to ours is the work presented in [19], 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. In their approach, the dynamic variation points are modelled by performing a mapping between the context and the different reconfiguration actions that can be executed at runtime. They use a different optimization algorithm (NSGA-II) 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 considerably lower than the reported in [19].

Summarizing, we have also identified significant differences among those approaches that generate the configurations at runtime. As far as we know, none of them is available for mobile devices, either because they are focused on different kinds of systems or because they use tools which are not available or suitable for these devices.

6.2. Optimization algorithms

In our work 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 other than CVL VSpec trees, such as FMs, and (2) to static SPLs. In [26], 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 [23], but using genetic algorithms, being even faster than the previous one. On the other hand, the proposal of Benavides et al. [38] 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 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.

7. 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 their architectural configuration according to the available resources. To this end, we first model the variability of the application's software architecture using CVL. Therefore, we can take advantage of available algorithms to optimally resolve the application's variability. Concretely, we propose the use of DAGAME, a genetic algorithm, to generate nearly-optimal configurations at runtime using the VSpecs tree, the context information and the resource usage and utility information as input. Then, when a new configuration has been generated, we calculate the differences between the previous and the new configurations, propagating these differences from the variability model to the architectural model. In this way, we generate a reconfiguration plan which is safely executed, placing the components in a quiescent state before they are reconfigured. In order to describe and evaluate our approach we have modelled a case study and we have defined a set of experiments to evaluate the efficiency of DAGAME applied to our case study as well as the optimality of its results. Furthermore, we have simulated the execution of the application on a mobile device with the reconfiguration service enabled and disabled, respectively, evaluating the ability of our DRS to extend the availability of resources in the mobile device. The results obtained show that, on the one hand, DAGAME is efficient and can be used to provide dynamic reconfiguration in mobile devices without introducing an excessive overhead and, on the other hand, that our DRS can significatively extend the availability of resources without sacrificing the overall utility provided by the application to the user. As part of our future work we will extend the evaluation to also measure the penalty in the consumption of resources that is introduced by the DRS service itself.

Additionally, as part of our on-going work, we are defining a multi-objective version of DAGAME, which will allow us to simultaneously take into account multiple objectives for dynamically reconfiguring the applications. Moreover, the new version of DAGAME will consider the availability of multiple resources.

Furthermore, we are extending our approach in order to support a wider subset of CVL features, which will allow the software architect to take advantage of an improved expressiveness.

Finally, we are going to explore the benefits of using the tranquillity property instead of the quiescence property to guarantee the application's consistency during the execution of the reconfiguration plan. If we find that the execution time of the reconfiguration plan can be considerably improved then we will provide an alternative implementation of our dynamic reconfiguration service.

Acknowledgements

This work is supported by the projects P09-TIC-5231, P12-TIC1814, TIN2012-34840 and INTER-TRUST FP7-317731.

References

[1] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, Dynamic software product lines, Computer 41 (4) (2008) 93–95.
[2] IBM, Autonomic computing white paper—an architectural blueprint for autonomic computing, IBM Corp., 2005.
[3] N. Gamez, L. Fuentes, M. Aragüez, Autonomic computing driven by feature models and architecture in famiware, in: Proceedings of the 5th European Conference on Software Architecture, ECSA'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 164–179.
[4] A. Haber, T. Kutz, H. Rendel, B. Rumpe, I. Schaefer, Delta-oriented architectural variability using monticore, in: Proceedings of the 5th European Conference on Software Architecture: Companion Volume, ECSA'11, ACM, New York, NY, USA, 2011, pp. 6:1–6:10.
[5] E.A. Barbosa, T. Batista, A. Garcia, E. Silva, PL-aspectualACME: an aspect-oriented architectural description language for software product lines, in: Proceedings of the 5th European Conference on Software Architecture, ECSA'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 139–146.
[6] H. Gomaa, Designing software product lines with UML 2.0: from use cases to pattern-based software architectures, in: 10th International Software Product Line Conference, 2006, p. 218.
[7] CVL, Common Variability Language. http://www.omgwiki.org/variability/.
[8] J.O. Kephart, D.M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50. http://dx.doi.org/10.1109/MC.2003.1160055.
[9] A. Chan, S. Chuang, MobiPADS: a reflective middleware for context-aware mobile computing, IEEE Trans. Softw. Eng. (2003) 1072–1085.
[10] T. Gu, H. Pung, D. Zhang, A service-oriented middleware for building context-aware services, J. Netw. Comput. Appl. 28 (1) (2005) 1–18.
[11] A. Janik, K. Zielinski, AAOP-based dynamically reconfigurable monitoring system, Inf. Softw. Technol. 52 (4) (2010) 380–396.
[12] N. Paspallis, Middleware-based development of context-aware applications with reusable components, University of Cyprus.
[13] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, U. Scholz, MUSIC: middleware support for self-adaptation in ubiquitous and service-oriented environments, in: Software Engineering for Self-Adaptive Systems, 2009, pp. 164–182.
[14] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, 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.
[15] P. Trinidad, A.R. Cortés, J. Peña, D. Benavides, Mapping feature models onto component models to build dynamic software product lines, in: SPLC, (2), 2007, 51–56.
[16] C. Ayora, V. Torres, V. Pelechano, G.H. Alférez, Applying CVL to business process variability management, in: Proceedings of the VARiability for You Workshop: Variability Modeling Made Useful for Everyone, VARY'12, ACM, New York, NY, USA, 2012, pp. 26–31. http://doi.acm.org/10.1145/2425415.2425421.
[17] S.-W. Cheng, V.V. Poladian, D. Garlan, B. Schmerl, Improving architecture-based self-adaptation through resource prediction, in: B.H.C. Cheng, R.d. Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems, in: Lecture Notes in Computer Science, vol. 5525, Springer, Berlin, Heidelberg, 2009, pp. 71–88.
URL: http://link.springer.com/chapter/10.1007/978-3-642-02161-9_4.
[18] H. Gomaa, K. Hashimoto, Dynamic software adaptation for service-oriented product lines, in: Proceedings of the 15th International Software Product Line Conference, Vol. 2, SPLC'11, ACM, New York, NY, USA, 2011, pp. 35:1–35:8.
[19] A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, 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, 2011, pp. 85–94.
[20] C. Cetina, J. Fons, V. Pelechano, Applying software product lines to build autonomic pervasive systems, in: Software Product Line Conference, 2008. SPLC'08. 12th International, IEEE, 2008, pp. 117–126.
[21] G.G. Pascual, Aspect-oriented reconfigurable middleware for pervasive systems, in: Proceedings of the CAiSE Doctoral Consortium, Vol. 731, CEUR-WS, 2011.
[22] G.G. Pascual, M. Pinto, L. Fuentes, A genetic algorithm for reconfiguring mobile applications using a dynamic software product line approach. Technical Report, Dept. Lenguajes y Ciencias de la Computación, Universidad de Málaga.
[23] J. Guo, J. White, G. Wang, J. Li, Y. Wang, A genetic algorithm for optimized feature selection with resource constraints in software product lines, J. Syst. Softw. 84 (12) (2011) 2208–2221.
[24] N. Bencomo, S. Hallsteinsen, E.S. de Almeida, A view of the dynamic software product line landscape, Computer 45 (10) (2012) 36–41. http://doi.ieeecomputersociety.org/10.1109/MC.2012.292.
[25] G.G. Pascual, M. Pinto, L. Fuentes, Automatic analysis of software architectures with variability, in: Safe and Secure Software Reuse, Springer, 2013, pp. 127–143.
[26] J. White, B. Dougherty, D.C. Schmidt, Selecting highly optimal architectural feature sets with filtered Cartesian flattening, J. Syst. Softw. 82 (8) (2009) 1268–1284.
[27] J. White, D. Schmidt, E. Wuchner, A. Nechypurenko, Automating product-line variant selection for mobile devices, in: Software Product Line Conference, 2007. SPLC 2007. 11th International, IEEE, 2007, pp. 129–140.
[28] M. Rosenmüller, N. Siegmund, M. Pukall, S. Apel, Tailoring dynamic software product lines, in: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering, ACM, 2011, pp. 3–12.
[29] L. Shen, X. Peng, J. Liu, W. Zhao, Towards feature-oriented variability reconfiguration in dynamic software product lines, in: Top Productivity Through Software Reuse, 2011, pp. 52–68.
[30] G. Brataas, S.O. Hallsteinsen, R. Rouvoy, F. Eliassen, Scalability of decision models for dynamic product lines in: SPLC, (2), 2007, pp. 23–32.
[31] J. Kramer, J. Magee, The evolving philosophers problem: dynamic change management, IEEE Trans. Softw. Eng. 16 (11) (1990) 1293–1306. http://dx.doi.org/10.1109/32.60317.
[32] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, D.A. Menascé, Software adaptation patterns for service-oriented architectures, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC'10, ACM, New York, NY, USA, 2010, pp. 462–469.
[33] Y. Vandewoude, P. Ebraert, Y. Berbers, T. D'Hondt, Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates, IEEE Trans. Softw. Eng. 33 (12) (2007) 856–868. http://dx.doi.org/10.1109/TSE.2007.70733.
[34] E. Vassev, M. Hinchey, B. Gaudin, Knowledge representation for self-adaptive behavior, in: Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering, C3S2E'12, ACM, New York, NY, USA, 2012, pp. 113–117. http://doi.acm.org/10.1145/2347583.2347599.
[35] G. Brataas, S. Jiang, R. Reichle, K. Geihs, Performance property prediction supporting variability for adaptive mobile systems, in: Proceedings of the 15th International Software Product Line Conference, Vol. 2, SPLC'11, ACM, New York, NY, USA, 2011, pp. 37:1–37:8. http://doi.acm.org/10.1145/2019136.2019178.
[36] M. Didonet, D. Fabro, J. Bézivin, P. Valduriez, Weaving models with the eclipse AMW plugin, in: Eclipse Modeling Symposium, Eclipse Summit Europe, 2006.
[37] S.-W. Cheng, D. Garlan, B. Schmerl, Making self-adaptation an engineering reality, in: O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. v. Moorsel, M. v. Steen (Eds.), Self-Star Properties in Complex Information Systems, in: Lecture Notes in Computer Science, vol. 3460, Springer, Berlin, Heidelberg, 2005, pp. 158–173.
URL: http://link.springer.com/chapter/10.1007/11428589_11.
[38] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, in: Advanced Information Systems Engineering, Springer, 2005, pp. 381–390.

Gustavo García Pascual is a Ph.D. Student in the Languages and Computer Science Department at the University of Málaga (Spain).

He received the B.Sc. degree in Telecommunications Engineering from the University of Málaga.

He is a member of the CAOSD Group and the GISUM Group, and his main research areas are Component-Based Software Engineering, Aspect-Oriented Software Development, Dynamic Software Product Lines and Self-adaptive middleware platforms.

Mónica Pinto is an associate professor in the Languages and Computer Science Department at the University of Málaga (Spain).

She received the M.Sc. degree in Computer Science in 1998 from the University of Málaga, and her Ph.D. in 2004 from the same University.

She is a member of the CAOSD Group, GISUM Group and she is involved in the AOSD-Europe Network of Excellence (http://www.aosd-europe.net).

Her main research areas are Component-based Software Engineering, Aspect-oriented Software Development, Architecture Description Languages, Model-driven Development and Context-aware Mobile Middlewares.

Lidia Fuentes received her M.Sc. degree in Computer Science from the University of Málaga (Spain) in 1992 and her Ph.D. in Computer Science in 1998 from the same University. She is a Full Professor at the Department of Computer Science of the University of Málaga since 2011 (previously, Lecturer and Associate Professor from 1993). Currently, she is the head of the CAOSD research group.

Her main research areas are Aspect-Oriented Software Development, Model-Driven Development, Software Product Lines, Agent-Oriented Software Engineering, Self-adaptive middleware platforms, Architecture Description Languages and Domain Specific Languages.