Specifying Adaptations through a DSL with an Application to Mobile Robot Navigation

André C. Santos[1,3], João M. P. Cardoso[2], Pedro C. Diniz[3], and Diogo R. Ferreira[1]

1   IST – Technical University of Lisbon, Portugal
    {acoelhosantos, diogo.ferreira}@ist.utl.pt
2   Faculty of Engineering, University of Porto, Portugal
    jmpc@acm.org
3   INESC-ID, Lisbon, Portugal
    pedro@esda.inesc-id.pt

Abstract

Developing applications for resource-constrained embedded systems is a challenging task specially when applications must adapt to changes in their operating conditions or environment. To ensure an appropriate response at all times, it is highly desirable to develop applications that can dynamically adapt their behavior at run-time. In this paper we introduce an architecture that allows the specification of adaptable behavior through an external, high-level and platform-independent domain-specific language (DSL). The DSL is used here to define adaptation rules that change the run-time behavior of the application depending on various operational factors, such as time constraints. We illustrate the use of the DSL in an application to mobile robot navigation using smartphones, where experimental results highlight the benefits of specifying the adaptable behavior in a flexible and external way to the main application logic.

1   Introduction

The continued miniaturization of computing devices has contributed to making embedded systems pervasive in a wide range of diverse contexts and thus with a wide variety of computational requirements (see e.g., [2]). Regardless of the device, be it mobile phones, vehicle equipments, medical instruments, or smart home components, all of these systems embody very stringent requirements in terms of reliability, maintainability, availability, safety, security, efficiency, energy consumption, among others. Overall, the diversity of embedded systems and requirements pose tremendous challenges to the development and maintainability of their software applications. In particular, this software must operate within acceptable performance parameters in resource-constrained environments while being subject to changing operating conditions (e.g., temporary unavailability of sensors, decreasing battery level, real-time requirements, memory limitations, intermittent connectivity).

Run-time adaptability is seen as a viable strategy to cope with these challenges (e.g., [14]). For example, the software implementation could leverage the use of different but equivalent

processing algorithms, changing algorithm parameters, switching sensors, or simply changing the frequency of some computations (see, e.g., [3, 19]). However, implementing dynamic behavior in embedded applications involves a considerable amount of effort, as the inclusion of such behavior in the application is complicated and error-prone due to the high degree of intertwining between application and adaptation code [13]. This additional programming effort usually requires a mixture of conditional coding and low-level operations, which translates into reduced code readability and more difficult maintenance. Furthermore, such efforts typically scale poorly when multiple adaptations are used, making continuous development harder [13]. These problems occur regardless of programming language, device or target platform, and they are further exacerbated by the existing plethora of languages, devices and platforms. In short, developing an embedded application with run-time adaptability is by force of circumstance a complex and time-consuming endeavor (e.g., [14]).

To reduce the development burden, there have been some attempts to support adaptability at several levels and through different mechanisms, for example through context-oriented programming (e.g., [7]). However, no current solution has provided the necessary infrastructure to achieve a flexible and domain-tailored approach. Considering the existing background and related work, the main contribution of the present work is an approach for the development of adaptable software applications for embedded systems based on a domain-specific language (DSL). Our approach can be applied to other fields besides embedded systems, however we emphasize on these types of systems since due to their characteristics, they are often more highly constrained than others, and thus in need for adaptive solutions.

The DSL enables the high-level specification of adaptation policies and strategies, using a flexible and simplified way of defining the rules that produce the necessary run-time reconfigurations, in an external way to the main application logic. The usefulness of run-time adaptability in embedded systems, as well as the advantages of having a dedicated approach to specify and manage the strategies for adaptation, are illustrated through a set of experimental results in a case study application. This work builds upon our preliminary DSL assessment in [18], and the case study is based on a previously developed prototype system for mobile robot navigation [17]. The case study allows us to demonstrate the feasibility of our DSL-based approach to flexibly specify adaptable behavior and easily verify its consistency, when compared to a general-purpose language such as Java.

The remainder of this paper is organized as follows. In Section 2 we introduce an adaptation-aware application architecture. Section 3 describes the DSL. Section 4 demonstrates the use of the DSL in an application to mobile robot navigation. Section 5 discusses relevant related work, and finally Section 6 concludes the paper.

2   Architectural Decoupling for Adaptations

Adaptation is a process, which modifies the behavior of a system in order to improve the interaction with the remaining components or the outside environment. Therefore, an application is adaptable when it is possible to adjust the execution of its main logic, thus making the application behavior dynamic. Our work focuses on the adaptation logic as an independent adaptation policy entity, that defines a procedure, composed by multiple strategies, which are plans of action that achieve a certain goal through a set of adaptation rules. This separation of application and adaptation concerns allows for the reuse of adaptation mechanisms, a higher-level of abstraction, potential for scalability and extensibility, and a simplified approach to integrate adaptations with software applications.

Figure 1 presents a diagram depicting the main entities and relationships involved in

a model where the adaptation logic is external to the application and takes the form of one or more adaptation policies that can target specific reconfiguration concerns (e.g., a policy mainly targeted at reducing the energy consumption of the application; or a policy mainly targeted at increasing the performance of the application). Additionally, an application is influenced by (i) a set of user requirements, i.e., functional and non-functional constraints that the application must comply with in order to perform as intended; (ii) an environmental context, i.e., properties related to the operating conditions (e.g., sound level, light intensity); and (iii) the system where it executes, i.e., the execution platform and the infrastructure (e.g., battery level, network resources).
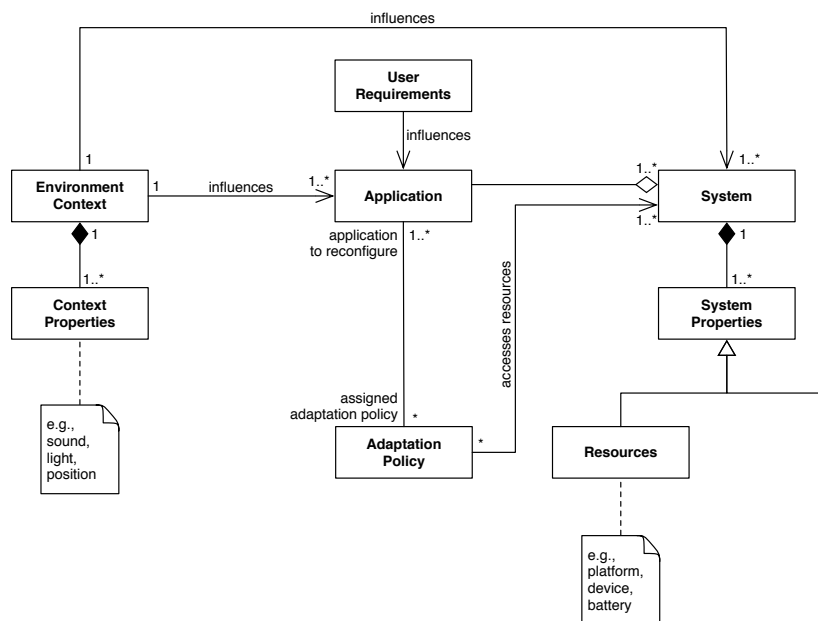
Figure 1 General entity diagram for an application model with an independent adaptation entity.

Moreover, in general, and regardless of software architecture, coding style, etc., an application comprises a series of computational steps, algorithmic components, and input/output parameters. As such, many embedded applications include components that can be configured via different parameters that influence the computational impact of the system as a whole, in terms of several observable metrics such as accuracy, execution time, energy, power, CPU load, quality-of-service, etc. This configurable interface allows for the selection of a number of system configurations thus enabling dynamic and adaptive application behavior.

In order to specify the dynamic behavior of embedded applications, we implement the adaptation logic through a DSL aiming at abstracting the adaptation concerns from the main application logic. DSLs offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages (GPLs) in their domain of application, since they provide a notation close to an application domain, and are based only on the concepts and features of that domain [5, 12]. Given the existence of numerous programming languages, platforms and devices, a DSL-based approach that would allow for a single specification to be deployed in multiple environments would be very useful.

The adaptive behavior specified with the DSL can then be coupled with software applications in embedded systems through numerous mechanisms, such as through joint compilation, interpretation at run-time, mapped to another independent software component, deployed to

another processor in a multicore embedded system, amongst other options. To this end, the proposed DSL is a specification mechanism that allows: (i) domain specificity by providing abstractions for commonly used adaptation actions; (ii) flexibility due to the adaptation independence from the application code and logic; (iii) portability and interoperability, targeting multiple embedded platforms and supporting several programming languages; (iv) verifiability and conflict detection, as domain abstractions allow for an easy understandability of the specified behavior; (v) productivity and comprehension improvement; and also (vi) an easier way to specify and deploy, for the same application, different strategies for different environments/services and/or target devices (e.g., in software product lines).

Although the use of a DSL is an efficient solution for the problem of abstracting and externalizing the adaptable behavior of embedded systems and applications, we also studied and considered other approaches to address the same problem, namely a DSL embedded in a GPL, and the use of a domain-specific library within a GPL. These alternatives represent interesting solutions, but they fall short as an independent solution, since either extending an existing GPL or using a library would be more restricting in terms of interoperability among different platforms and reusability between different systems.

3   A DSL for the Specification of Adaptations

The proposed DSL focuses on adaptation concerns, exposing high-level constructs for looping and setting periodic tasks, conflict avoidance, condition testing, time and memory evaluations, among others. These constructs allow for the flexible specification of adaptation policies that can range from simple parameter changes to complex code reconfigurations.

In the proposed DSL, an adaptation policy is specified as a set of *strategies* comprising four sections: *declarations*, *operations*, *rules*, and *code*. Figure 2 presents a model of an adaptation policy with its main structural components.
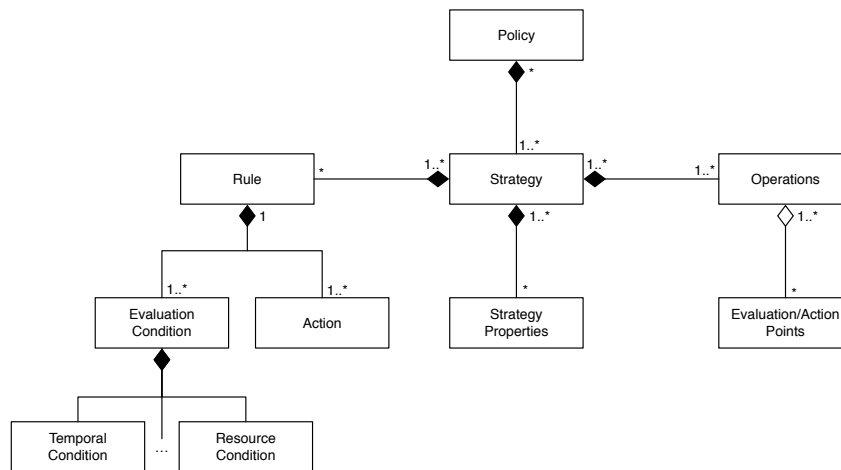
Figure 2 Overview model for an adaptation policy entity.

Declarations are reserved for information that is required for the specification of the adaptation process (e.g., variables to be used, algorithm parameters, imported functions). *Operations* specify mainly system interfacing with information on where the adaptation rules will be triggered (e.g., connection points). The *rules* section specifies the adaptation actions that apply reconfigurations to the application. Finally, in the *code* section, functions and other components can be defined in another programming language (e.g., C) to promote

extensibility. The DSL also provides abstractions to access some application- and system-related properties. This access to certain characteristics is provided by the monitoring tasks that in a later stage are incorporated into the application by the implementation toolchain.

An example of an adaptation policy specified with the DSL, associated with the inference of human activity context, is shown in Listing 1. In this example application, the context is calculated by an inference process imported to the DSL (line 2). Operationally, locations for rule evaluations are defined (lines 4–7). The *rules* section (lines 9–18) defines two rules that express periodic adjustments to the inference when the battery of the device reaches a low level or when the computation time takes longer than the defined rate. In order to acquire data on the energy level of the device, an additional function in Java is defined in the *code* section (lines 20–22), whose details are omitted for simplicity.

■ **Listing 1** DSL specification for an adaptable activity inference system.

```
1  strategy activityAdaptationStrategy{
2     imported function [String context] fftKnnInf(int FftSamples=2048);
3
4     operations{
5        r1 evaluation point "location_1";
6        r2 evaluation point "location_2";
7     }
8
9     rules{
10       r1: every(60sec){
11          if(code.java.getEnergyLvl() < 30){fftKnnInf.FftSamples=512;}
12          else{fftKnnInf.FftSamples=2048;}
13       }
14       r2: every(10sec){
15          if(fftKnnInf.rate < 1Hz){fftKnnInf.FftSamples--;}
16          else{fftKnnInf.FftSamples++;}
17       }
18    }
19
20    code.java{
21       double getEnergyLvl(){ (...) }
22    }
23 }
```

3.1 Language Components

A policy is the adaptation "program" that defines strategies of adaptation, further composed of multiple properties and other components. A strategy is a high-level entity that embodies the mechanisms of adaptation that are tailored to a specific purpose (e.g., energy-aware adaptations, execution time compliance adaptations). Furthermore, the strategy entity can also be defined to receive configuration parameters in order to be adaptable to different situations or conditions. This allows for the strategy elements to be reused across environments with different characteristics.

Declarations   Within a strategy, the *declarations* section allows for the specification of variables, functions, and other components to be used in all other sections. Here, variable declaration has similar semantics to other programming languages with the additional

inclusion of valid value ranges that the variable may assume. Functions from the source application can be imported and are linked to the respective implementations. This section may also define default values for the input parameters of the imported functions.

Operations   The *operations* section describes important operational blocks in the computational process. The main objective of this section is to provide information on the execution of the application through execution blocks, giving an understanding on the execution flow and on interfacing connection points. The *operations* section is built with a main execution block that can be associated with execution properties. Other alternative execution blocks can also be defined. Moreover, sub-block structures can be defined to frame specific steps or components of the application's workflow. Such sub-block structures are used to concentrate operation steps that may be activated or deactivated, allowing a more dynamic and powerful adaptation structure. Operational connection points define references to locations where the adaptations will be triggered and therefore executed in the application's source code. Connection points only define the target location for rule executions; other adaptation properties, such as execution periodicity are defined in the rules section. Points can be associated with function calls or with specific locations in the source code, identified by special-purpose annotations (e.g., "`//@ evaluation point location_1`" for Java).

Rules   The *rules* section specifies multiple adaptation actions, responsible for performing the necessary changes that control the behavior of the target application. An individual rule is composed of an identifier, a triggering condition (e.g., periodically or by events) and the adaptation action code. The execution of a rule is atomic in the sense that its operations either all occur, or nothing occurs, denoting an atomic transaction. Rule management is conducted in this section and thus adding, removing or modifying existing rules can be accomplished without scattered changes. The existence of multiple rule blocks assigned to different conditions or events could cause conflicts, as incompatible actions could be invoked if multiple rules where activated simultaneously. However, with a centralized location for the adaptation rules, verification and validation can be more easily accomplished. The resolution of conflicts is performed through prioritization, where rule blocks are prioritized by their order of specification (default conflict solver) or through explicit prioritization using an `evaluate` control command (specific conflict solver), where boolean logic and prioritization functions (e.g., `first`) can be applied. Additionally, predicates can be used for finer grain control of the flow of rule execution, providing a simpler yet powerful mechanism to protect against incompatibilities arising from rules which for example try to access common resources.

Code   The *code* section allows the developer to extend the DSL by adding functionality that is not present (e.g., platform-specific code), or to extend the application without making changes directly to it. Variables and functions defined in this section have a global scope and thus can be used in all other sections. The *code* section must indicate the programming language in which the enclosed programming code will be specified. The use of this section comes with the cost of reducing the DSL's interoperability, as language- and platform-specific code may be introduced here.

3.2   Policy Correctness

In an adaptation policy, it is likely that reconfigurations, such as a parameter changes, may lead to incompatibilities, execution errors and integrity conflicts. Addressing these conflicting situations is of paramount importance for policy correctness, and thus verification

and validation are required processes conducted on several levels for evaluating different aspects (e.g., adaptation rule conflicts), and different targets (e.g., all or only specific DSL sections). In this paper, we focus on analyzing the problems and conflicting situations for the *rules* section, which is a core component in the specification of the adaptable behavior. In particular, rules and their actions are potentially in conflict if they: (i) share at least a subset of triggering conditions, causing more than one rule to be triggered to execute at the same time; (ii) manipulate a subset of the same parameters, which may cause incompatible behavior in the execution of actions; (iii) override or overlap themselves, causing one rule to become unreachable or redundant; or (iv) are incompatible due to requirements or objectives, since even with different triggering conditions, or different actions, there can be additional specific functional requirements by the stakeholders.

In order to better perceive and assess the set of adaptation rules defined, we propose a verification process based on automata theory [10]. This approach allows to model the *rules* section into directed non-deterministic finite automata where conflicting situations can be identified, both statically and dynamically. Adaptation rules can be viewed as an automata with a set of adaptation states, an alphabet of different triggering conditions, and a transition function that maps the transformation from one adaptation state to another, according to the provided input condition. Through automata operations other properties and characteristics can be extrapolated, for example, (i) the product operation provides the combination of all possible adaptation states and transitions; (ii) minimization allows the removal of useless and unreachable states; or (iii) intersection to identify common states. Furthermore, with an automata-based model, adding, removing and changing rules, can be easily perceived and thus analyzed for conflicts.

3.3 Interfacing and Implementation

In this work, we developed a toolchain that incorporates the independent adaptations into the target source code of the application to be adapted. First, the developer must analyze and evaluate the application source for possible adaptations. Second, depending on the application there may be minor modifications of the source code to be done to explicitly identify functions, inputs, and outputs. With the application better prepared to be adapted, it is possible to specify the adaptations using the DSL. The adaptation code specified with the DSL must then be verified and validated to assess potential errors and conflicts. With a valid DSL adaptation specification, the DSL code is translated into the target source code language (e.g., DSL → Java). The compilation and code generation allow the weaving of the adaptations into the application's original source code, and so the adaptations are incorporated at compile-time. With the adaptations incorporated, the complete application can be compiled using standard compilers (e.g., javac). The adaptable application can now be deployed to the execution environment.

4 Application to Mobile Robot Navigation

This section presents a case study for adaptation specification based on our own previous work [17], where we developed a navigation system comprising a Lego NXT Mindstorms robot together with a Nokia N80/N95 smartphone (see Figure 3). The mobile robot was intended to explore the environment while simultaneously inferring its location. In this system, the mobile robot is controlled by the smartphone, where navigation algorithms are executed, generating controls that are transmitted back to the robot.
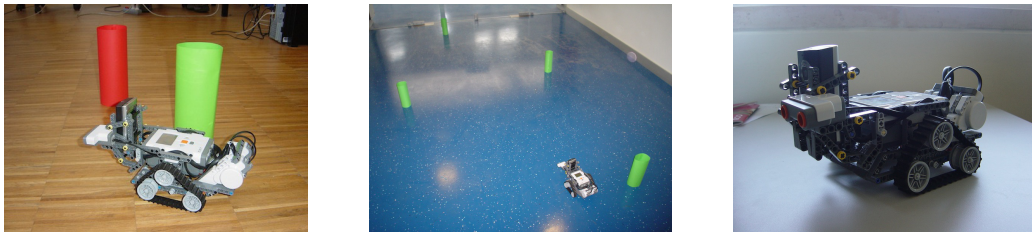
Figure 3 Mobile robot and smartphone system used in the case study application.

The concept of the application is that from continuously captured images obtained from the camera of the smartphone, special landmarks and features can be detected and used to update an internal model that uses such information to both navigate and locate itself in the environment. The application is composed of multiple algorithms, which expose several characteristics and input parameters (e.g., number of samples for computing the location) whose configuration impacts both the output and the processing requirements of the navigation (e.g., execution time, memory consumption). Managing these characteristics allow the use of adaptations for optimizations and for guaranteed continuous execution. As localization is inherently uncertain, it has been addressed with probabilistic methods, namely *particle filters* [8], which is the method used for adaptation in this case study. Additionally, for experimental testing, two setups were used: Setup 1 – a Java ME Platform SDK 3.0 mobile device emulator; and Setup 2 – a Nokia N95 smartphone.

4.1 Particle Filter Algorithm and Adaptation Analysis

The particle filter algorithm is used to track the evolution of the robot's pose (i.e., position and orientation) by building a sample-based representation, which approximately estimates the state of the robot's pose. The set of samples used for estimation are known as particles, and represent at each timestamp a hypothesis of what the true state of the robot's pose might be (an estimation based on simulation). The evaluation of the multiple hypothesis, given from the particles, allows for an overall global estimation of the correct robot's location. Structurally, the algorithm is composed of three main phases: *prediction*, *update* and *resample*; which are executed and looped over time, as the robot moves within its environment. Figure 4 represents a simulation of the estimation model for localization based on the particle filter algorithm, depicting the mobile robot, the particles, and the best particle estimation.
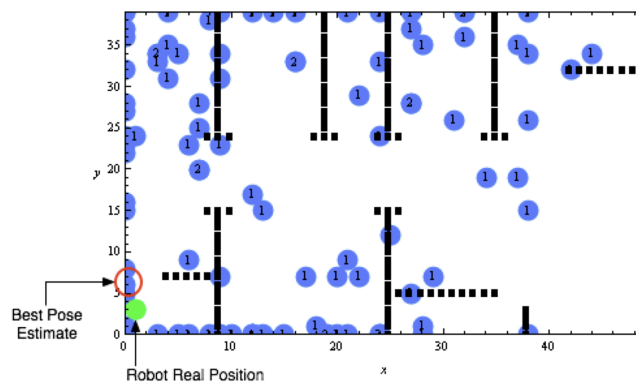
Figure 4 Mobile robot and particle positions and weights after several iterations of the algorithm, depicting also the position of the best particle, which is an estimate of the robot's real position.

The implementation used in this work is based on the approach presented in [16] and is applied for global localization, i.e., identification of the robot's position in an *a priori* known map. In this case study, the environment map is viewed as a 2-dimensional occupancy grid, and accuracy measurements are performed with the euclidean distance between the real robot position and the best overall estimate computed.

Through analysis and experimental testing conducted on the algorithm, for higher accuracy, the number of particles should be as high as possible, limited, however, to the size of the environment and the amount of computational resources available (to improve efficiency). The size of the environment influences the computational complexity of the algorithm, as larger maps require more particles and more movements to produce reasonable results, whilst also inheriting a higher degree of uncertainty in the results produced. Also, the more the robot moves in the environment, the more time will be available for estimation, therefore producing higher certainty in the location estimation, due to continuous refinements.

4.2   Adjusting the Number of Particles

One of the most relevant problems detected with the implementation performed in [17] was the difficulty to define the finite number of particles. When choosing the number of particles to use, it is very important to take into consideration the available computational resources, the time constraint to comply and the navigation requirements. A low number of particles may not be enough to provide a good pose estimate, while a high number may provide a better estimation, but at a much higher computational cost, which may not be feasible. Using a fixed number of particles will neither be effective in terms of accuracy, neither in optimizing the computation of the algorithm in the presence of varying conditions.

Figure 5 shows how different map sizes with different particle numbers influence the euclidean distance of the estimation to the real robot position (i.e., accuracy), execution time, memory used, and power consumed.
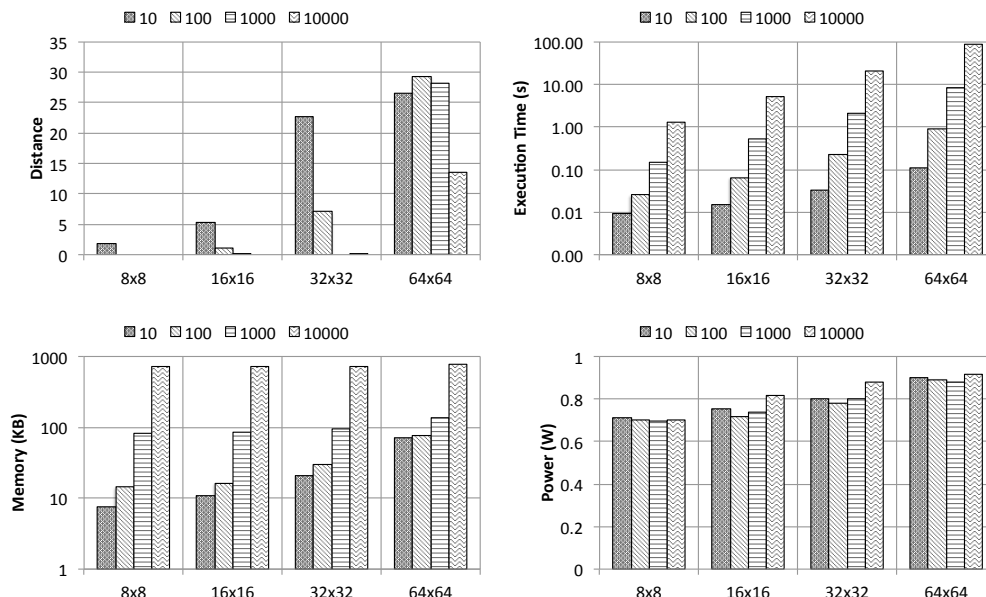
Figure 5 Measurements for distance, execution time, memory (using setup 1), and power (using setup 2) according to different map sizes and number of particles (10, 100, 1000, 10000).

Possible adaptation policies could consider the number of particles as a function of the environment map size, or as a function of the available execution time or free memory for computation. For the same number of particles, as the map size increases, the euclidean distance, execution time, memory and power all increase. For the same map size, increasing the number of particles improves accuracy (i.e., decreases distance), but increases execution time and memory, and to a lesser extent power consumption.

4.2.1 Adjusting to the Map Size

As the localization is accomplished through a map of the environment, in order to save memory and reduce the execution complexity, the entire map might not be completely loaded or only be provided on demand. With each new map, comes the possibility to reconfigure the number of particles, for example, as a function of the map width and height. Experiments conducted using this adaptation policy suggest that adapting the number of particles according to the map size, i.e., $(width \times height)$, $(width \times height)/2$, and $(width \times height)/3$, represent more efficient solutions both in accuracy and in the execution time and memory, than with a fixed number of particles, i.e., 10, 50, 100, 500, 1000, and 5000.

A DSL specification for this adaptation scenario is presented in Listing 2. The specification imports two functions, one encapsulating the particle filter algorithm – `particleFilter` – and another for retrieving the current map size – `getMapSize` (lines 1 and 2, respectively). There is one rule defined – `r_map` – to adjust the number of particles used in the algorithm, according to the map width and height (lines 9–11). Before starting the execution of the particle filter algorithm, the map size is used to recalculate and assign the number of particles to be used, as specified in the operations section (line 5).

■ **Listing 2** DSL specification for the number of particles adaptation considering the map size.

```
1 import function particleFilter(int particles=100);
2 import function [int w, int h] getMapSize();
3
4 operations{
5    r_map evaluate before particleFilter;
6 }
7
8 rules{
9    r_map: every(particleFilter){
10       particleFilter.particles = getMapSize().w x getMapSize().h;
11    }
12 }
```

4.2.2 Adjusting to Computational Constraints

Additionally, new behavior can be added in order to further adapt in accordance to requirements specific to the device energy, execution time or memory conditions. Even using an adaptable number of particles defined as shown in Section 4.2.1, which contributed to better efficiency in the tradeoff between accuracy and computational requirements, the change and variation over time in the device's computational conditions (e.g., device energy dropping below a 20% threshold level) would cause additional difficulties on execution.

Herein, we extend the adaptable behavior presented before with additional reconfigurations that further adjust the number of particles used when the device's energy level is below 20%,

when it is consuming a large amount of memory, and when in violation of an execution time constraint. To demonstrate the adaptable behavior, we designed a scenario of navigation, where the mobile robot explores a territory composed of eight areas (two areas with size 8×8, two with size 16×16, two with size 32×32, and two with size 64×64). Using this adaptation specification, the original algorithm is now equipped with reconfigurations that improve its accuracy in situations where both the computational conditions and maps change.

■ **Listing 3** DSL specification for the adaptation of the number of particles according to map size, device energy level, time, and memory limitations.

```
1  import function particleFilter(int particles=100);
2  import function [int w, int h] getMapSize();
3  import function [int level] getEnergyLevel();
4
5  operations{
6      r_energy evaluation before particleFilter;
7      r_map evaluation before particleFilter;
8      r_time evaluation point "resample";
9      r_memory evaluation point "resample";
10 }
11
12 rules{
13     evaluate: order(r_map, r_energy, r_time, t_memory);
14
15     r_map: every(particleFilter){
16         particleFilter.particles = getMapSize().w x getMapSize().h;
17     }
18     r_energy: every(5sec){
19         if(getEnergyLevel() < 20){ <p_energy_down == 0>
20             particleFilter.particles -= particleFilter.particles / 2;
21             <p_energy_down = 1>
22         }else{ <p_energy_down == 1>
23             particleFilter.particles += particleFilter.particles / 2;
24             <p_energy_down = 0>
25         }
26     }
27     r_time: every(resample){
28         if(particleFilter.elapsed_time >= 20){
29             particleFilter.particles -= particleFilter.particles / 4;
30             <p_time_down = 1>
31         }else{ <p_memory_down == 0 && p_energy_down == 0>
32             particleFilter.particles += particleFilter.particles / 4;
33             <p_time_down = 0>
34         }
35     }
36     r_memory: every(resample){
37         if(particleFilter.memory_consumed >= 3000){
38             particleFilter.particles -= particleFilter.particles / 4;
39             <p_memory_down = 1>
40         }else{ <p_time_down == 0 && p_energy_down == 0>
41             particleFilter.particles += particleFilter.particles / 4;
42             <p_memory_down = 0>
43         }
44     }
45 }
```

From experimental results comparing the adaptable specification for the particle number in contrast with three different fixed values (i.e., 50, 1000, 5000), the average and cumulative distance is lower than in all other fixed particle number tested, meaning that the overall accuracy of the estimation was higher. Regarding execution time and memory, the adaptable version is comparable to the use of 1000 fixed number of particles (on average the adaptable specification used 860 particles), however this is higher than for fixed 50 particles and much lower than for 5000 fixed particles. Also, the energy consumed in the scenario was lower for the adaptable number of particles (74mAh in contrast to 77mAh for fixed 1000 particles).

The advantages of this adaptable behavior and the possibility of further configuration to be more tailored to other operating conditions, stimulate the desire to specify it through our DSL approach. A possible specification for such behavior is presented in Listing 3, that shows that with the additional rules (when in comparison to Listing 2) it is possible to further refine the number of particles used, taking into account how much energy (lines 18–26), time (lines 27–35), and memory (lines 36–44) are available.

4.2.3   Dealing with Conflicts

Considering Listing 3 with four different rules for adaptation, the need for an evaluation order and priority becomes imperative. From the analysis of the rule automata, some transitions were detected as potential conflicting situations. These problematic situations were detected automatically due to the similarity between code actions, i.e., the same algorithm parameter was manipulated and the operation performed is the opposite (addition and subtraction on `particleFilter.particles`). To ensure the correct adaptive behavior, and according to the information perceived from the automata, additional restriction predicates needed to be defined. For example, considering the case when for low energy conditions the number of particles is reduced, it should not be increased if the time constraint was satisfied. The automata-based detection of such a problematic situation is presented in Figure 6.
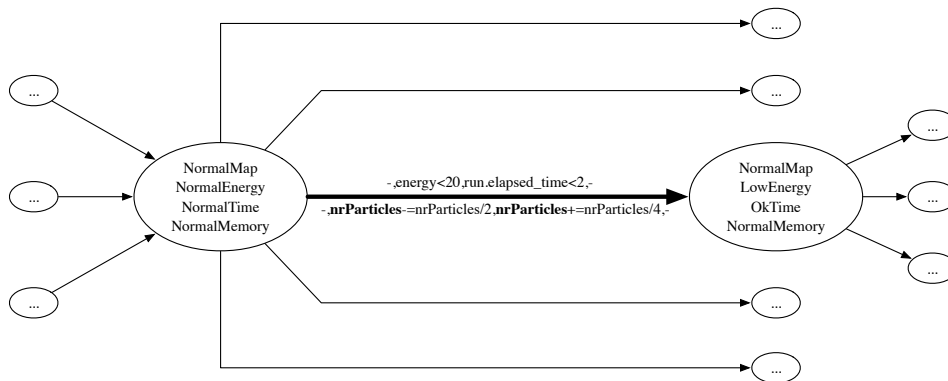
Figure 6 Detection of a conflicting situation through the cartesian product of the individual automata representing the rules. For simplicity, these automata are only an excerpt.

Figure 6 depicts a situation where the conflicting manipulation of the `particleFilter particles` parameter with contradictory operations was detected (transition in bold) in the transition to an adaptable state due to energy becoming low (decreasing particles) and the execution time being satisfied (increasing particles).

For such task, the specification in Listing 3 defines predicates for the execution of the adaptation rules. The predicates define additional evaluation conditions for the adaptation rules. This evaluation prioritization guarantees, in this case, that the `r_map` has no dependency

towards other rules, `r_energy` rules depends only on itself, `r_time` and `r_memory` when in violation can be executed, however, when not in violation, they will only increase the number of particles, if and only if, the other rules have not been executed. Concretely, for example, considering rule `r_time`, when in violation of the time constraint, it sets the predicate `p_time_down` to 1 (an assignment for true → execution occurred) meaning the action for number of particles decrease occurred. If it is not in violation of the time constraint, then an increase of the number of particles could take place, however, in order to be compatible with the other rules, the action only executes if the predicates `p_memory_down` and `p_energy_down` are assigned 0 (meaning that they are set to false and were not executed).

4.3 Discussion

Besides defining the adaptable behavior with the DSL, for comparison we implemented the equivalent behavior inside the application logic, since it is the most common procedure to incorporate adaptations. Data on the comparison between the adaptable behavior specified in a DSL version and in a Java (GPL) version are provided in Figure 7, where case 0 is the default application, case 1 adds the adaptations due to the map size (Section 4.2.1), and case 2 adds the adaptations due to computational constraints (Section 4.2.2). The transition cases 0 → 1 and 1 → 2, focus on the changes from one case to the other.

DSL	1	2	1 → 2		Java	0	1	2	0 → 1		1 → 2	
LoC	13	41	28	215.38%	**LoC**	5229	5236	5325	7	0.13%	89	1.70%
Words	31	95	64	206.45%	**Methods**	305	305	306	0	0.00%	1	0.33%
Imports	2	3	1	50.00%	**Method LoC (avg)**	12.53	12.55	12.65	0.02	0.15%	0.10	0.81%
Operation instructions	2	5	3	150.00%	**Attributes**	131	132	135	1	0.76%	3	2.27%
Rule blocks	1	4	3	300.00%	**McCabe Cyclomatic Complexity (avg)**	2.45	2.45	2.48	0	0.00%	0.03	1.31%
Transition points			3		**Altered Methods**				2		3	
Similarity			15%		**Transition points**				3		2	
					Similarity				56%		56%	

Figure 7 Comparison between the particle filter adaptations specified in the DSL and in Java. LoC of the DSL may slightly vary from the specifications presented due to code formatting.

From the analysis of Figure 7, it is possible to verify how the specification of adaptable behavior reflects itself both in the DSL and in the GPL. The main observations that can be drawn from this comparison are:

- Average textual similarity for the Java versions was conducted solely on the modified methods. The unaltered code was not considered as its large size (due to the rest of the application) would overwhelm the similarity results.
- Transition points for the DSL consist in an added function, and rule connection points and their corresponding actions. In Java, they consist mostly on method changes and new methods; and most importantly on different files (two to three separate files).
- Regarding lines of code, in absolute values the DSL increase from case 1 → 2 is three times less than in the GPL. Of course, due to the size of the Java code, i.e., many classes and packages in comparison to one DSL specification file, in relative percentage terms

the DSL increase in lines of code is much higher.

- In the DSL all the modifications necessary to be made in order to transform the specification from case $1 \to 2$ were performed solely in one specification file. In contrast, in the Java version, three different classes had to be modified. Also, the Java code introduced for adaptations is intertwined with the application logic.
- Due to the evaluation and action locations of the adaptation rules, in the DSL the modifications were confined to the rule block section, while in the Java version, the adaptable behavior rules where placed at two different code locations.
- The use of predicates allows the correct prioritization and also the conflict avoidance between the different rule actions. In Java, the predicates required added control variables, branches and overall more confusing coding. The predicates were also embedded within some instructions, becoming intrusive to already defined statements of code.
- The rule relative to the adjustment of particle number in consequence of the map size was defined in the Java code in a different location than the other rules. In the DSL all rule behavior is defined in one location.

It is important to mention the minimal additional effort that was required to perform minor modifications to the original application, in order to prepare the application for the weaving of the generated adaptable behavior code specified with the DSL.

5   Related Work

Adaptation in software applications has commonly been accomplished through the use of conditional expressions, parameterization, and exceptions [4]. In today's dynamic computational environments and requirements, autonomic computing and true adaptive behavior cannot simply be accomplished through such methods, as they are error-prone and introduce complexity by intertwining adaptation and application behaviors, scaling poorly and thus rendering software evolution and maintenance hard. To this end, several approaches have been proposed to support software adaptations, such as (i) frameworks and architectures, (ii) context-oriented programming, and (iii) dynamic aspect-oriented programming. Other approaches such as feature-oriented programming and change-oriented software engineering are also of relevance. Our approach differs as we focus on a completely independent domain language, which does not extend nor is tailored specifically to another host language, platform or environment, offering more flexibility, structure and comprehension.

Framework and architectural models   These offer dynamic adaptation infrastructures, however with no wide adoption, possibly due to limited adaptation support or due to effectively low adaptation facilities in practice [14]. Some examples of such architectural approaches include MADAM [4] and Rainbow [6]. Furthermore, these approaches require software to be developed according to new and complex component-based architectures which are not ideal solutions to already developed and deployed applications.

Context-Oriented Programming (COP)   These concepts have commonly been implemented as extensions to several languages [1] (e.g., Subjective-C [7]). However, each language extension comes with its own approach to the COP paradigm and implementations commonly suffer from execution overhead [1]. The objective of COP languages is to modify the behavior of a program by associating code definitions with context-related layers that are activated or deactivated according to the current context. The behavior of objects and methods thus depends on the context in which they execute. Autonomic behavior can be accomplished by

executing code variations in reaction to changes in context states. Although adaptations are performed in applications, the adaptation specification distinguishes from our approach as it depends, and is tailored, solely to context states. Also, the context-based adaptations are normally embedded within the application code itself.

Dynamic Aspect-Oriented Programming (AOP) Aims at improving the separation of concerns and thus can be used to encapsulate the adaptations that are required to implement an autonomic system [9]. With this technique, an adaptation can be created using aspects, and woven statically or dynamically, providing an extremely powerful tool to allow the application to be modified [15] (e.g., AspectJ [11]). The AOP concerns are similar to our implementation approach, as aspects alter the behavior of the application code by inserting additional concerns, which can be adaptation-related, at various points in a program (similar to our evaluation points). In fact, an aspect-oriented approach can be used in our work to implement the necessary modifications at the application code level. However, our approach differs from traditional AOP as we focus on specifying adaptation both at the code level and at the design level, with constructs tailored specifically to the adaptation domain in order to define relations between adaptations, the periodic evaluation for adaptations triggering, etc.

6 Conclusions

In this paper we proposed a DSL-based approach to specify adaptable behavior in embedded applications in a flexible way and externally to the main application logic. Using this DSL-based approach, adaptation strategies can be specified and modified without tampering with the application code. Furthermore, different strategies defined in the DSL can be shared and deployed over different platforms and programming languages, promoting fast prototyping. We illustrated the application of the proposed approach in the case study of a mobile robot with a navigation application running on a smartphone. With the DSL, it was possible to easily and flexibly specify the adaptable behavior of that application. The experimental results highlight the benefits of specifying the adaptable behavior through the DSL-based approach, when compared to implementing the same behavior by re-programming directly the application. We also showed how adaptation strategies are specified, and evaluated the impact of modifying and extending an existing strategy with new rules. In short, we demonstrated that not only is the DSL code more succinct, but changes and improvements are also easier to implement.

─── **References** ───

1   Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-Oriented Programming Languages. In *Proc. of the Int'l Workshop on Context-Oriented Programming (COP'09–ECOOP'09)*, pages 6:1–6:6. ACM, 2009.

2   Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A Survey on Context-Aware Systems. *Int'l Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

3   Davide Figo, Pedro C. Diniz, Diogo R. Ferreira, and João M. P. Cardoso. Preprocessing Techniques for Context Recognition from Accelerometer Data. *Personal Ubiquitous Computing*, 14(7):645–662, 2010.

4   J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

5   M. Fowler. *Domain-Specific Languages.* Addison-Wesley. Pearson Education, 2010.

6   D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

7   Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective–C: Bringing Context to Mobile Platform Programming. In *Proc. of the 3rd Int'l Conf. on Software Language Engineering (SLE'10)*, volume 6563 of *LNCS*, pages 246–265. Springer, 2010.

8   N.J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation. *Proc. of the IEEE Radar and Signal Processing*, 140(2):107–113, 1993.

9   Philip Greenwood and Lynne Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. *Proc. of the 2004 Dynamic Aspects Workshop (DAW'04), RIACS*, pages 76–88, 2003.

10 John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

11 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *LNCS*, pages 327–354. Springer, 2001.

12 Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37:316–344, December 2005.

13 M. Mikalsen, J. Floch, N. Paspallis, G.A. Papadopoulos, and P.A. Ruiz. Putting Context in Context: The Role and Design of Context Management in a Mobility and Adaptation Enabling Middleware. In *Proc. of the 7th Int'l Conf. on Mobile Data Management (MDM'06)*, pages 76–83, 2006.

14 Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th Int'l Conf. on Software Engineering (ICSE'08)*, pages 899–910. ACM, 2008.

15 Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *Proc. of the 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147. ACM, 2002.

16 Ioannis Rekleitis. *Cooperative Localization and Multi-Robot Exploration.* PhD thesis, School of Computer Science, McGill University, Montréal, 2003.

17 André C. Santos. Autonomous Mobile Robot Navigation using Smartphones. Master's thesis, Instituto Superior Técnico – Technical University of Lisbon, 2008.

18 André C. Santos, Pedro C. Diniz, João M. P. Cardoso, and Diogo R. Ferreira. A Domain-Specific Language for the Specification of Adaptable Context Inference. In *Proc. of the IEEE/IFIP Int'l Conf. on Embedded and Ubiquitous Computing (EUC'11)*, pages 268–273. IEEE Computer Society, 2011.

19 Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving Energy Efficiency of Location Sensing on Smartphones. In *Proc. of the 8th Int'l Conf. on Mobile Systems, Applications, and Services (MobiSys'10)*, pages 315–330. ACM, 2010.