Specifying Dynamic Adaptations for Embedded Applications Using a DSL

André C. Santos, João M. P. Cardoso, Pedro C. Diniz, Diogo R. Ferreira, Zlatko Petrov

***Abstract*—Embedded systems are severely resource constrained and thus can benefit from adaptations to enhance their functionality in highly dynamic operating conditions. Adaptations, however, often require additional programming effort or complex architectural solutions, resulting in long design cycles, troublesome maintenance, and impractical use for legacy applications. In this paper, we introduce an adaptation logic for the dynamic reconfiguration of embedded applications and its implementation via a domain-specific language. We illustrate the approach in a real-world case study of a navigation application for avionics.**

***Index Terms*—Embedded systems, software adaptation, application reconfiguration, domain-specific language, avionics.**

I. Introduction

THE reliance on mobile embedded systems demands these devices to be increasingly powerful while meeting stringent energy and performance requirements. As a result, their software components must be adaptable to a wide range of very dynamic (run-time) scenarios in order to continue to fulfill their set of requirements, e.g., continuously delivering acceptable service levels [1]. While in some cases their run-time adaptation can consist of simple algorithmic parameter tuning, in more sophisticated scenarios, adaptations can require a substantial transformation of the underlying code structure. As a result, adaptations inevitably grow in complexity and thus in terms of implementation and development cost.

In this work, we introduce an approach for dynamic reconfiguration of embedded applications, through an external, high-level and platform-neutral, domain-specific language (DSL). The DSL separates the adaptation logic from the core application logic by enabling the specification of adaptation policies that produce the necessary adaptability at run-time. The adaptation logic specified in the DSL is then incorporated into the application through code generation and weaving.

To illustrate the proposed approach, we describe and discuss its use in a case study involving an industry-developed stereo navigation application – StereoNav (described in detail in [2]). In essence, the application takes two images as input and extracts special features that are used for pose estimation, thus enabling both avionics localization and navigation.

The remainder of this paper is organized as follows. Section II and III present the adaptation logic and DSL implementation. Section IV describes the application of our approach to the case study. Section V presents related work, and Section VI summarizes and concludes the paper.

André C. Santos and Diogo R. Ferreira are with the Department of Computer Science and Engineering, IST – Technical University of Lisbon, Portugal, e-mail: acoelhosantos@tecnico.ulisboa.pt. João M. P. Cardoso is with the Department of Informatics Engineering at FEUP – University of Porto, Portugal. Pedro C. Diniz is with INESC–ID, Portugal. Zlatko Petrov is with Honeywell International s.r.o., Czech Republic.

Manuscript received MONTH DAY, YEAR; revised MONTH DAY, YEAR.

II. Enabling Application Adaptability

In order to model and implement run-time adaptability, the application logic and the adaptation logic are defined separately and later combined for execution. This separation of concerns is essential, as the application logic focuses on the core functionality, while the adaptation logic adds a layer of application configurability. Furthermore, such separation is essential for behavior understanding, rapid prototyping, flexible maintenance and conceptual independence.

Regarding the core application logic, each application is developed with its own goals and implementation details. However, all share common characteristics such as a sequence of computational steps, a set of algorithms, input parameters, and output values. Some of these characteristics allow adaptations to reconfigure the application behavior by changing parameters, activating or deactivating components, inserting new control logic, etc. Having these application characteristics clearly identified and exposed is essential for the purpose of implementing the adaptation logic.

The adaptation logic is represented as a policy that defines strategies that target adaptation goals through a set of *adaptation rules*. Each rule is triggered when specific conditions occur and as a consequence applies the appropriate actions. Since the adaptation policy is external to the application, it can be applicable to multiple applications, thus promoting reusability in the specification of adaptable behavior. On the other hand, each application may have multiple adaptation strategies for different concerns.

To specify an independent logic for adaptation, an approach that operates at a different level of abstraction than the application logic is necessary. For this purpose, we introduce a novel DSL with platform-independent and high-level abstractions tailored to the specification of adaptable behavior for embedded systems, namely: (i) activate/deactivate specific sections of application code to enable/disable certain computational steps; (ii) change function parameters in order to configure the behavior of certain algorithms within the application code; (iii) establish the frequency of function execution and thus speed up or speed down recurrent computations defined in the application code. With specific constructs, the proposed DSL can thus be more succinct and intuitive than writing adaptation code in a general-purpose programming language, such as C.

Figure 1 illustrates the flow and relations between the aforementioned entities, depicting the application logic as

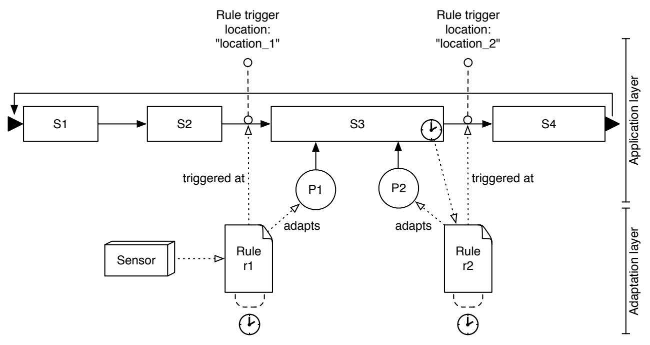

Fig. 1. Specifying adaptation behavior considering two separate logic layers and depicting adaptation rules and their triggering locations. Computing stages are designated as `S`, and their input parameters as `P`.

```
 1 strategy stereoNavImgResAdapt1{
 2   int[][] imgRes = {{320, 240}, {640, 480}};
 3   import function stereoNav(int ransacIterations=5000,
         int imgWidthRes=640, int imgHeightRes=480);
 4   import function [int vehicleSpeed] getVehicleSpeed();
 5   operations{
 6     r3 evaluation point "e1";
 7   }
 8   rules{
 9     r3: every(stereoNav):
10     retrieve getVehicleSpeed().vehicleSpeed as speed{
11       if(speed > 50){
12         stereoNav.imgWidthRes = imgRes[0][0];
13         stereoNav.imgHeightRes = imgRes[0][1];
14       }else{
15         stereoNav.imgWidthRes = imgRes[1][0];
16         stereoNav.imgHeightRes = imgRes[1][1];
17       }
18     }
19   }
20 }
```

Fig. 2. DSL specification code for adapting image resolution according to the vehicle's current speed in the StereoNav application.

a process, and highlighting a segment that is reconfigured through an adaptation policy composed of two adaptation rules that modify the input parameters for the `S3` computing stage, before execution due to sensor data (`r1`), and after execution due to information from the execution time (`r2`).

With respect to the implementation, the fact that the DSL is independent of the application code allows for adaptations to be coupled with the application through several possible compile- or run-time mechanisms, such as: (i) injecting adaptation code in the application at a bytecode level; (ii) weaving adaptation code directly into the application code; (iii) executing the application in a controlled virtual execution environment. In the present approach, we apply a joint-compilation process, where initially the DSL code is translated into the target GPL of the application using a custom compiler; secondly, the translated adaptation code is weaved statically into the application source code; thirdly, the combined adaptation and application code is compiled using a standard compiler, generating the adaptive application.

III. Specifying Adaptability Using a DSL

In the DSL, an adaptation policy is specified by *strategy* entities, composed of *declarations*, *operations*, and *rules*. *Declarations* are reserved for static information (e.g., variables, default values, function imports). *Operations* specify the evaluation/action points where the adaptation rules will be evaluated and the actions triggered in the application code (e.g., before or after a code function, or at a specific annotated code point). The *rules* section specifies the activating conditions and the adaptation actions that reconfigure the application. Additionally, for extensibility, a supplementary *code* section may be used for platform-specific code.

An example of an adaptation policy specified with the DSL is shown in Figure 2. This is a strategy for the StereoNav application where an image resolution input parameter is adjusted according to the state of a vehicle speed variable. Lines 2–4 refer to declarations, specifying an array of image resolutions to be used (line 2), and two imported functions representing an iteration of the StereoNav algorithm (line 3) and the retrieval of the vehicle speed (line 4). Lines 5–7 define the location `e1` for rule execution (line 6), which points to a location in the application code. In lines 8–19, the rule `r3` specifies a periodic adjustment of image resolution parameter depending on the vehicle speed.

In order to detect possible rule conflicts, we use a verification process based on automata theory. This approach allows the modeling of the *rules* by translating them into automata and then, through automata operations, characteristics of the adaptations can be determined, namely: (i) the cartesian product obtains the combination of all possible adaptation states and transitions; (ii) minimization identifies and removes useless or unreachable states; and (iii) intersection detects common states. Modeling and translating the concepts of an adaptation rule to an automaton is performed through an algorithm, which analyzes the code control flow of the rule and extracts possible adaptation states and the transitions between them. Using the automata model, it is also possible to simulate the adaptation behaviors admissible with the defined rule set.

IV. Case Study: Stereo Navigation Application

The StereoNav application comprises a sequence of algorithmic operations: *debayering*, *rectification*, *feature extraction*, *feature matching*, *3D reprojection*, *pose estimation*, and *refinement*. Throughout these operations, StereoNav requires correct and timely navigation information to maintain an acceptable level of quality-of-service (QoS), and to do so it must adapt to changes in vehicle speed, limited time for computation, and availability of computational resources. Our run-time adaptation approach is a useful instrument not only to prevent QoS degradation but also to improve QoS and thus the navigation itself. Moreover, the approach allows the adaptations to be untangled from the core application logic.

A. Experimental Setup and Methodology

In this case study, the most computationally demanding operations are the most relevant to target, as their reconfiguration will have a greater impact in the overall behavior of the application. *Feature extraction* and *pose estimation* are

```
r4: every(stereoNav):
retrieve getVehicleSpeed().vehicleSpeed as speed{
  if(speed < 25){
    stereoNav.ransacIterations = 5800;
  }else if(speed >= 25 && speed < 50){
    stereoNav.ransacIterations = 5500;
  }else if(speed >= 50 && speed < 75){
    stereoNav.ransacIterations = 4800;
  }else if(speed >= 75 && speed < 100){
    stereoNav.ransacIterations = 4300;
  }else if(speed >= 100){
    stereoNav.ransacIterations = 3900;
  }
}
```

Fig. 3. Rule specification code for adapting the number of *RANSAC* iterations according to the vehicle's speed.

```
r5: every(stereoNav):
retrieve getTimeConstraint().time as timeCons{
  if(timeCons >= stereoNav.elapsed_time){
    stereoNav.imgWidthRes = imgRes[0][0];
    stereoNav.imgHeightRes = imgRes[0][1];
  }else{
    stereoNav.imgWidthRes = imgRes[1][0];
    stereoNav.imgHeightRes = imgRes[1][1];
  }
}
```

Fig. 4. Rule specification code for adapting image resolution according to a time constraint for the computation time.

the most demanding operations (89.3% and 5.6% of total execution time, respectively), and they expose two possible parameters for adaptation that yield a significant computational impact: (i) the resolution of the processed images in the *feature extraction* operation, and (ii) the number of iterations of the *RANdom SAmple Consensus (RANSAC)* estimation algorithm in the *pose estimation* operation. Herein, we are interested in defining adaptations, specifying them with the DSL, and evaluating the benefits of our approach.

For these experiments, we used a C implementation of the StereoNav algorithm and two testing setups: a PC environment (setup 1); and an FPGA board environment (setup 2). For the *feature extraction* operation, we considered two image resolutions of 640×480 (high resolution) and 320×240 (low resolution). Regarding the number of *RANSAC* iterations for the *pose estimation* operation, we introduced configurations guaranteeing 90%, 92%, 94%, 96%, and 97% probability of correctness, corresponding to 3900, 4300, 4800, 5500, and 5800 iterations, respectively. The execution time is on average $3\times$ to $4\times$ greater for the higher resolution, and roughly increases linearly with the number of iterations.

B. Strategy: Adapting to the Vehicle Speed

Decreasing the image resolution as the vehicle speed increases allows continuous navigation information since, the faster the vehicle moves, the faster the application needs to determine its position. The DSL specification for this strategy was presented earlier in Section I and specified in Figure 2.

In addition, as the speed of the vehicle increases, the number of *RANSAC* iterations must be reduced in order to decrease the computational strain and thus the execution time. The DSL specification strategy for this adaptation is much similar to the earlier specification in Figure 2, with the exception of the `rules` section, which now targets other adjustments. Figure 3 depicts the new rule `r4` evaluated at the beginning of every `stereoNav` execution, retrieving the vehicle speed through the imported function and, depending on its value, setting the number of iterations for the *RANSAC* algorithm. In this example, the speed thresholds are merely illustrative as they may change with the characteristics of the vehicle.

C. Strategy: Adapting to a Time Constraint

Considering a requirement for the algorithm to complete its execution within a required time frame, the adaptation of image resolution can be implemented in the following way: if the time constraint is violated, then the image resolution is decreased; if the time constraint is satisfied, then the image resolution can be increased. Such strategy is similar to the specification in Figure 2, however the vehicle speed function is replaced by a function that informs on the time constraint (`getTimeConstraint`) and a new rule is added, based on the elapsed time of the last `stereoNav` execution, which can be obtained via an `elapsed_time` macro (code for this macro is added when weaving). The rule retrieves the time constraint, and if the constraint is greater than or equal to the execution time of the previous StereoNav step, then the StereoNav algorithm uses the high image resolution, otherwise it switches to the low resolution (see Figure 4).

A second adaptation for time frame compliance consists in selecting the number of iterations for the *RANSAC* algorithm according to the time available for the *pose estimation* operation, which is possible to do with prior knowledge about the average execution time of that operation. Given an available execution time, it is possible to select the highest number of *RANSAC* iterations that perform within the time constraint. The implementation of this strategy is presented in Figure 5, where two functions are imported (lines 2–3) to provide the control variables needed to choose the appropriate number of iterations (total available time and the time already spent up to the *pose estimation* operation). Lines 5–20 set the appropriate number of iterations according to the available time for the *pose estimation* operation. Some lines are omitted for simplicity as they are equal to the specification in Figure 2.

D. Discussion

The practical experience presented here reveals that the DSL allows for the easy decoupling between application and adaptation-related code, as well as maintenance and modification over time. Each adaptation was specified separately to show the impact of different requirements, which need changes mostly within rules to include addition decision criteria. All strategies allow to keep the navigation accurate even in the presence of problematic situations.

As the DSL adaptation code is compiled and weaved within the application source code, having the adaptations directly implemented into the application code would require a substantial coding effort by the addition of functions, variables, threading, etc. Due to the interplay with the main application logic, such programming effort would involve much more than just coding the adaptation code. More specifically, the coding

```
// (...)
import function [double tPrev] getTimeSpent();
import function [double tTotal] getTimeConstraint();
// (...)
rules{
  r6: every(stereoNav):
  retrieve (getTimeConstraint().tTotal - getTimeSpent().
      tPrev) as tAvailable{
    if(tAvailable > 0.08){
      stereoNav.ransacIterations = 5800;
    }else if(tAvailable > 0.075){
      stereoNav.ransacIterations = 5500;
    }else if(tAvailable > 0.06){
      stereoNav.ransacIterations = 4800;
    }else if(tAvailable > 0.05){
      stereoNav.ransacIterations = 4300;
    }else{
      stereoNav.ransacIterations = 3900;
    }
  }
}
```

Fig. 5. DSL specification code for adapting the number of *RANSAC* iterations according to a time constraint for the overall computation time.

effort measures to the addition of 586 lines of code, 372 statements, and 0.6% additional branches, which is 3× more code than the equivalent DSL code, with a higher word-per-line ratio and thus higher textual density.

The experiment reported here therefore suggests that the manipulation of strategies using the DSL is thus less complex and translates into easier readability and maintenance than the direct modification of the original application code. Consequently, adding adaptive behavior through the DSL requires less code additions, modifications and removals, as changes are confined to fewer locations.

V. Related Work

Embedded systems are gaining momentum due to active research in topics such as context-aware and ubiquitous computing (e.g., [3], [4]). In these types of systems, run-time adaptations are required to achieve performance goals under changing operating conditions. These adaptations are typically accomplished through the use of conditional expressions, parameterization, and/or exceptions [1]. However, implementing adaptations in such a way is error-prone and introduces an undesired degree of complexity by intertwining adaptation and application code, which scales poorly and renders software evolution and maintenance hard. More sophisticated solutions, namely through architectural approaches and programming language extensions, have been proposed to mitigate these problems and better support adaptations.

Architectural-based models offer dynamic adaptations but are not widely adopted (e.g., [1], [5]), possibly due to the limited scope of the supported adaptations, or due to the lack of adaptation facilities in practice [6]. Also, these approaches require applications to be developed according to certain component-based structures, which complicate introducing adaptations into already existing applications.

Programming-based solutions offer adaptations focused usually on extensions to host languages. Context-oriented programming has been typically implemented as an add-on to several languages (e.g., [7]) to modify the behavior of an application by associating code definitions with context-related layers that are activated or deactivated according to the current context. However, adaptations depend on context states alone, and the end result is similar to embedding conditional statements in the application code. Aspect-oriented programming fosters a separation of concerns by encapsulating crosscutting concerns (e.g., [8]). The run-time adaptations addressed in this work could be regarded as being one of such concerns, and possibly they could be woven statically or dynamically into the application. However, in our approach we can take advantage of constructs that are tailored specifically for adaptation, and at a design level that is above the application code.

As DSLs are tailored to specific domains, they are able to offer substantial support for adaptation expressiveness and ease of use, and thus have been also applied to particular adaptation and control specifications for software systems (e.g., [7], [9]).

VI. Conclusion

In this article we proposed an approach for run-time adaptability in embedded applications, which usually operate under constrained conditions. Our approach is implemented through a DSL whose purpose is to specify the adaptable behavior at a higher-level of abstraction. In this approach, adaptations are decoupled from the main application logic, allowing for better software design, prototyping and maintenance. For validation, we presented a case study that highlights the advantages of adaptations in embedded applications, as well as the benefits of using the DSL-based approach to define them.

Acknowledgment

This work was partially supported by *Fundação para a Ciência e a Tecnologia (FCT)* under grant SFRH/BD/47409/2008.

References

[1] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven, “Using Architecture Models for Runtime Adaptability,” *IEEE Software*, vol. 23, no. 2, pp. 62–70, 2006.

[2] REFLECT Consortium, “Rendering FPGAs to Multi-Core Embedded Computing (REFLECT) – Technical Report about Application Requirements for Reconfigurability and Hardware Templates,” Deliverable D1.5 – FP7 THEME ICT-2009-4, Tech. Rep., 2009.

[3] T. van Kasteren, A. Noulas, G. Englebienne, and B. Kröse, “Accurate Activity Recognition in a Home Setting,” in *Proceedings of the 10th International Conference on Ubiquitous Computing (UbiComp’08)*. ACM, 2008, pp. 1–9.

[4] M. Baldauf, S. Dustdar, and F. Rosenberg, “A Survey on Context-Aware Systems,” *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2, no. 4, pp. 263–277, 2007.

[5] IBM, “An Architectural Blueprint for Autonomic Computing,” IBM, Tech. Rep., 2003.

[6] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Runtime Software Adaptation: Framework, Approaches, and Styles,” in *Proceedings of the 30th International Conference on Software Engineering (ICSE’08)*. ACM, 2008, pp. 899–910.

[7] T. Kamina, T. Aotani, and H. Masuhara, “EventCJ: A Context-Oriented Programming Language with Declarative Event-based Context Transition,” in *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD’11)*. ACM, 2011, pp. 253–264.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “An Overview of AspectJ,” in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP’01)*, ser. LNCS. Springer, 2001, vol. 2072, pp. 327–354.

[9] S. Aboubekr, G. Delaval, and E. Rutten, “A Programming Language for Adaptation Control: Case Study,” *SIGBED Review*, vol. 6, no. 3, pp. 11:1–11:5, 2009.