Noname manuscript No.
(will be inserted by the editor)

A DSL for Specifying Run-time Adaptations for Embedded Systems

An Application to Vehicle Stereo Navigation

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

Received: date / Accepted: date

Abstract The traditional approach for specifying adaptive behavior in embedded applications requires developers to engage in error-prone programming tasks. This results in long design cycles and in the inherent inability to explore and evaluate a wide variety of alternative adaptation behaviors, critical for systems exposed to dynamic operational and situational environments. In this paper, we introduce a domain-specific language (DSL) for specifying and implementing run-time adaptable application behavior. We illustrate our approach using a real-life stereo navigation application as a case study, highlighting the impact and benefits of dynamically adapting algorithm parameters. The experiments reveal our approach effective, as such run-time adaptations are easily specified in a higher-level by the DSL and thus at a lower programming effort than when using a general-purpose language such as C.

Keywords Embedded systems · Domain-specific languages · Run-time adaptations · Adaptable behavior · Stereo navigation

1 Introduction

Embedded applications operate in resource-constrained environments subject to constantly changing operational situations. These characteristics are challenging for developing such applications, since the volatility often causes a

André C. Santos (✉), Diogo R. Ferreira
Technical University of Lisbon, Instituto Superior Técnico (IST), Portugal
E-mail: acoelhosantos@ist.utl.pt

João M. P. Cardoso
University of Porto, Faculty of Eng. (FEUP), Informatics Eng. Department, Porto, Portugal

Pedro C. Diniz
INESC–ID, Lisbon, Portugal

Zlatko Petrov
Honeywell International s.r.o., Czech Republic

decrease in performance and an increase in computational cost. Furthermore, embedded applications embody requirements such as reliability, maintainability, availability, security, run-time performance, and energy efficiency, which convey further development challenges [27].

Nowadays, it is becoming highly desirable to design software applications that are adaptable at run-time, thus increasing application operational and situational awareness. Adaptations can take various forms, namely: (i) at the algorithmic level, where one can make use of different processing algorithms or changing algorithm parameters (e.g., [14], [36]); (ii) at a system level, by simply changing the period at which some computations are performed, or by relying on alternate resources that provide equivalent information (e.g., [41]). Adaptability can therefore be used and even required to keep applications running, despite the changes in operational situation (e.g., loss of sensor connection) or the presence of specific requirements (e.g., execution deadline).

Developing and maintaining adaptive applications, however, is a very challenging and error-prone process, as implementing dynamic behavior in an application often requires low-level cumbersome programming tasks, or complex architectural application restructuring. Moreover, frequently, functional application logic and adaptation behavior get mixed, which is potentially risky due to the high degree of complexity introduced by the intertwining of the application and adaptation behaviors [30]. Furthermore, mixing two logics is also a troublesome development process and an overall bad practice. To address this challenge, we introduce a software architecture that allows applications to adapt at run-time, by having their adaptable behavior defined in an external, high-level and platform-independent domain-specific language (DSL). The DSL allows the specification of adaptation strategies, defined in terms of rules that produce the required application adaptability, thus avoiding these aspects from becoming intertwined within the application logic. Our approach has been in continuous development, improving the DSL specification, support infrastructure, and its real-world applicability [36, 37].

To highlight the benefits of the proposed approach, we describe and evaluate its use in the context of a case study based on an industry-developed application for stereo navigation (*StereoNav*) [32]. *StereoNav* consists of an embedded sub-system responsible for vehicle localization in cases where the vehicle's main satellite navigation system has failed or is temporarily unavailable and the vehicle has to localize itself through other methods during a period of time. The *StereoNav* application is complex, executes in an embedded environment, and includes a processing algorithm composed of several parameterizable operations, so it becomes an ideal case study for the use of our DSL-based approach to specify the adaptation behavior. By adjusting the parameters that influence computational cost (e.g., execution time), we ensure a dynamic, real-time compliance with requirements defined by the industry developer (e.g., quality-of-service).

The remainder of this paper is structured as follows. Section 2 describes the approach and its characteristics. Section 3 overviews the implementation toolchain for the approach. Section 4 presents the case study and the experimental evaluation conducted. Section 5 provides an overview of related work. Finally, Section 6 concludes the paper and suggests future work.

2 A DSL-based Approach for Adaptation Specification

General-purpose languages (GPLs), such as object-oriented languages like Java or C++, are used to program nearly any application or system, addressing potentially any problem that needs to be tackled. However, as more complicated problems arise, their complexity spawns the need for more concrete, domain-tailored programming solutions that solve the problem more efficiently. To this end, generally, DSLs allow the concise description of a domain logic reducing the semantic distance between the problem and the programmed solution [7, 39]. Deursen et al. [12] defined a DSL formally as: *"a programming or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain"*.

The use of DSLs is justified by numerous advantages, namely to enhance productivity, reliability and maintainability, since they are more concise and thus written more quickly and therefore easier to maintain; and also to allow easier reasoning and validation since they provide the notation to express the semantics of a domain. In addition, there are other relevant general benefits to the use of DSLs [38]: (i) concrete expression of domain knowledge as DSLs are tailored towards a narrow, specific domain and thus are designed to provide the exact formalisms suitable for that domain; (ii) possible direct involvement of domain experts, often non-programmers; (iii) modest implementation cost as DSLs are typically implemented by a translator that transforms the DSL code into other compatible code; (iv) reliability and correctness that is easily verified. As disadvantages, the generation of a new language for every domain can have potential high startup costs due to design, implementation and documentation. Also, initial tool limitation, lesser trained programmers and additional required mechanisms for integration slow language ramp-up. There is also the difficulty of balancing between domain-specificity and GPL programming constructs. However, on the long run, DSLs payoff.

Considering both advantages and disadvantages mentioned, the bottom line is that DSLs offer substantial gains in expressiveness and ease of use compared with GPLs in their domain of application, since they provide a notation close to an application domain, and is based only on the concepts and features of that domain [29]. As such, a DSL is a means of describing and generating members of a program family within a given problem domain, without the need for extensive knowledge about general programming, and thus raise the level of abstraction beyond coding and consequently make development faster and easier [35]. Moreover, DSLs allow independence from the implementation platform, thus the need for specific knowledge of each platform is greatly avoided, as well as the intervention of an expert in the technology [35].

2.1 DSL-based Approach

Our approach defines an adaptation logic that is external to the application, and takes the form of an adaptation policy entity composed of strategies that can target specific adaptation concerns (see Figure 1). Our implementation for the adaptation policy entity is accomplished with a DSL, tailored specifically for defining adaptation strategies in an independent, scalable, flexible and task-specific way; namely for: (i) activating and deactivating specific sections of code to enable/disable computational steps, (ii) changing function parameters to reconfigure algorithms, (iii) modifying the frequency of function execution.

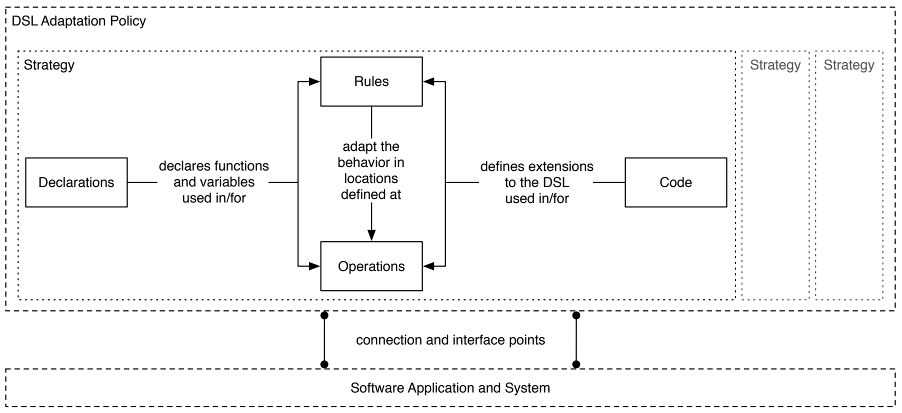

DSL Adaptation Policy
Strategy
Rules
Declarations
declares functions and variables used in/for
adapt the behavior in locations defined at
defines extensions to the DSL used in/for
Code
Operations
Strategy
Strategy
connection and interface points
Software Application and System

Fig. 1 Overview of an adaptation policy structure for a softwares adaptable behavior.

The proposed DSL embodies the adaptation-related concerns as it defines a set of high-level abstractions for looping and for periodic tasks, algorithm parameter changes, testing of conditions, and rule declarations. Within the adaptation domain, the proposed DSL is thus more succinct and its notation more intuitive than using code written in a GPL, such as C. Also, a proper specification of adaptation behavior provides a more powerful mechanism to define different configurations and their triggering conditions, as opposed to using external libraries or APIs. Being high-level and with domain abstractions also allows a wide applicability to most general applications. Consequently, the behavior that is specified through the DSL allows rapid prototyping of adaptable processes, flexible behavior management, and a clear evaluation of possible conflicting adaptations. In addition, we believe that the use our DSL-based approach provides:

- An easy and non-intrusive way to express adaptability behavior due to the domain-specific language constructs. Abstractions, such as the ones related to periodicity and execution rates, concentrate in simple DSL constructs complex behavior and avoid the cumbersome code needed in the final implementation using the target programming language (e.g., C or Java).

– An easy way for exploring different adaptability rules during the development of the solution (Section 4 shows experiments when developing rules incrementally). This is an important aspect as it may reduce the development time considerably considering that in a traditional approach, modifying rules usually requires changes to the application code.
– An easy way to verify adaptability behavior as the DSL may also be used to insert code for monitoring and debugging (Section 4.7 shows an example). The separation of concerns and the rule-based approach provided by our DSL makes easier a verification process. For example, we show in Section 3.2 how to use automata to verify possible rule conflicts during static analysis. In fact due to the extraction of finite automata from the rules expressed using the DSL, users may consider to insert code that follows the automata states and compares those states to the rules applied during runtime. This brings an additional verification process during execution.
– Additional support when mapping adaptation-related computations to the target architecture. The separation of concerns provided by our approach allows mapping tools to decide about the use for different target architecture cores, responsible to execute the adaptive behavior (i.e., mapping it to the same core of the application or to a different core). This is especially important in the presence of complex, computational intensive, adaptive behavior as using a specific core makes the execution of the adaptation-related computations concurrent (Section 4.7 shows an example using an architecture with two cores).
– A specification of adaptability requirements that can provide a more formal notation when eliciting non-functional requirements. Also, analysis specific to the DSL which can report errors and warnings that otherwise may not be possible to identify.
– Code reusability and consequently portability across platforms. This is a key advantage in the context of applications that need to be maintained across different platforms and may require architecture-specific strategies.

Due to the independency characteristics of the DSL, integrating the adaptation code into the application can be accomplished through several different mechanisms, such as libraries and APIs, middleware layers, compilers, or interpreters. In spite of the multiple solutions for integrating DSL code into the application code, one of the benefits of a DSL towards interfacing, is the freedom to develop and use any supporting infrastructure that best fits the developers needs. This freedom prevails whether the interfacing and integration is intended to be static or dynamic, less or more intrusive, at compile time or run-time, and using whichever weaving techniques. Nevertheless, regardless of the mechanism, for better logical partitioning and conceptual separation, the original application code should not require major modifications in order to work with our approach. In the end, adaptation specifications are abstracted from the details that defined how the DSL code is integrated into the software application and underlying system.

2.2 DSL Specification

In our DSL, an adaptation *policy* is specified as *strategies* composed of the following components: *declarations*, *operations*, *rules*, and an additional auxiliary *code* section. *Declarations* are reserved for static information that is required for the specification of the adaptation process (e.g., variables to be used, algorithm parameters, function imports). *Operations* specify mainly where the adaptation rules are triggered (e.g., oeprational points for evaluation and action). The *rules* section specifies the actions for adaptability; and in the *code* section, external functions can be defined in supported GPLs. A commented example of adaptation specified with the DSL is shown in Figure 2. It depicts a possible strategy for *StereoNav* where the resolution of the captured images is adjusted according to the vehicle speed.

```
// ---- strategy ----
strategy stereoNavImgResAdapt1{
  // ---- declarations section ----
  // importing a reference to the recurrent stereoNav function
  import function stereoNav(int ransacIterations=5000, int
      imgWidthRes=640, int imgHeightRes=480);
  // importing a reference to a function to get vehicle speed
  import function [int speed] getVehicleSpeed();

  // ---- operations section ----
  operations{
    // rule to be evaluated before the execution
    // of the stereoNav function
    r1 evaluation before stereoNav;
  }

  // ---- rule section ----
  rules{
    // rule invoked at every stereoNav execution
    r1: every(stereoNav){
      // depending on speed, change the image parameter values
      if(getVehicleSpeed().speed <= 50){
        stereoNav.imgWidthRes = 640;
        stereoNav.imgHeightRes = 480;
      }else{
        stereoNav.imgWidthRes = 320;
        stereoNav.imgHeightRes = 240;
      }
    }
  }

  // ---- code section ----
  // (no additional code required for this example)
}
```

Fig. 2 DSL code for adapting image resolution according to the vehicle current speed. Additional comments added to help with the comprehension of the example.

In the DSL code presented, lines 3–7 refer to declarations. They specify the import of two functions: `stereoNav` and `getVehicleSpeed`. The provided default parameter values for `stereoNav` allow for a baseline failsafe execution with no adaptation. Lines 8–13 define the operations, highlighting the evaluation location for rule `r1`, which is triggered before the execution of the call to `stereoNav`. Lines 14–27 enclose the rules section, defining rule `r1`. Rule `r1` retrieves the vehicle speed through the provided function, and depending on its value assigns different values for the image resolution parameters of the `stereoNav` function. Line 17 specifies that rule r1 must be executed for each call of the `stereoNav` function. The auxiliary code section could be provided in the end of the specification.

2.2.1 Policies and Strategies

A policy is the adaptation "program" specifying the modifications to be applied within the application. Any software application is assigned one adaptation policy, while, an adaptation policy can potentially be used by one or more software applications, allowing reusability.

A policy defines strategies of adaptation, further composed of multiple properties and other components that characterize how the strategy is enforced. As an aggregator entity, the policy allows the administration and control over strategies, such as their activation and deactivation, or extension composition schemes. A policy is required to be composed of at least one strategy, however, the possibility to include multiple strategies allows for better adaptive behavior organization. With the definition of multiple strategies, only the first strategy defined is activated, being all others inactive.

The execution of a strategy can often be perceived as a parallel component to the main application logic, in the sense that its execution is concurrent to the application's main workflow. Furthermore, strategy entities can be defined to receive configuration parameters and output values, further promoting reusability to different operational situations with particular characteristics.

2.2.2 Declarations

Declarations are the initial structural section in the arrangement of a DSL strategy. This section's purpose is to describe the necessary fields to be used within the strategy, that provide state, such as variables; and also to indicate references to functions from the target software application that are used within an adaptation specification defined with the DSL.

Variables are declared by specifying a type, an identifier, an initial value, and a DSL-specific supplementary property for value ranges to define (whenever known) the set or range of valid values that a numerical variable may assume. Ranges allow a mechanism for variable saturation, e.g., increments are only considered until the defined maximum value. Regarding functions, it is possible to declare references to application defined functions, and therefore their identifier specified in the strategy must be equal to the original function name and must exist within the application source code. Within a DSL-defined policy, for functions accepting or returning multiple values (i.e., inputs and outputs), the different values are accessed with the DSL using a dot syntax. The DSL also provides additional macro instructions related to functions that encompass sets of instructions and actions that allow for useful functionalities when specifying the adaptations to be performed (e.g., `elapsed_time`, `rate`).

2.2.3 Operations

The *operations* section is responsible for specifying the adaptation's operational connections to the system's computational process. The structure of this section is composed mainly from blocks that specify the special locations for adaptation evaluation and action.

The *operations* section is built with a main block, and a set of possible sub-block structures that are used to define to frame specific steps or components of the system's workflow. Such sub-block structures are used to concentrate operation steps that may be activated or deactivated. An operation point defines a reference to the evaluation and action locations where the referenced rule will be triggered and therefore executed in the source code. Operation points are associated with function calls or with specific locations in the application's source code. Multiple points can be defined for the same rule, allowing the evaluation/action at different points in time. Operation points only define the target location for rule evaluations, and other rule properties, such as rule execution periodicity, are defined in the rule itself. Without the specification of the specific points where rules should be evaluated, it would be a task of the weaver to analyze the application code and to select the observation/monitoring and the action points. In the current weaver, this analysis is not performed and it is a task of the user to explicitly specify those points using the DSL.

2.2.4 Rules

The *rules* section specifies multiple adaptation rules, responsible for performing the necessary adjustments that adapt the behavior of the target application, at the points specified within the *operations* section. Each rule is composed by a rule identifier, a triggering periodicity and condition, and a set of actions. Rule management is concentrated in this section and thus adding, removing or modifying existing rules is accomplished without an added overhead and without changes scattered across several locations. For prioritization and dependency, an evaluation order for rules can be provided, as well as predicates to constrain the execution of their actions (e.g., execution of one action requires the prior execution of another).

Rules are triggered by events, fired when particular conditions occur or periodically. Common triggering conditions are related to memory, CPU, energy, and execution times (e.g., when memory is low, when energy consumption is high). These triggering conditions are provided by imported functions or by the infrastructural support of the DSL (e.g., code generated when integrating adaptations). The execution of a rule may change the application to a new operational state, and there should always exist a transition or a sequence of transitions that allows the application to return to its initial state. In practice, rule executions are modelled as finite state machines. Furthermore, the execution of a rule is atomic, in the sense that when the execution is started it must complete entirely, to avoid partially applied changes, which could result in incomplete and erroneous situations of the application. Such problematic actions could compromise the integrity of the entire application and also the benefits expected from the adaptations.

2.2.5 Code

The *code* section is an auxiliary section to the main DSL structure. Its purpose is to allow developers to extend the DSL specification in order to add functionality using the target programming language. This section also helps the integration with applications. In the *code* section, functions and other external components can be defined in any supported programming language (e.g., C, Java). The *code* section must be parameterized with the name of the programming language in which the code is written.

Regarding integration, the target language code in this section is added to the application source code. Depending on the language, the weaver integrates the code differently. Mainly, the code within this section is inserted to the application code near the adaptations and must be without errors and with all necessary components. For example, in Java, a method defined in the code section is placed within the class where the adaptations that use it are weaved.

3 Implementation: Programming Toolchain

Regarding implementation, the toolchain to support our approach incorporates the adaptations statically into the target source code of the program to be adapted. This toolchain currently supports C and Java programs and involves the validation of the DSL code, compilation of DSL code, injection of the DSL code into the program code, and the compilation of the adaptive program source code. An overview of the toolchain is presented in Figure 3.

For the integration and interfacing between the adaptation and the application code, we are currently applying a joint-compilation process. As such, aiming to support different target programming languages, the compilation process must translate the DSL code into the target programming language, through a compiler tool that knows how the domain abstractions defined with the DSL are represented in another language. Initially, a compilation process translates the DSL code into a target GPL of the application, secondly, the compiled adaptation code is weaved in the application's source code. To implement this process, several solutions surface depending on the target language and platform. Our solution aims at weaving the adaptation code within the application source code at compile-time, and thus requires access to the original source code of the application.

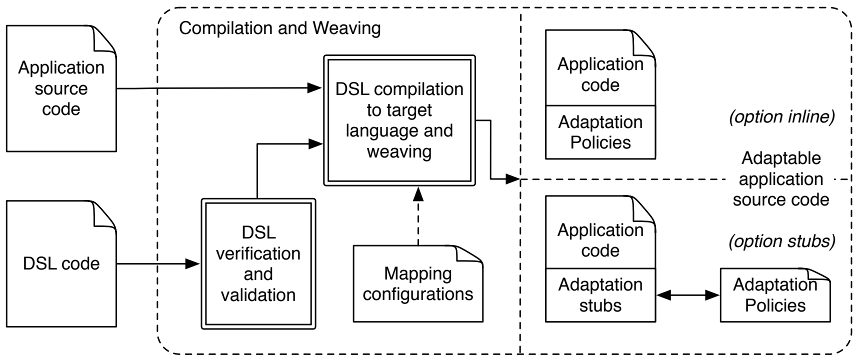


Fig. 3 Overview of the DSL's implementation and programming toolchain.

3.1 Application Source Code Analysis and Adaptation Specification

Each application is different in the sense that its functionality and requirements may constrain or enable certain adaptations. Furthermore, if the application has not been developed having adaptable behavior in mind, the coding style may also determine the range of possible adaptations. Having a set of adaptations in mind to be applied, an analysis of the source code must be conducted to evaluate if these adaptations can, in fact, be applied.

If the application source code was not developed with the intention for the implementation of adaptations, some modifications may need to be applied before adaptations are incorporated, namely to explicitly identify functions, their inputs and outputs, variables, etc. Some code restructuring may also be accomplished, as non-structured code complicates the implementation of adaptations. Nevertheless, the need for these modifications is fairly reduced or even completely mitigated if the application is developed based on best practices, i.e., well-formed and well-developed code (e.g., [5], [10]). The bottom line is that nameable components, clearly identified variables, parameters, and clear conceptual separation between application functionalities allow a more direct integration of adaptations within the application code.

With knowledge of the application's source code and with its restructuring to accommodate the incorporation of adaptations, it is possible to specify the adaptation policies using the DSL. After specification, the adaptation code is verified to provide a validation of the adaptable behavior.

3.2 Verification and Validation Process

Verification and validation of an adaptation policy defined using the DSL is a required process for checking specification consistency, and for determining potential conflicts (e.g., incompatibilities, integrity errors). The verification and validation process is conducted at several levels, each one evaluating different aspects (e.g., rule inter-operability verification), and different targets (e.g., all or only specific sections). Due to its importance, we focus on the specific verification step for analyzing possible conflicts within the rules section, which is a core component in the specification of the adaptable behavior. The existence of multiple rules with several triggering conditions and adaptation actions, may cause potential conflicting situations to arise, namely: (i) rules that share at least a subset of triggering conditions; (ii) rules that manipulate a subset of the same parameters; (iii) overriding or overlapping rules; (iv) rules incompatible due to requirements or objectives. In order to verify the set of adaptation rules defined, we propose a verification process based on automata theory [21].

Adaptation rules are interpreted as automata with a set of adaptation states, a set of triggering conditions, and a transition function that maps the transformation from one adaptation state to another, according to the provided input conditions. Also, automata can hold supplementary data, such as guards, conditions and time restrictions. This process thus allows to model the *rules* section as different automata, and through automata operations, potential conflicting situations are identified, both statically and dynamically, and through automatic and manual mechanisms. Modeling and translating an adaptation rule to an automaton is based on the rule's code control flow graph [2], to identify operational states and the paths that might be traversed with the rule's actions during execution. For example, as rules are often defined as *if-then-else* statements, each branch may hold information defining a new adaptation state, its triggering condition and the set of actions (see Figure 4).

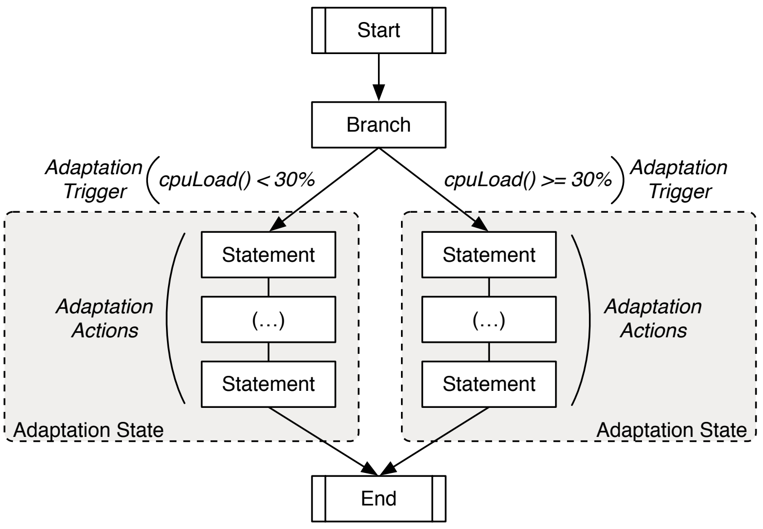


Fig. 4 Rule control-flow translation between domain concepts and automaton components.

3.3 Adaptation Compilation and Weaving

With a valid DSL adaptation specification, the DSL code is translated into the target source code language (e.g., DSL $\rightarrow$ Java, DSL $\rightarrow$ C). This translation is accomplished through specific DSL code generators that allow the posterior incorporation of the adaptations into the application's original source code. A static weaving is used to support the integration of adaptation code with the application's code at compile time. Our current prototype implementation for compilation and weaving is defined by the multiples components, which we describe in the following sections.

For further costumization, there are supplementary compilation and weaving options that extend the configurability of the developed toolchain to specific languages, platforms, devices, etc. At the DSL compilation and code generation stage, the additional compilation-specific mapping and translation characteristics describe details on how the DSL specification is translated and implemented. These configurations are optional and simply allow more customizations for the adaptation policies specified. The supplementary configurations could be defined within the DSL or as a separate parallel auxiliary configurability mechanism.

3.3.1 Compilation

The adaptations defined within DSL code are translated to code abstractions of the target programming language. The compiler currently supports C and Java as target languages. This process stage allows a direct translation of concepts, being rules the DSL section where most of the compilation is concentrated. The most relevant compilation translations are:

- Declared variables are translated to global scope variables with the same type and initialization. Saturation enforcing is performed by capping the variable value using if-then-else statements.
- Dot syntax access to variables in the DSL are translated to the appropriate access format depending on the type of variable and scope.
- DSL specific macros are compiled to functions and the necessary support code (e.g., the `elapsed_time` macro requires the measurement of a function's start and end time).
- Rule temporal triggers (i.e., periodicity) are implemented as timers. In Java, the implementation uses Timer and TimerTask classes. In C the implementation uses a developed library which steers the implementation with specific adjustments according to the platform, i.e., hardware timers with our Xilinx embedded boards, and in Linux using the signal library.
- Rule prioritization, evaluation order, and predicates are implemented using control variables and branching statements.

3.3.2 Weaving

The parts of the DSL code compiled to the target language need to be integrated with the application. Our current weaving process is based on the aspect-oriented approach, as the generated code from the DSL is woven within the application source code at specific code locations. For code insertion, the current implementation requires the processing of the application source code to obtain a data structure that is used to aid with DSL code insertion. The most relevant weaving operations are as follows:

– An analysis of the source code for verification of the existance of the operation point locations defined earlier in the DSL code.
– Insertion of the compiled DSL code for rule execution is accomplished at the operation locations defined.
– DSL defined function default values are translated and within the application code as initialization values for the functions. Besides initializations, rules may further influence the assignments of these values.
– Additional code provided in GPL is inserted as-is near the code location where it is used. In Java, a method defined in the code section is placed within the class where the adaptations that use it are weaved; whereas in C, a function is placed within the same file.

Moreover, additional mapping-specific configurations can describe implementation details on how the DSL code should be integrated, namely on the type of adaptation code produced or platform particularities. Such configurations defined steer the type of mapping options that are used when both translating source-to-source the DSL code to the application code, and the mechanisms used in weaving. Figure 3 depicts two different targets of the process of compilation: an application with embedded inline adaptations, and an application with stub functions to connect to the adaptation code. Although adding stub functions for adaptations may cause additional computational complexity (i.e., function calling), this solution is more modular and maintains the separation between the main application logic and the adaption logic. Figure 5 shows an example in C of the generated code relative to adaptations implemented within stub functions or inlined within the application code.

With explicit mapping configurations and using the stub function generation, more beneficial solutions can be defined. For example, in a multicore scenario and considering direct communication between cores, a specific mapping configuration could be defined so that the adaptation logic sits within an independent processing core. Adaptation stub functions within the application code would be replaced with the appropriate calls to the other core, since the adaptation code itself would be defined inside another function to be executed in the other core (see Figure 6).

```c
// (...)
if(vehicleSpeed < 25){
  // ommitted for simplicity of the example
}else{
  ransacIterations = 5000;
}
// (...)
```

```c
// (...)
r_ransac_speed(vehicleSpeed);
// (...)
```

Fig. 5 Generated code excerpts for the same adaptation defined inlined (top) or within stub functions (bottom) in the application code (in C).

```c
// (...)
void r_ransac_speed_put(int vehicleSpeed){
  putfsl(3,0); // code for adaptation request
  putfsl(vehicleSpeed,0); // send the vehicle speed
}
void r_ransac_speed_get{
  putfsl(4,0); // code for specific adaptation get
  getfsl(ransacIterations,0); // receive iterations
}
// (...)
```

```c
// (...)
...{
  getfsl(codeReceivedFromProducer,0); // strategy identifier
  switch(codeReceivedFromProducer){
    // (...)
    case 3 : // adaptation ransac iteration speed
      updateRansacIterationNumberThroughVehicleSpeed();
      break;
    case 4 : // adapted ransac iteration
      sendRansacIterations();
      break;
  }
}
void updateRansacIterationNumberThroughVehicleSpeed(){
  getfsl(vehicleSpeed,0);
  if(vehicleSpeed < 25){
    // ommitted for simplicity of the example
  }else{
    ransacIterations = 5000;
  }
}
void sendRansacIterations(){ putfsl(ransacIterations,0); }
// (...)
```

Fig. 6 Generated code for adaptations as stub functions targeted at two cores with communication via direct channels using port number 0 (in C). The first section defines the code that sits within the application code, and the second section the adaptation code.

3.4 Adaptive Code Compilation and Deployment

With the adaptations compiled and incorporated into the application source code, the complete program is again compiled using now the standard tools approriate for the application source code language (e.g., gcc [18], javac [31]). In short, the application has now been compiled from a new program code, based on the original source code plus the adaptations, and therefore the new application can now be deployed and executed in the target environment.

4 Case Study: Stereo Navigation

To validate our approach we use a case study application for avionics, which consists of an industry-developed embedded navigation system, named as Stereo Navigation Application (StereoNav), and whose main stages are described in detail in [32]. The *StereoNav* application takes as input two independent images from the same or multiple cameras, extracts features, and then represents them in a 3D-space. The analysis of these features allows for pose estimation of a vehicle, thus supporting its localization and navigation.

Furthermore, the *StereoNav* navigation process includes several input parameters such as image capture frequency or resolution, whose configuration impacts both the output and the computation requirements of the process (e.g., execution time, memory). Also, there is an explicit interest and requirement of the original industry developer to easily manage these parameters. These characteristics make *StereoNav* an ideal scenario to illustrate several aspects of the use of our DSL-based approach to specify adaptations to the algorithm and measure their impact.

However, additionally to the *StereoNav* case study, and to show the expressiveness of this approach and its applicability to most general applications, we have been focusing on Java-based case studies that require adaptations, namely: (i) physical activity context-inference application that executes on constrained mobile phone environments and thus adaptions are applied to optimize the context inference process, by customizing different methods to better infer the desired contexts [36]; and (ii) mobile robot navigation application used for localization where several adaptations provide configurations that allow a more optimized execution of the algorithm, vital in constrained mobile phone environments where it executes [37].

4.1 Experimental Setup

The *StereoNav* application is part of an industry-developed navigation system, whose main algorithm was developed in the C programming language and is prepared for execution on traditional personal computers and embedded computing systems. For our experimental evaluation, we used three setups:

- **Setup 1**: PC with a 2.0 GHz Intel Core 2 Duo and 2 GB of 667 MHz DDR2 SDRAM. C code was compiled with *gcc*. Execution times were measured with the `time.h` library using the `clock()` function.

- **Setup 2**: Xilinx FPGA, with a PowerPC processor running at 400 MHz and with heap size of 256 MB and stack size of 4 MB. The C code was compiled using *ppc-gcc*, a *gcc* compiler instance targeting the PowerPC processor, and *-O2* optimization level. Execution times were measured using hardware timers implemented in the FPGA.

- **Setup 3**: ML510 FPGA Development Board – Xilinx Virtex-5 FPGA, model XC5VFX130T. The architecture consists of two Xilinx MicroBlaze (MB) processors (MB0 and MB1), each one with their own on-chip data and instruction memories, an external 1 GB DDR2 SDRAM memory shared by the two processors, direct and blocking communication channels (FIFOs) between the two processors, timers and UART components. The *StereoNav* application data is loaded on the external memory.

4.2 Algorithm

The overall structure of the algorithm is presented in Figure 7 depicting its key operations and parameters. The first three operations are performed concurrently for each camera sensor, whereas the remaining operations execute sequentially. An explanatory summary of each operation follows.

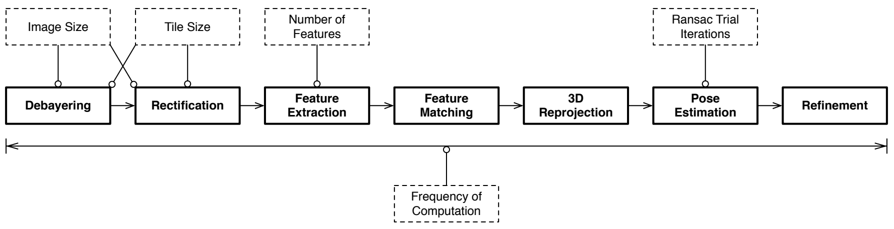


Fig. 7 *StereoNav* algorithm: main steps (solid boxes) and input parameters (dashed boxes).

Debayering interpolates the input image data in a Bayer grid to GRGB output. Various interpolation methods can be used differing in the quality of the produced output image and computational cost.

Rectification projects the stereo images onto a common image plane, allowing correction of image distortion by transforming the images into a standard coordinate system. Rectification can be performed through different "warping" techniques, that yield different results both in terms of quality and computational cost (e.g., bilinear interpolation gives better results but it is ten times slower than nearest neighbor [32]).

Feature extraction detects elements that can serve as reference locations in the image, and it is usually performed with corner detectors (e.g., *Harris corner detector* [20]). Typically, around 100–1000 features are extracted [32] and this information is stored as compactly as possible in a hash-like structure.

Feature Matching generates assignments between the extracted features, using feature data from the previous frame, the current frame, and from different cameras. Its objective is to detect feature-vectors that are identical. As the probability of having a correct match in a cluttered urban environment is low, a circular check mechanism is used in order to improve assignment accuracy.

3D Reprojection derives 3D coordinates of a point from different image projections of that point, given the feature matching from the previous step. A 3D reprojection may be performed for a set of features at a given time, so one can calculate a set of points in the 3D space at once [33].

Pose Estimation produces the transformation between two camera reference frames, allowing to determine the ego-motion from that information using the dead reckoning (odometry integration) or a SLAM (Simultaneous Localization and Mapping) approach. This estimation is performed using the *RANSAC (RANdom SAmple Consensus)* algorithm [15].

Refinement may include some operational tweaks in order to produce the most accurate position estimate.

4.3 Adaptation Analysis

The *StereoNav* application requires a high-level of accuracy to maintain an acceptable quality-of-service (QoS), measured in terms of timely correct navigation information. The overall algorithm must be able to handle certain problematic situations, such as changes in vehicle speed and availability of computational resources. Possible failures or application performance degradation are eliminated by a dynamic adaptation of the application, as certain troublesome situations may lead to significant reduction of service's quality or even the complete loss of service altogether [33].

The stereo navigation algorithm includes several operations. The most computationally demanding are the most relevant for adaptation when the execution time and/or energy savings are the primer objectives. Figure 8 reveals that (in both setups) the *feature extraction* and *pose estimation* operations are the most time-consuming, on average accounting for 89.3% and 5.6% of the total execution time, respectively. Analyzing these operations, there are two main candidate parameters that yield a significant computational impact: (i) the resolution of the processed images in the *feature extraction* operation, and (ii) the number of iterations within the *pose estimation* operation.

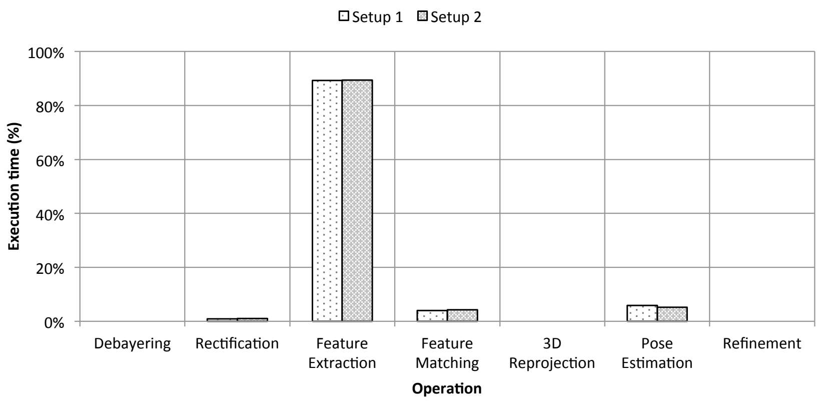


Fig. 8 Execution time impact of the main operations in *StereoNav* considering both setups.

4.4 Adjusting the Image Resolution

Changing the image resolution mainly influences the *feature extraction*, which is the most computationally-intensive operation. For the purpose of our scenarios, we considered the two image resolutions of 640×480 (high resolution) and 320×240 (low resolution). In both setups used, the execution time is on average above 3 to 4 times greater for the higher than for the lower resolution.

4.4.1 Adapting to the Vehicle Speed

Considering the influence of image resolution in the overall execution time, it is of interest for the application to have an adaptation strategy that dynamically reconfigures the resolution according to the current vehicle speed. Decreasing the image resolution as the vehicle speed increases aids navigation since, the faster the vehicle moves, the faster the application needs to determine its location. Calculating the image resolution in relation to vehicle speed (in km/h) is performed as follows: 640×480 if $speed \leq 50$; 320×240 if $speed > 50$.

Figure 9 shows the change in vehicle speed, the overall algorithm execution time, and the image resolution over time. In this strategy, when the speed increases, the image resolution decreases, consequently decreasing the overall execution time (decreasing resolution from 640×480 to 320×240 yields a sixfold computational time decrease).

The DSL code for this adaptation strategy was presented in Figure 2 and specifies that for each `stereoNav` step, the vehicle speed is measured and according to its value, the size of the input images is modified.

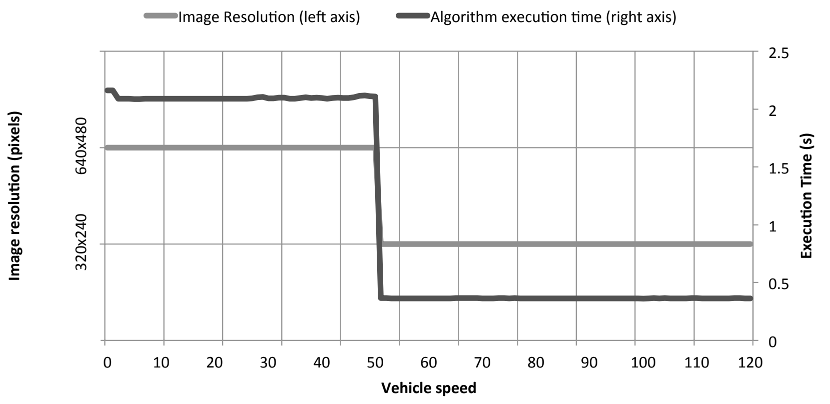


Fig. 9 Adjustments to image resolution and impact on the overall execution time, with respect to vehicle speed (using setup 1).

4.4.2 Adapting to a Time Constraint

Ensuring the algorithm executes within a time constraint, due to resource constraints or imposed deadlines, requires a strategy where if this time constraint is not satisfied, then the image resolution is decreased for the subsequent execution, otherwise the image resolution is increased, improving QoS. Experimental results of this strategy considering a time window with variable size are presented in Figure 10.

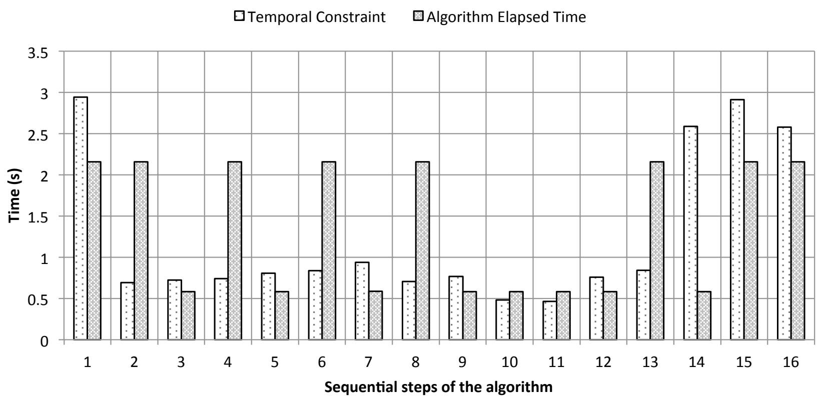


Fig. 10 Adjustment of image resolution to a time constraint (using setup 1). Steps 1, 2, 4, 6, 8, 13, 15, 16 use high resolution and steps 3, 5, 7, 9, 10, 11, 12, 14 use low resolution.

In each iteration where the algorithm exceeds the time constraint, the next iteration will have a lower image resolution to satisfy the time constraint (e.g., iterations #1 and #2). On the other hand, satisfying the time constraint causes the next iteration to be executed with higher image resolution (e.g., iterations #3 and #4). Since only two image resolutions are being used, violating the constraint in low resolution or satisfying with high resolution, causes the system to maintain the resolution as it cannot decrease or increase the resolution beyond that. As constraint violations cause subsequent delayed executions, the strategy for image resolution adaptation, whose illustrated results are shown in Figure 10, causes a time violation in approximately 44% of the iterations. In contrast, using a fixed high image resolution (640 $\times$ 480) would cause a percentage of constraint violations around 75%. Using a fixed low image resolution (320 $\times$ 240) would fail in only 12.5% of cases. Although using a low resolution results in fewer violations, the average image quality used would be lower limiting the overall application QoS.

The strategy for adapting image resolution according to a time constraint is specified in Figure 11. Contrasting with the DSL code from Figure 2, the vehicle speed function is replaced by a function that outputs the computation time requirement (line 2) allocated for an iteration of the algorithm. Lines 3–5 define the evaluation location for the rule. Lines 6–16 define a rule that depends on the DSL infrastructure provided macro, which holds the elapsed time of the last `stereoNav` execution. Rule `r2` retrieves the time constraint and if the constraint is greater or equal than execution time of the previous `stereoNav` iteration, then the navigation algorithm is parameterized to use the high image resolution. Otherwise, the low resolution for the images is used.

```
 1 import function stereoNav(int ransacIterations=5000, int
       imgWidthRes=640, int imgHeightRes=480);
 2 import function [double time] getTimeConstraint();

 3 operations{
 4   r2 evaluation after stereoNav;
 5 }

 6 rules{
 7   r2: every(stereoNav){
 8     if(getTimeConstraint().time >= stereoNav.elapsed_time){
 9       stereoNav.imgWidthRes = 640;
10       stereoNav.imgHeightRes = 480;
11     }else{
12       stereoNav.imgWidthRes = 320;
13       stereoNav.imgHeightRes = 240;
14     }
15   }
16 }
```

Fig. 11 DSL code for adapting the image resolution according to a time constraint.

4.5 Adjusting the Number of *RANSAC* Iterations

The *pose estimation* operation is implemented by the *RANSAC* iterative algorithm. The higher the number of algorithm iterations, the higher computation demand (e.g., execution time) but also the higher the probability of a correct pose estimation. Therefore, this algorithm is an ideal candidate for run-time algorithmic adaptation based on available system resources. As the base acceptability criteria for pose estimation is 90% [33] we introduce configurations guaranteeing 90%, 92%, 94%, 96%, and 97%, corresponding to 3900, 4300, 4800, 5500, and 5800 iterations, respectively.

4.5.1 Adapting to the Vehicle Speed

Similarly to the strategy of adapting the image resolution according to vehicle speed, we now consider an adaptation strategy that varies the number of *RANSAC* iterations. As the speed of the vehicle increases, the number of iterations must be reduced causing less computational strain and therefore less execution time is required for the operation. The adjustment in the number of *RANSAC* iterations due to vehicle speed is defined as follows: 5800 if $speed < 25$; 5500 if $25 \leq speed < 50$; 4800 if $50 \leq speed < 75$; 4300 if $75 \leq speed < 100$; and 3900 if $speed \geq 100$. Figure 12 plots the results measured in one scenario, depicting the number of iterations and execution time of the *RANSAC* algorithm as the vehicle speed increases.

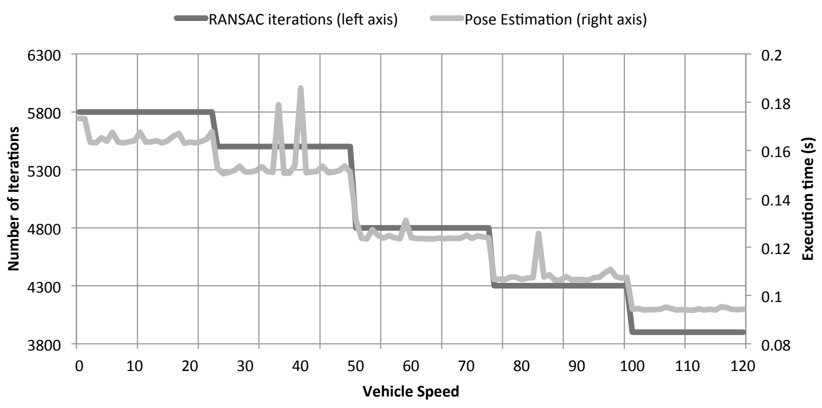


Fig. 12 Number of *RANSAC* iterations and corresponding execution time of *pose estimation* regarding the variation of vehicle speed (using setup 1).

The DSL specification code for this adaptation control strategy is similar to Figure 2, with the exception of the `rules` section, which is presented in Figure 13. The code presented specifies that rule `r3` executes before the beginning of every `stereoNav` iteration, retrieving the vehicle speed through a function. Depending on its evaluation through conditions, it assigns the number of iterations to be conducted by the *RANSAC* algorithm. This is performed by changing the input parameter value of the *StereoNav* algorithm.

```
rules{
  r3: every(stereoNav){
    int speed = getVehicleSpeed().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. 13 DSL code for adapting the *RANSAC* iterations according to vehicle speed.

4.5.2 Adapting to a Time Constraint

Another adaptation strategy of interest is based on a time constraint for the execution of the *pose estimation* operation, as proposed in [34]. This strategy is described as: (i) measure the execution time of the operations that precede *pose estimation* (t_{pre}) to verify how much time has already been elapsed; (ii) get the execution time allowed for the whole *StereoNav* algorithm (t_{total}) considering the available computational resources; (iii) calculate the execution time for the *pose estimation* operation as $t_{est} = t_{total} - t_{pre}$ (here, the execution time required for the computation of the refinement operation is assumed to be negligible, as shown in Figure 8); (iv) select the number of iterations for the *RANSAC* algorithm according to the time available for the operation (t_{est}), when the execution times of predefined numbers of iterations are known. If there is not enough time to perform the operation with the lowest predefined iteration number, then the time requirement for execution is not satisfied.

The proposed adaptation is possible due to previous knowledge of the average execution time which the operations require. Knowing the available execution time, it is possible to perform the highest number of *RANSAC* iterations and maintain QoS levels as highest as possible given the time constraint. Figure 14 depicts the behavior of this adaptation strategy in a testing scenario where the *RANSAC* iterations are adjusted to satisfy time constraints. In this testing scenario, only four iterations missed the constraint (iterations 1, 14, 15, 18 had execution times greater than the available time). The implementation in the DSL of such strategy is presented in Figure 15, where two new imported functions (lines 1 and 2) output the control variables needed to assign the iteration number. Lines 7 to 22 set the appropriate iteration number according to the available time for *pose estimation*.

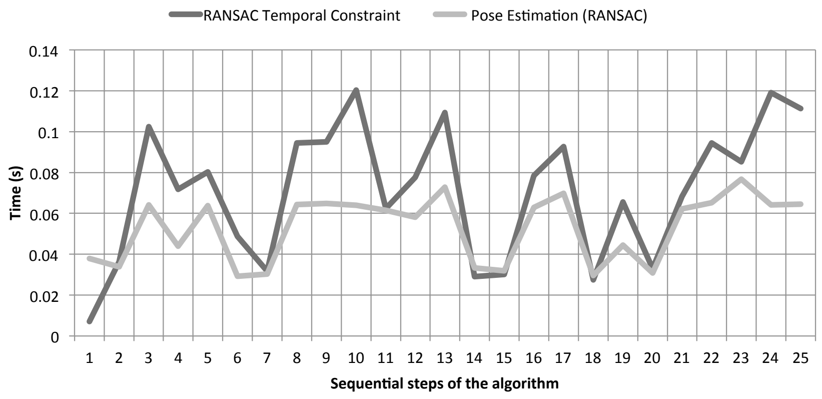


Fig. 14 Time allowed in seconds and corresponding execution time of the *pose estimation* operation. The number of iterations used sequentially in each step were: 3900, 3900, 5800, 4800, 5800, 3900, 3900, 5800, 5800, 5800, 4800, 5500, 5800, 3900, 3900, 5500, 5800, 3900, 4800, 3900, 4800, 5800, 5800, 5800, 5800 (using setup 1).

```
import function stereoNav(int ransacIterations=5000, int
    imgWidthRes=320, int imgHeightRes=240);
import function [double tAnt] getRansacTimeAnt();
import function [double tTotal] getTimeBudget();

operations{
  r4 evaluation point "beginRansac"; // at the defined label
}

rules{
  r4: every(stereoNav){
    budget = getTimeBudget().tTotal - getRansacTimeAnt().tAnt;
    if(budget > 0.08){
      stereoNav.ransacIterations = 5800;
    }else if(budget > 0.075){
      stereoNav.ransacIterations = 5500;
    }else if(budget > 0.06){
      stereoNav.ransacIterations = 4800;
    }else if(budget > 0.05){
      stereoNav.ransacIterations = 4300;
    }else{
      stereoNav.ransacIterations = 3900;
    }
  }
}
```

Fig. 15 DSL code for the *RANSAC* iteration adaptation strategy according to an available time budget for the computation of the algorithm.

4.6 Adjusting Multiple Parameters

For this strategy we use previous knowledge from experiments with the algorithm. Considering the two input parameters that have been used for adaptation, critical to the most time consuming operations, their impact is different as the image resolution parameter causes greater impact in the application than the number of iterations of the *RANSAC* algorithm. Due to this difference in impact, the image resolution is used for coarser adjustments and the number of iterations for finer adjustments.

Since the application provides information for navigation, it must comply with requirements for frequency of execution. One can devise an overall adaptation strategy that adjusts the parameters identified according to the number of frames per second (FPS) required to maintain a suitable navigation information at the presented vehicle speed. Faster speeds reduce the available time budget for the application to execute and therefore require faster execution frequency. The strategy is defined as follows: (i) the frequency of computation is calculated as a function of the vehicle speed; (ii) from the required frequency of computation, a time budget is defined; (iii) considering the available time budget, the algorithm configuration for execution must yield the best results while satisfying the time constraint; (iv) to ensure the time constraint compliance, it should be verified if the algorithm executed within the budget.

The DSL code for this adaptation strategy is presented in Figure 16. The specification contemplates two imported functions, one function defined internally and three different rules. Lines 1 to 5 define necessary variables. Lines 6 to 7 specify the necessary imported functions. Lines 8 to 11 specify the operations section. `Rules` are specified from lines 12 to 40. Rule `r1` computes the difference between the available time and the elapsed time from the execution of the `stereoNav` function. If the elapsed time is less than the available time, then there is room for improvement and rule `r3` is invoked. If the elapsed time exceeded the available time, then reduction of the execution time must be accomplish and therefore rule `r2` is invoked. Lines 41 to 43 specify the code for a *C* function `calcFPStime`, which computes the frequency of computation.

```
 1 int[][] imgRes = {{160, 120}, {320, 240}, {400, 320}, {480,
       400}, {560, 480}, {640, 480}};
 2 int imgResCounter = 1[0..6]; // value range from 0 to 6
 3 int[] iterations = {3900, 4300, 4800, 5500, 5800};
 4 int iterCounter = 1[0..4]; // value range from 0 to 4
 5 double deadline = 0;

 6 import function stereoNav(int ransacIterations=iterations[
       iterCounter], int imgWidthRes=imgRes[imgResCounter][1], int
        imgHeightRes=imgRes[imgResCounter][2]);
 7 import function [int speed] getVehicleSpeed();

 8 operations{
 9   r0 evaluation before stereoNav;
10   r1 evaluation after stereoNav;
11 }

12 rules{
13   r0:every(stereoNav){
14     deadline = code.c.calcFPStime(getVehicleSpeed().speed);
15   }
16   r1:every(stereoNav){
    slack = stereoNav.elapsed_time - deadline;
    if(slack > 0){ // exceeded the available time
      evaluate r2;
    }else{ // satisfied the deadline
      evaluate r3;
    }
  }
  r2{
    if(slack > 0.5){ // reduce exec. time through image
        resolution
      stereoNav.imgWidthRes = imgRes[imgResCounter--][1];
      stereoNav.imgHeightRes = imgRes[imgResCounter--][2];
    }else{ // reduce time through adjustment in iterations
      stereoNav.ransacIterations = iterations[iterCounter--];
    }
  }
  r3{
    if(slack < -0.5){ // improve quality through image
        resolution
      stereoNav.imgWidthRes = imgRes[imgResCounter++][1];
      stereoNav.imgHeightRes = imgRes[imgResCounter++][2];
    }else{ // improve quality through adjustment in iterations
      stereoNav.ransacIterations = iterations[iterCounter++];
    }
  }
}

code.c{
  double calcFPStime(int speed){return speed/10;} // in seconds
}
```

Fig. 16 DSL code for the a composed adaptation strategy that adjusts the image resolution and the number of *RANSAC* iterations.

4.7 Targeting Different Execution Scenarios

In order to show and to evaluate a possible mapping of the adaptation and the application codes into different processors, we prototyped an embedded dual-core system, as described by setup 3.

4.7.1 Execution Scenarios

For execution scenarios, we conduct experiments on the strategies previously presented considering different options for compilation and weaving. The strategies considered are as follows: (A) adjusting the image resolution according to vehicle speed; (B) adjusting the image resolution according to time constraints; (C) adjusting the *RANSAC* iterations according to vehicle speed; (D) adjusting the *RANSAC* iterations according to time constraints; (E) combination of strategies A and C; and (F) combination of strategies B and D.

Regarding the interface between adaptation and application code, we consider three scenarios: (i) adaptations embedded inline executing in a single core (1-MB inline); (ii) adaptations integrated as stub functions executed in a single core (1-MB stubs); and (iii) adaptations integrated as stub functions executed in a second core (2-MB stubs). For the multicore architecture, processors MB0 and MB1 are responsible to execute the application and the adaptations, respectively. MB0 executes the stereo navigation application code and requests the adaptations to MB1. MB1 receives the requests from MB0, executes the adaptation behavior accordingly, and reply, e.g., with parameter values. The communication between the application and the adaptations (i.e., between MB0 and MB1) is via the direct communication channels.

4.7.2 Impact on Execution Time

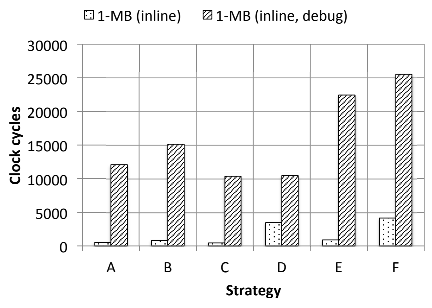
Table 1 presents the comparison for execution time considering the execution of the generated adaptation-related code according to the different adaptation strategies and execution platforms, as presented previously. Table 1 shows that the embedded inline adaptation code is the fastest, on average, mainly because it neither has the overhead of calling functions nor communication primitives (around 1.06× speedup regarding 1MB stubs and 1.01× speedup regarding 2MB stubs). With respect to the adaptation code encapsulated into stub functions, the version executed on 2MB is faster because although it requires additional communication primitives between the two cores, the adaptation strategies are not being executed in the same core, and thus relieve the application core from additional processing (around 1.05× speedup). Although in this experiment we expected minor performance improvements as the adaptation strategy is very simple and not computationally intensive, it reflects an example of a DSL specification with separation of adaptation and application logic making feasible the generation of different implementations.

Table 1 Execution time (in clock cycles) for different strategies and architectures.

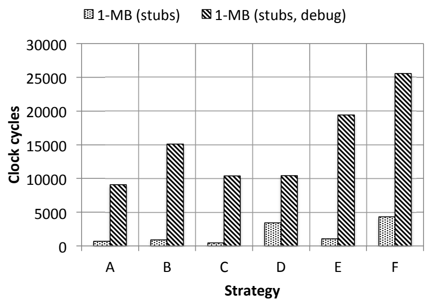
Strategy	1–MB (inline)	1–MB (stubs)	2–MB (stubs)
A	543	680	581
B	801	882	868
C	461	458	331
D	3308	3416	3413
E	880	1041	813
F	4163	4323	4275

Additionally, we performed the same experimental tests with added debug information to the adaptation strategies. In this case, the strategies are responsible for printing verbosely the information regarding several aspects of the adaptation actions being performed. Our objective is to assess the impact of this additional code complexity added on the execution time considering the single and the multicore architecture. Higher complex strategies show how the different execution scenarios scale regarding the execution time metric.

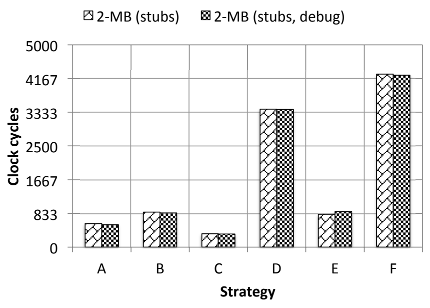
Figure 17 shows the execution time (clock cycles), regarding the execution of each strategy when seen by the processor core (MB0) executing the application, and according to the different execution scenarios. For each strategy, Figure 17 considers the initial versions and the newer versions with the added debug instructions and thus complexity.



(a) 1MB-Inline.



(b) 1MB-Stubs.



(c) 2MB-Stubs.

Fig. 17 Execution time in clock cycles of each strategy according to the different execution scenarios, measured from the main application logic.

Figure 17 shows that with the increase of the complexity of the adaptation strategies, the benefits of the use of two cores is evident, as the execution time does not increase as it does with the single core versions. In the multicore version, as the computation of the strategies is on a dedicated core, the application overall execution time is almost the same as the adaptations are computed in parallel. The benefit of this parallelism can thus be achieved at a higher-level of abstraction and is therefore not dependent on the power of a compiler to extract this parallelism.

4.7.3 Generated Code Analysis

The code generated, considering the different execution scenarios, is of different complexity and modularity, and thus also of different comprehension levels.

Table 2 presents a comparison of the different C code generated consisting of the application and the adaptation code according to several metrics, measured using the Source Monitor tool [11].

Table 2 Evaluation metrics for different C code generated regarding the incorporation of the adaptation code on the application source code (blank lines are ignored).

Code Metric	Original	Adaptation Prepared	1–MB (inline)	1–MB (stubs)	2–MB (stubs)
Files	58	58	60	60	61
Lines	13083	13525	14111	14173	14342
Statements	7367	7363	7735	7783	7889
Functions	165	170	185	202	207
Avg. Complexity	8.07	7.82	7.71	7.14	7.03

In the metrics presented, the average complexity represents the arithmetic average of all complexity values measured for each function, which represent the number of execution paths through a function, to which the number of branch statements, boolean logic, loops, and others contribute to [28]. Higher values of complexity reveal less understandable code.

Table 2 also includes metric values for the version of the application reflecting the preparation of the original source code to be adapted (e.g., changing some hardcoded values to variables that are of interest for adaptation). From the results presented for mapping configurations, for the generated code in C, it is possible to observe that there are many differences between the implementations. The metrics show that the embedded inline version is more complex than all the versions involving adaptations encapsulated within stub functions. Although using stub functions increases the number of lines, statements and functions, the average values for the complexity metric decrease. The adaptations inlined in the application code are less scattered, but are tangled and with code repetitions, whilst adaptations encapsulated into stub functions are modular, readable and easier to maintain. Integrating the adaptation code results in an increase on the number of files, number of lines, statements and functions. However, the average complexity is lower. The difference in metrics between the mechanisms for DSL adaptation code generation highlight the benefits of multiple possible versions and also of the separation of concerns provided by having the generation options as mapping configurations at the adaptation logic level.

5 Related Work

Embedded systems are gaining momentum due to active research in topics such as context-aware computing and ubiquitous computing (e.g., [6], [23]). Due to the architectural characteristics of such systems, run-time adaptations of embedded applications are required to achieve performance goals under changing operating situations [13]. The concept of adaptation has been commonly applied in embedded systems targeting energy concerns, because of their limited power capability. Examples of such adaptation include energy-efficient location-aware applications that switch between different sensors (e.g., [41]).

Software adaptation techniques have been addressed in many areas of research and under many forms [24]. As such, adaptations can be defined in a multitude of ways, however, with different objectives, costs and capabilities. Adaptation specification and implementation can range from simple conditional expressions to highly complex software architectures. While some simple solutions incorporated within the application logic can suit more humble adaptation needs, other more elaborate needs can only be accomplished with more flexible and independent solutions that separate the application from the adaptation logic. This separation is beneficial not only in terms of design, but also because adaptations often go beyond the core functionality of the application. Also, this separation into layers can express different design alternatives and configurations of the same software [9]. Specifically, techniques for software adaptations have been tackled through conditional branching (e.g., [16]), context-oriented programming (e.g., [4], [19]), aspect-oriented programming (e.g., [26], [40]), feature-oriented programming (e.g., [3]), and architectures/frameworks (e.g., [16], [17]).

When adding adaptations to software, the use of additional parameters and conditional expressions are the most commonly seen due to their low-barrier to initial development. However, although introducing additional coding segments solves some simple and straightforward adaptation efforts, their continuous usage introduces clutter, confusion and code comprehension difficulties by mixing adaptation with application functionality. These drawbacks eventually make software evolution less flexible and more costly.

When concentrating on contextual information, context-oriented programming (COP) approaches (e.g., Subjective-C [19], ContextJ [4]) are relevant to apply layered operational behavior, dependent on contexts, to certain code sections. However, although most COP approaches foster some degree of domain specificity, more than often their end result is similar to hard-coded conditional statements. Furthermore, their common approach of extending other host languages lock them to both their host's benefits and drawbacks, invalidating a broader and more general solution to adaptation specification.

Alternatively, focusing on cross-cutting concerns, the AOP paradigm [25] allows the separation of concerns aiming at defining aspects that can specify behavior to be woven into applications and thus possibly for adaptation (e.g., AspectJ [26], AspectC++ [40]). As AOP defines an approach, it has several possible implementations, which cause development and design challenges. Moreover, traditional pointcut mechanisms do not typically include constructs and semantic for most weaving actions needed by run-time adaptation behavior (e.g., program execution points obeying to certain periodicity). Furthermore, being somewhat general for any aspect-weaving necessities, adaptation-specific abstractions have been somewhat neglected.

As DSLs are tailored to specific application domains, they offer substantial advantages in expressiveness and ease of use when compared to general-purpose programming languages (GPLs) [29]. Several DSLs have been proposed for software adaptation (e.g., [1], [22]). For example, in the context of a video processing application [1] the authors presented an extension to the BZR language for adaptation control on a mobile phone, where the video display modes are controlled by the adaptive system according to the status of computing resources. Unlike our DSL, their adaptive behavior is specified in terms of hierarchical automata and through contract policies.

Finally, several ad-hoc solutions have been proposed (e.g., [8]), as authors develop their own specific adaptation approaches tailored to their specific cases and applications. Although beneficial for the applications they are developing, these solutions usually lack generality and reusability.

6 Conclusion

This paper presented the feasibility of our DSL-based approach to specify adaptable behavior for a real-life vehicle navigation application provided by industry. The experimental results highlight not only the benefits and impact of the application's adaptation, but most importantly, our DSL's advantage of providing a high-level programmable approach to define such adaptability.

With the DSL described here, adaptation strategies are decoupled from the applications original code and are easily modified without having to rewrite the application code. Furthermore, different adaptation strategies can be shared and deployed to different platforms and target languages thus promoting programmer and application portability. With the behavior defined in the DSL, it is possible to see how different strategies are specified and to understand the impact of adding new rules, including different functions, and evaluating the differences from one adaptation to another. We also note the degree of similarity between the specifications of different strategies. This aspect facilitates behavior modifications, thus reducing the complexity of managing DSL code while allowing for rapid testing and validation of different strategies as for example, when changing the target vehicle, or when trying to accommodate other speed ranges.

Our ongoing and future work is focused on DSL extensions and improvements, namely through: additional case studies, extending conflict and traceability analysis, and extending our compiler/weaver support to allow multiple implementations for the same DSL abstractions in order to take futher advantages of plataform characteristics.

Acknowledgements The work presented was partially supported by *Fundação para a Ciência e a Tecnologia* (FCT) under grant SFRH/BD/47409/2008. Cardoso, Diniz, and Petrov acknowledge the partial support by the European Framework Programme 7 (FP7) under contract No. 248976. Any views expressed in this work are those of the author(s) and do not necessarily reflect the views of the European Commission. The authors would also like to thank Ali Azarian for his help with the FPGA setup used for experimental evaluation.

References

1. Aboubekr S, Delaval G, Rutten E (2009) A Programming Language for Adaptation Control: Case Study. SIGBED Review 6(3):11:1–11:5
2. Allen FE (1970) Control Flow Analysis. In: Proceedings of a Symposium on Compiler Optimization, ACM, pp 1–19
3. Apel S, Leich T, Rosenmüller M, Saake G (2005) FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Generative Programming and Component Engineering, Lecture Notes in Computer Science, vol 3676, Springer, pp 125–140
4. Appeltauer M, Hirschfeld R, Haupt M, Masuhara H (2011) ContextJ: Context-oriented Programming with Java. Information and Media Technologies 6(2):399–419
5. Arnold K, Gosling J, Holmes D (2000) The Java Programming Language, vol 2, 3rd edn. Addison-Wesley
6. Baldauf M, Dustdar S, Rosenberg F (2007) A Survey on Context-Aware Systems. International Journal of Ad Hoc and Ubiquitous Computing 2(4):263–277
7. Bell J, Bellegarde F, Hook J, Kieburtz RB, Kotov A, Lewis J, McKinney L, Oliva DP, Sheard T, Tong L, Walton L, Zhou T (1994) Software Design for Reliability and Reuse: A Proof-of-Concept Demonstration. In: Proceedings of the Conference on TRI-Ada, ACM, pp 396–404
8. Bishop J (1994) Languages for Configuration Programming: A Comparison. Tech. rep., Computer Science Department, University of Pretoria
9. Bobrow DG, Goldstein IP (1980) Representing Design Alternatives. In: Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior
10. Broemmer D, Mac F (2002) J2EE Best Practices: Java Design Patterns, Automation, and Performance, 1st edn. John Wiley & Sons, Inc.
11. Campwood Software (last visited in June 2013) SourceMonitor Version 3.4. `http://www.campwoodsw.com/sourcemonitor.html`
12. van Deursen A, Klint P, Visser J (2000) Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices 35(6):26–36
13. Ensink B, Stanley J, Adve V (2003) Program Control Language: A Programming Language for Adaptive Distributed Applications. Journal of Parallel and Distributed Computing 63(11):1082–1104
14. Figo D, Diniz PC, Ferreira DR, Cardoso JMP (2010) Preprocessing Techniques for Context Recognition from Accelerometer Data. Personal Ubiquitous Computing 14(7):645–662
15. Fischler MA, Bolles RC (1981) Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. Communications of the ACM 24(6):381–395
16. Floch J, Hallsteinsen S, Stav E, Eliassen F, Lund K, Gjorven E (2006) Using Architecture Models for Runtime Adaptability. IEEE Software 23(2):62–70
17. Garlan D, Cheng SW, Huang AC, Schmerl B, Steenkiste P (2004) Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure. IEEE Computer 37(10):46–54
18. GNU (last visited in January 2013) GCC – The GNU Compiler Collection. `http://gcc.gnu.org/`
19. González S, Cardozo N, Mens K, Cádiz A, Libbrecht JC, Goffaux J (2010) Subjective–C: Bringing Context to Mobile Platform Programming. In: Proceedings of the 3rd International Conference on Software Language Engineering (SLE'10), Springer, LNCS, vol 6563, pp 246–265
20. Harris C, Stephens M (1988) A Combined Corner and Edge Detector. In: Proceedings of the 4th Alvey Vision Conference, pp 147–151
21. Hopcroft J, Motwani R, Ullman J (1979) Introduction to Automata Theory, Languages, and Computation. Addison-Wesley
22. Kamina T, Aotani T, Masuhara H (2011) 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,

pp 253–264
23. van Kasteren T, Noulas A, Englebienne G, Kröse B (2008) Accurate Activity Recognition in a Home Setting. In: Proceedings of the 10th International Conference on Ubiquitous Computing (UbiComp '08), ACM, pp 1–9
24. Kell S (2008) A Survey of Practical Software Adaptation Techniques. Journal of Universal Computer Science 14(13):2110–2157
25. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, marc Loingtier J, Irwin J (1997) Aspect-Oriented Programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Springer, pp 220–242
26. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W (2001) An Overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), LNCS, vol 2072, Springer, pp 327–354
27. Marwedel P (2010) Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems. Embedded Systems, Springer
28. McConnell S (2004) Code Complete. DV-Professional, Microsoft Press
29. Mernik M, Heering J, Sloane AM (2005) When and How to Develop Domain-Specific Languages. ACM Computing Surveys 37:316–344
30. Mikalsen M, Floch J, Paspallis N, Papadopoulos G, Ruiz P (2006) Putting Context in Context: The Role and Design of Context Management in a Mobility and Adaptation Enabling Middleware. In: Proceedings of the 7th International Conference on Mobile Data Management (MDM'06), pp 76–83
31. Oracle (last visited in January 2013) Java Programming Language Compiler. `http://docs.oracle.com/javase/1.4.2/docs/tooldocs/windows/javac.html`
32. REFLECT Consortium (2009) Rendering FPGAs to Multi-Core Embedded Computing (REFLECT) – Technical Report about Application Requirements for Reconfigurability and Hardware Templates. Tech. rep., Deliverable D1.5 – FP7 THEME ICT-2009-4
33. REFLECT Consortium (2009) Rendering FPGAs to Multi-Core Embedded Computing (REFLECT) – Technical report of applications delivered by Honeywell. Tech. rep., Deliverable D1.2 – FP7 THEME ICT-2009-4
34. REFLECT Consortium (2009) Rendering FPGAs to Multi-Core Embedded Computing (REFLECT) – Technical report on Generic Architectures and Reconfigurable Schemes. Tech. rep., Deliverable D2.4 – FP7 THEME ICT-2009-4
35. Sánchez P, Jiménez M, Rosique F, Álvarez B, Iborra A (2011) A Framework for Developing Home Automation Systems: From Requirements to Code. Journal of Systems and Software, Elsevier 84(6):1008–1021
36. Santos AC, Diniz PC, Cardoso JM, Ferreira DR (2011) A Domain-Specific Language for the Specification of Adaptable Context Inference. In: Proceedings of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC'11), IEEE Computer Society, pp 268–273
37. Santos AC, Cardoso JMP, Diniz PC, Ferreira DR (2013) Specifying Adaptations through a DSL with an Application to Mobile Robot Navigation. In: Leal JP, Rocha R, Simões A (eds) 2nd Symposium on Languages, Applications and Technologies, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, OpenAccess Series in Informatics (OASIcs), vol 29, pp 219–234
38. Spinellis D (2001) Notable Design Patterns for Domain-Specific Languages. Journal of Systems and Software 56(1):91–99
39. Spinellis D, Guruprasad V (1997) Lightweight Languages as Software Engineering Tools. In: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL'97), USENIX, pp 6–6
40. Tartler R, Lohmann D, Scheler F, Spinczyk O (2010) AspectC++: An Integrated Approach for Static and Dynamic Adaptation of System Software. Knowledge-Based Systems 23(7):704–720
41. Zhuang Z, Kim KH, Singh JP (2010) Improving Energy Efficiency of Location Sensing on Smartphones. In: Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10), ACM, pp 315–330