

Chapter 8

Engineering Adaptive Embedded Software: Managing Complexity and Evolution

Kardelen Hatun, Arjan de Roo, Lodewijk Bergmans, Christoph Bockisch,
and Mehmet Akşit

Abstract Software plays an increasingly important role in the development of electronic systems. In particular, software is used to control the behaviour of systems in advanced ways that enable system features that would not be feasible otherwise. Making such an embedded system adaptive can improve its performance in certain situations, or extend its applicability to a broader range of situations. In this chapter we explain why this is the case, and how adaptivity can provide a competitive advantage. However, realising and maintaining adaptive embedded software brings its own challenges, sometimes even so prohibitive that the benefits of adaptivity are given up. We explain how adaptivity compromises the ability to manage software complexity and the ability to maintain evolving embedded software. To improve on this, we present our approach of a systematic method towards the development of adaptive embedded software. Two case studies explain two concrete applications of this approach: The first application is a method and corresponding tool set for developing flexible (adaptive) schedulers. It is shown how to use this method to develop application-specific schedulers in a modular way, exploiting a domain-specific language for concisely describing schedulers. We demonstrate that evolving requirements can be handled conveniently. The second application is a method that supports the development of Multi-Objective Optimisations, especially for physical control problems. We discuss the software engineering challenges involved in developing such systems, and explain the various steps and domain-specific languages of the method. Then we illustrate how this method was applied to an industrial case.

K. Hatun (✉) • A. de Roo • L. Bergmans • C. Bockisch • M. Akşit
Software Engineering group, Faculty of Electrical Engineering, Mathematics and Computer
Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands
e-mail: k.hatun@ewi.utwente.nl; a.j.deroo@ewi.utwente.nl; l.m.j.bergmans@ewi.utwente.nl;
c.m.bockisch@ewi.utwente.nl; m.aksit@ewi.utwente.nl

8.1 Motivation

As the amount of software involved in electronic systems grows – millions of lines of code are not unusual – managing the complexity of the software is a serious concern to development organisations. Moreover, embedded software is typically long-lived, and thus has to be maintained; such software also usually evolves over time: Customers more and more expect the feature set of software-based systems to grow with new software updates.

Making an embedded system *adaptive* can improve its performance in a broader range of situations. In the next section we will explain why this is the case, and how adaptivity can provide a competitive advantage. In this chapter we will explain what the challenges are in creating maintainable, adaptive embedded software, present the approach we have taken to counter these challenges; and we present two case studies that demonstrate this approach. These case studies have taken place in the context of the development of software for professional printers.

The first case study addresses adaptive behaviour with respect to task scheduling. A scheduler aims to optimise system behaviour by ordering tasks in the best possible way, considering given goals and constraints. They are typically tightly integrated with the rest of the system and therefore difficult to maintain or even replace.

The second case study deals with control of physical (sub-)systems; in particular, control problems in a continuous domain. Optimising control typically involves making trade-offs between multiple optimisation targets, which may even change at run-time. For example, the user decides for certain print jobs to value speed over quality or energy consumption, but for the next job image quality is be considered more important.

8.1.1 *Adaptivity Provides a Competitive Advantage*

Adaptive control can provide a competitive advantage by enabling a system to achieve better performance with the same hardware characteristics, only through better *control*. Our scope is control problems with multiple parameters, which may vary during the operation of the system. There are several types of parameters; some of those are set to achieve the desirable behaviour, some are a reflection of the context, such as environmental conditions, peer systems, and user requests.

We base our terminology used in the remainder of this chapter on the terms introduced in Sect. 2.3.2. All parameters, i.e. *decision variables* and *dependent variables*, are subject to *system constraints* for proper operation of the system: For example, in a printer, the speed of paper transportation must be limited, because at too high speeds, paper handling will fail. But also, the speed may be constrained, depending on parameters like the degree of paper heating, and the type of image being produced.

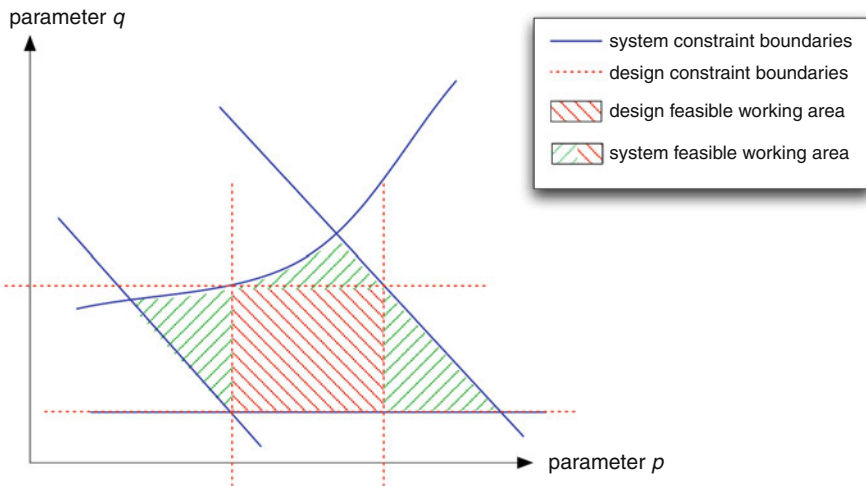


Fig. 8.1 Illustration of the constrained working area for two parameters

Figure 8.1 illustrates the space of possible system parameters through the example of a control problem with two parameters, p and q , which may vary independently. The solid lines designate the upper and lower boundaries of system constraints. To avoid that the system will operate outside the valid boundaries, typically the involved parameters are constrained at design-time to values where this can be guaranteed: This is shown by the dashed *design constraint* boundaries in the figure. These define a rectangular design working area where all constraints (both design constraints and system constraints) are guaranteed to be met.

Figure 8.2 shows that with the same system constraint boundaries, it is possible to define another design working area, where the parameters get different design constraints. In this case the viable range for parameter q is extended (allowing it to reach higher values), thereby substantially reducing the viable range for parameter p . This alternative working area can be defined either as a design-time engineering decision, which defines the characteristics of the particular system. Alternatively, systems can be operated in other *modes*, where each mode defines its own designed working area. This is a common practise, that can have a negative impact on the control software: Assume the control software is designed such that at many decision points in the software, the current mode will influence the behaviour. Then in all these locations it has to be checked first in which mode the system is operating, and different control actions are executed depending on the result. It is therefore important (and there are viable techniques for this, often based on explicit state machine models) to pay explicit attention to structure such behaviour; otherwise the software becomes easily very complex, error-prone, and difficult to understand and maintain.

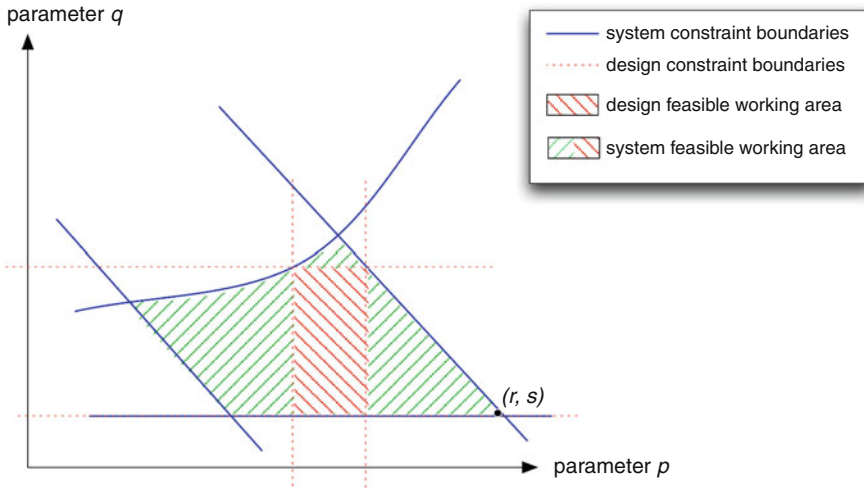


Fig. 8.2 Different design-time trade-off of the design working area

However, even when working with multiple modes or viable ranges, this still leaves parts of the system's feasible working area unexploited, because they cannot be reached while guaranteeing all design constraints are satisfied. Consider for example the point (r, s) in Fig. 8.2: this point lies within the system constraint boundaries, but it can only be reached by relaxing the design constraints on parameter p . Parameter q then needs to be controlled carefully, as it can no longer vary arbitrarily, even within its design constraints. This does require making a *trade-off* between parameters p and q ; here making the trade-off to keep the value of parameter q low, thereby enabling a higher value for parameter p . Note that it is possible that some parameters cannot be controlled, but are *dependent variables*.

As a result, the viable operational range of the system has been extended; much higher values for p can now be reached; if p is, e.g. productivity, or quality, or some other feature for which high values are attractive, a more competitive system (with "better" specifications) has been obtained. To be clear: the system has only become "better" by allowing it to make trade-offs, such as delivering higher quality at lower speeds, whereas without adaptivity, the engineering efforts would be focused, e.g. on improving the quality without sacrificing speed or other qualities.

Finally, we would like to point out that the above situations describe snapshots during the development life-cycle and life-time of a system: as a system is refined and improved, the situation will inevitably change. Accordingly, system constraints may change, or new system constraints may be added. Figure 8.3 illustrates this, sketching entirely different system working areas and designed working areas.

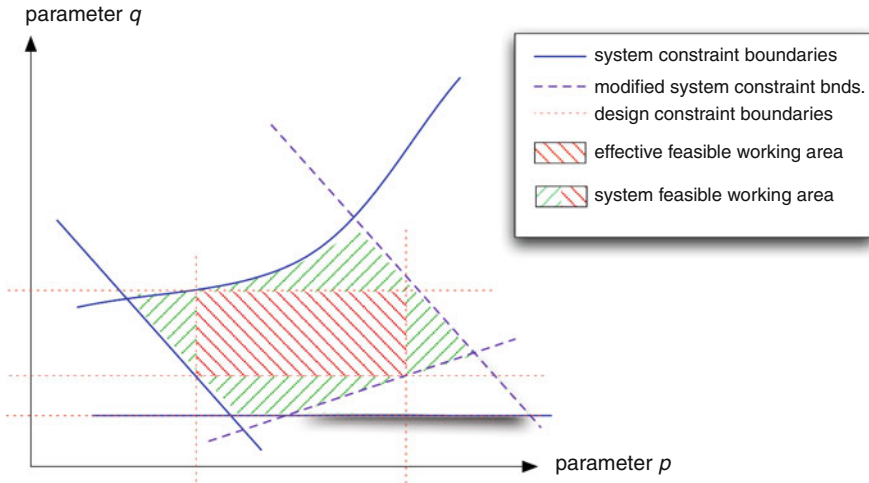


Fig. 8.3 Evolution of the constrained working area

8.1.2 The Implementation of Adaptivity Affects Software Quality

It is commonly known that there are virtually no limitations to the kinds of behaviour that can be implemented in software: Any behaviour that can be precisely specified, in a mathematical formula, or as an algorithm, can be implemented. The only *theoretical* limitations are the required resources, such as computing power, available memory, or available time.

However, there are pertinent *practical* limitations to the ability to realise and maintain software, in particular for large and complex systems. The major challenges of software development are:

- *Managing complexity.* How can we decompose the software into manageable parts, which can be developed and understood independently? How can we keep the number and complexity of dependencies between those parts manageable? And how can we ensure that those parts can be put together such that the resulting whole works correctly?
- *Managing evolution.* Typically, 75–90% of software development effort and costs are spent *after* the initial delivery of a system; hence the ability to enhance, extend, revise, and correct the behaviour of a software system is crucial.

The answers of software engineering to these challenges are manifold [28, 29], but these are the typical qualities that are strived for:

- *Modularity* to achieve locality of change,
- *Comprehensibility* to be able to understand the behaviour of the software, and

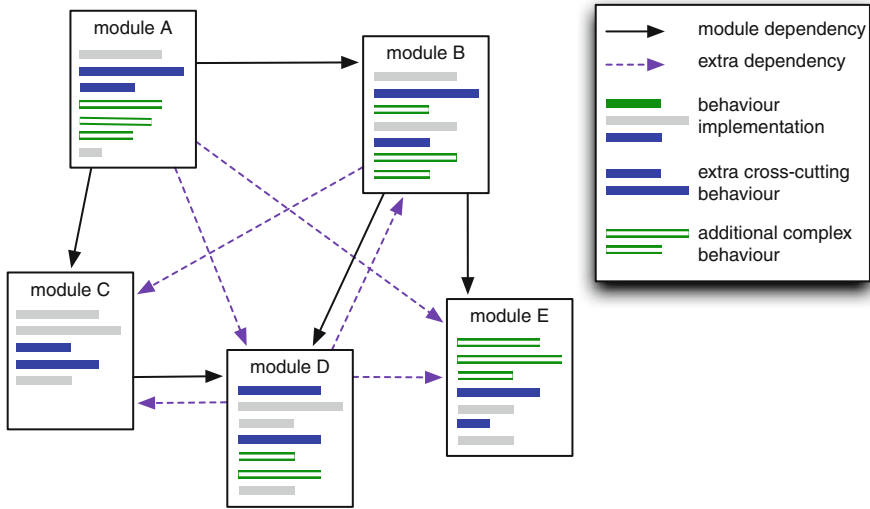


Fig. 8.4 Illustration of the impact of (I.) additional dependencies, (II.) additional cross-cutting behaviour and (III.) more complex behaviour on the structure of the software

- *Evolvability* to be able to modify behaviour without invasively manipulating the previous (and potentially still relevant) version.

We will now discuss three related reasons (also illustrated in Fig. 8.4) why the realisation of adaptivity in software conflicts with these quality goals.

I. Adaptive Behaviour Causes Increased Dependencies between Modules

Software modules aim at addressing particular concerns, such as managing a specific part of the hardware (motors, heaters, inkjet printheads, etc.), implementing different functions (paper transport, paper heating, stapling, etc.), or implementing specific applications (such as various modes of printing, scanning, and copying). A key factor in the manageability of software is the number (and type) of dependencies between modules.

Adaptive control and dynamic optimisation techniques work by carefully monitoring the system state, and accordingly setting or adjusting certain parameters, or selecting appropriate policies. Typically, to make more effective optimisations, more information about the context and state of the system must be taken into account. Similarly, more effective optimisations may adjust a larger set of parameters (of more actuators). As a result, more precise or effective control and optimisations are achieved usually at the cost of causing more dependencies between software modules:

- More sensors and actuators are involved in controlling certain behaviour, including those that are in totally different subsystems (represented by other modules).
- More information about the state of the (software) system is acquired, also from other software modules.
- Information about the activities and plans of other modules may be required to decide upon the best possible settings or optimisation policies.

Note that there can also be *other* reasons for many additional dependencies, such as bad decomposition choices.

II. Adaptive Behaviour is Cross-Cutting

Besides the fact that adaptive controllers have to interface with a lot of modules, the adaptive behaviour itself is typically *scattered* (either distributed, or replicated) over multiple modules; examples are:

- Gathering information from multiple locations (sensor data, state information, activities and plans), or
- Adjusting the values, policies, plans, or activities of multiple other modules, so that these behave in accordance with the desired (adaptive) behaviour.

The implementation of such scattered behaviour is typically tightly interwoven (or *tangled*) with the implementation of the other modules; behaviour that is both scattered and tangled, is also referred to as *cross-cutting behaviour* [14]. Examples involve monitoring the energy usage of the various parts of a system, or coordinated control of multiple printer units (paper feeder, printer, stapling units, sorters, etc.). A result of cross-cutting, adaptive behaviour is that its implementation and maintenance involves adjusting the implementation of multiple modules, which can be painful, but especially bears risks to the stability and reliability of the system.

III. Implementing Adaptivity Results in Higher Complexity

The most simple special case of control is to set parameters to constant values – i.e. to actually not adaptively control–, which clearly has a low complexity. In more powerful controllers, optimisation algorithms set values dynamically, depending on the actual context. Implementing behaviour that is different in many different circumstances, depending on state information, can cause a large increase in the complexity of the code (for example due to many conditionals and control statements). A direct result of the increased dependencies between modules, as well as of scattered and tangled behaviour (as explained under I. and II.) is a much higher complexity of the structure of the software. This will generally result in software of reduced quality, which is certainly more difficult to manage and maintain, resulting in a larger time-to-market and higher development and maintenance costs.

8.1.3 Overview of Evolvability in Embedded Software

A number of modelling approaches have been introduced to manage the complexity of embedded control software [17, 30]. The common feature among them is that they support expressing reactive behaviour using state transition diagrams (STD). The work on UML [25] and ObjecTime/Rose RealTime [27] has focused more on how to structure STD-based specifications within the context of an object-oriented design.

A key feature of the above approaches is that they lead to implementations whose structure is dominated by the STDs that specify the reactive behaviour. Either the program as a whole is dominated by this STD structure, or at least the implementation of some modules within the program is dominated by it. This is clearly suitable for those parts of the system where that reactive behaviour is actually dominant, but this is not always the case: Embedded system design involves many more concerns, including physical (continuous-time) models, optimisation problems, activity-scheduling issues, resource management, and many more.

Essentially, STDs are a notation, or language, specifically designed for the domain of event-based, reactive behaviour. We will argue that, next to STD-based development methods, alternative domain-specific approaches are needed: They allow the development of solutions in their respective domains such that they retain the modularity of the problem domain and hence become much easier to develop, maintain, and evolve. One of the key challenges then is to combine these different modelling techniques into well-integrated systems.

It is a key issue of the approach presented in this chapter to separate specific parts of the system, while being able to compose these modules into an integrated system. This compositional power distinguishes our approach from the majority of other work in the area of Domain-Specific Languages (DSLs) [24], Model-Driven Engineering [21], and Generative Programming [7].

8.2 Approach

In the remainder of this chapter we will present a systematic approach of the development of adaptive software. The aim of this approach is to *enable the – efficient – development of adaptive software without sacrificing the quality of the software system*. In this section we present the general approach, the subsequent sections show two concrete cases that demonstrate the application of the approach to two different domains:

- Section 8.3 discusses the development of flexible task scheduling. This section extends an earlier publication of our approach for developing a scheduler workbench [18].
- Section 8.4 presents our method, called the MO2 method, for the development of (adaptive) multi-objective optimisation of system behaviour. This method is

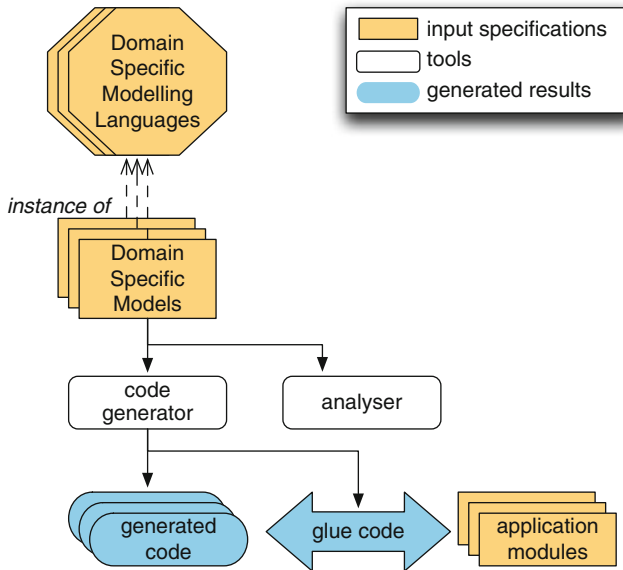


Fig. 8.5 The core ingredients of our approach towards designing adaptive software

illustrated by a case study. More details about this method and the case study can be found in [11, Chaps. 4 and 5] and [10], which improves over an earlier version [9].

Our approach is based on the model-driven paradigm; we propose to develop, or adopt, models that focus on the domain that needs to be made adaptive, for example the task scheduling domain, or the physical domain. These are so-called *Domain-Specific Models* (DSMs). We have several motivations for this:

1. Domain-specific models are closer to the concepts and way of thinking of domain experts, and are usually more concise and expressive with respect to the domain: This makes it easier to manage their complexity.
2. Developing models for a system's adaptive behaviour causes a separation of the domain-specific elements from other concerns; this creates a modularity in the model that can reduce its complexity and improve maintainability (among others as a result of locality of changes).
3. One of the goals of models is to focus on expressing what needs to be done, rather than how; the latter can then be expressed separately, as part of the process of going towards an implementation.

Domain-specific models need to be expressed in a domain-specific (modelling) language (DSL). If for a specific domain no suitable modelling languages are available, they will need to be defined first.

Figure 8.5 provides an overview of the approach, and shows that one or more domain-specific models, expressed as instances of domain-specific modelling

languages, serve as input to some tools: One of the tools is an *analyser*, which exploits the knowledge about the domain to analyse the models, e.g. for correctness constraints, feasibility, and performance criteria. Another tool is a *code generator*, which takes the domain-specific models as input, and uses those, together with knowledge about the domain, such as specific techniques, algorithms, and solutions, to generate code that implements the domain-specific models.

The domain-specific models normally describe only a part of the system, whereas the remainder is implemented by *application modules*, which are typically implemented in a General Purpose Language (GPL), such as Java, C#, C++, etc. Hence application modules must be integrated (or at least they must interface) with the generated code from the domain-specific models; similarly, the code from multiple domain-specific models must be integrated: This is the purpose of the glue code, which is partially created by the code generator and partially hand-crafted to match the target system. As a result, the application modules, as well as the domain-specific models, remain independent of each other, regardless of how they are integrated, and the various modules can evolve mostly independently; all code that is needed to integrate the generated component with the rest of the system is localised in the glue code.

The above description of our approach omits many details; a more complete picture is provided in Fig. 8.6. In this figure yellow clouds represent knowledge that is involved in the process explicitly: To define an appropriate domain-specific modelling language, ample knowledge and understanding of the specific domain is required, as indicated by the arrow from “domain knowledge” to “Domain-Specific Modelling Languages”. Each domain-specific model is really an application-specific model, expressed in a domain-specific modelling language. Creating a domain-specific model hence involves both general knowledge of the domain, as well as application-specific knowledge. The application modules are also based on *application* domain knowledge. A special part of the domain knowledge is formed by algorithms and solution techniques from the domain, knowledge of these is used to design the code generator and analyser tools.

A particular characteristic of composing domain-specific models and application modules is that the first tends to inject dependencies into all other modules in the system. Thus, in a direct implementation the code generated from the domain-specific models would be cross-cutting the application modules. To avoid this, the glue code is responsible for establishing the required links, while retaining the modularity property of all generated and application code. This balancing act can be achieved with appropriate tools, which can “weave” for example generated domain-specific code into application code. Weaving is a technology adopted from aspect-oriented programming (AOP) [15]. A *code weaver* tool can operate on the generated code, and does not need to change the source code. Thus, code generated from modules remains separate and modular with respect to application modules, so that the latter can still be properly maintained.

Figure 8.6 explicitly shows the generated executable application, and the output of the analyser (typically reports for the model designer) as distinct documents (the green, rounded rectangles).

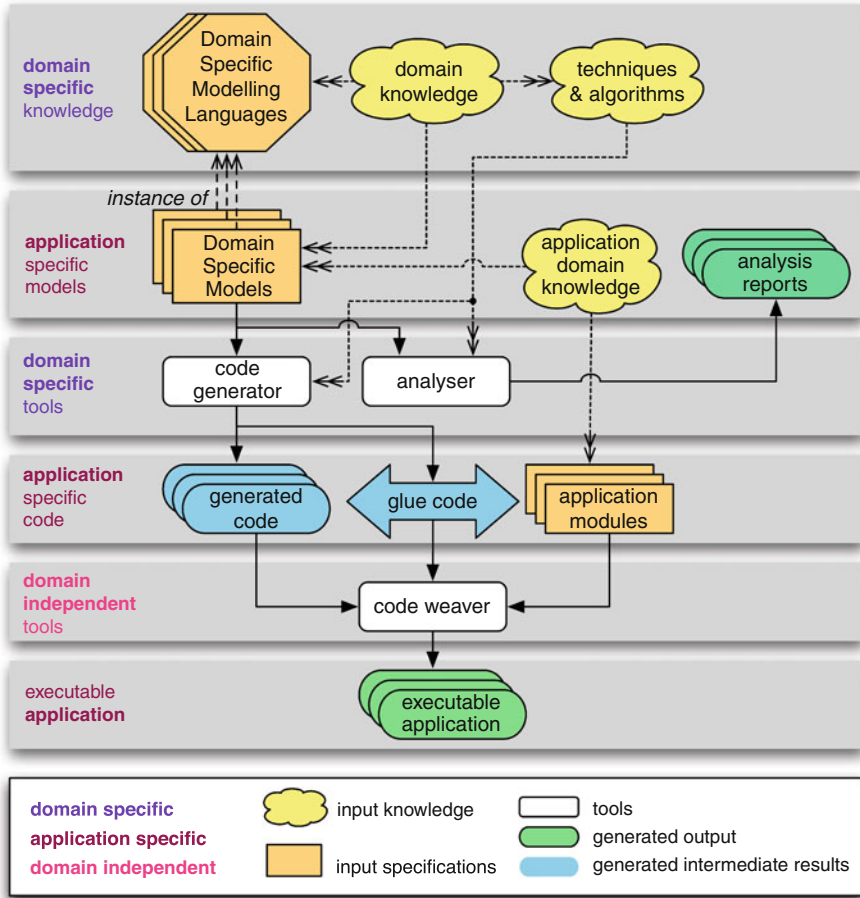


Fig. 8.6 An overview of our model-based approach of designing adaptive systems

We can distinguish several layers in the approach (the grey boxes), each with its distinct purpose and abstraction level; at the top the domain-specific (but application-independent) knowledge is located. This layer serves as input for constructing the domain-specific models and the code generator and analyser tools. The latter two are *domain*-specific tools, but not *application*-specific. The code they generate and the application modules, however, *are* application-specific, as well as the executable application that is the output of the code weaver. A code weaver itself is a generic tool that is independent of any application domain.

In the following sections we will illustrate this general approach with two concrete case studies we have performed within the field of professional printing. Examples of involved domains are physics (thermodynamics, etc.) and planning of tasks for finishing a print job.

8.3 Flexible Task Scheduling with an Automatically Generated Scheduler

One specific domain in which embedded controller software frequently performs optimisation is that of *allocating resources to jobs over a period of time while optimising one or more objectives* [4]. The component performing this optimisation is called a *scheduler*; it has a direct impact on resource management and, as a consequence, system performance. It is evident that a scheduler communicates with multiple system components, therefore its implementation is likely to be highly coupled. The design issues attached to a system scheduler can be categorised in two parts:

1. *Specification of scheduler components.* A scheduler component consists of (1) information about the base system in the form of data structures, (2) communication interfaces with the base system, and (3) a *scheduling algorithm* that can solve the scheduling problem according to the objectives of that base system. The concrete definition of these elements may vary greatly depending on the application area: The scheduling requirements (on information, communication, and objective) of a safety-critical system are very different from those of an operating system. The challenge lies in expressing system characteristics and the desired scheduling algorithm at the same level and defining the scheduling algorithm with the tasks and resources of the base system.
2. *Non-intrusive integration of schedulers.* Recalling the scheduling definition in the first paragraph, it is obvious that a scheduler has to constantly communicate about the current state of the resources and tasks, and at the same time must enforce constraints which may be predefined or introduced on-the-fly. These aspects alone pose a challenge in creating expressive and flexible interfaces for the scheduler. However there is another challenge which is the by-product of software/system evolution. Since schedulers involve system-specific parts, when systems evolve it is necessary to alter or – depending on the amount of change performed in the system – completely replace the scheduler. This is difficult given the scheduler’s tight integration within the system software. This problem calls for a non-intrusive integration mechanism between the scheduler and the system and a cost-effective way of altering/replacing the scheduler implementation.

8.3.1 Scheduling Workbench

We have developed a *Scheduling Workbench*, to *model and integrate schedulers into existing applications* (an overview is depicted in Fig. 8.7). We have *not* developed new scheduling algorithms, but provide a means to easily integrate schedulers into a system. Our “domain-specific scheduling language” acts as a meta-model and the entities involved in the scheduling process as well as the interface between them

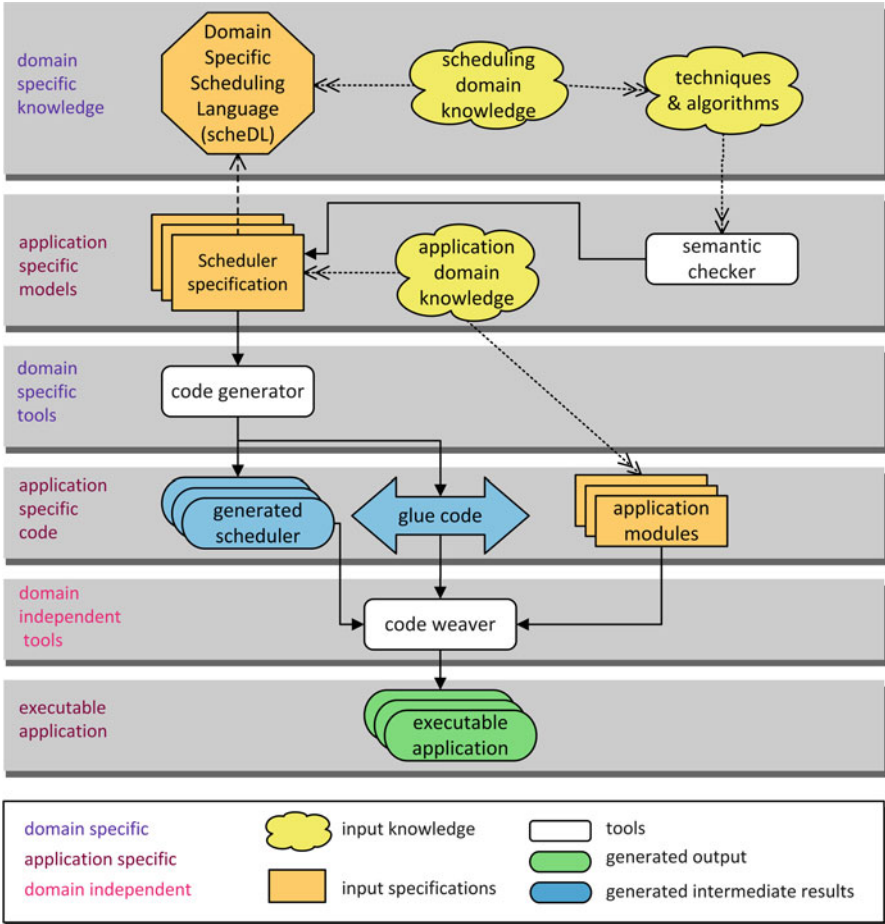


Fig. 8.7 Our approach instantiated for the scheduling domain

and a scheduling algorithm can be defined in this meta-model. Thus, “generated scheduler” refers to code which is generated from the “scheduler specification” and which interfaces with an implementation of a scheduling algorithm.

The first step of development of such a workbench is a comprehensive domain analysis. Domain analysis is the process where the fundamental domain concepts and their variability are determined. In domain modelling these concepts and their relationships are expressed in a model. In Fig. 8.7 this model is shown as *scheduling domain knowledge*. From this model we have derived a grammar for a scheduling domain-specific language, shown as an octagon in Fig. 8.7 connected to scheduling domain knowledge.

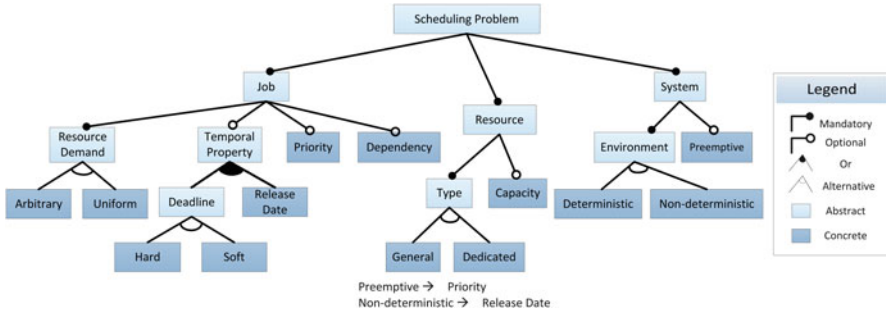


Fig. 8.8 A simplified version of the feature model for scheduling

8.3.1.1 Domain Analysis and Modelling

For domain analysis we have analysed commonalities and variabilities of scheduling approaches. We started by extracting the core domain concepts by means of a literature study and identifying the building blocks of the scheduling domain. After finding the core concepts, we have analysed the variations of these concepts. We have expressed our findings about the domain using feature modelling [19].

In Fig. 8.8 a feature model for scheduling is shown (the full model can be found in [18]). In the feature model notation every domain concept is mapped onto abstract features and their variations are mapped onto concrete features. For example, if we look at the **deadline** feature, we see that a deadline can be either **hard** or **soft**. It is also possible to define cross-tree constraints as logical expressions, shown in Fig. 8.8 below the feature tree. This is additional semantic information about the domain which is used to implement the *semantic checker* shown in Fig. 8.7.

8.3.1.2 Domain-Specific Language Design and Code Generation

Following the advice of Mernik et al. [24], we have designed a DSL for the scheduling domain (the “scheduling domain-specific language”, **schedL**) using the feature model presented in the previous subsection. We have used the feature model as the abstract syntax and turned it into the concrete grammar of our language, especially by defining keywords. In our approach we chose to generate *general-purpose language (GPL)* code from our DSL, which is called generative programming [7]. This method allows a user to program on a higher abstraction layer, and to obtain code in widely supported and robust programming languages (e.g. Java, or C++) at the same time. The *code generator* knows how to process a *scheduler specification* and how to incorporate it with *application domain knowledge* to obtain GPL code (cf. the arrow from “application domain knowledge” to “code generator” in Fig. 8.7).

Let us illustrate the benefits of our approach by explaining how we handle the two main challenges presented in Sect. 8.3. The first one expresses the requirements of a scheduling problem using the concepts found in the scheduling domain. At the end of the domain analysis and modelling phase, we have a model of the scheduling domain, which includes core concepts, their variations and the relationships of those concepts. From this model we have created a grammar for `scheDL`. Using this language we can define `jobs`, `resources`, and `schedulers` with concise domain-specific language constructs.

The block structure syntax of `scheDL` promotes readability and makes it accessible even to non-programmers. Using a DSL we are able to describe the scheduling requirements and a scheduler concisely. This is also useful for maintainability since altering the scheduler only requires changing the short specification file written in `scheDL`.

The second challenge mentioned in Sect. 8.3 concerns a scheduler's tight integration with a system. In `scheDL` we offer language constructs to model behavioural interactions between the system and the scheduler in the form of event declarations. During code generation these event declarations are turned into software components of a special kind, called *aspects*. The language mechanism of aspects enables to compose the behaviour of aspect components with the events of other components without intrusively modifying them to make the events explicit in the code.

8.3.2 Example Cases

We have tested our Scheduling Workbench on the `DemoPrinter` application, which has been developed in Java and simulates a professional printer. Figure 8.9 shows the static structure of this application; for brevity we have left out some utility classes from this view. In the following, we demonstrate our workbench by applying to three example cases. The examples are written in `scheDL`¹; the examples are (1) replacing the existing scheduler of the demo printer, (2) adding a new hardware component to be scheduled, and (3) supporting multiple scheduling policies with the new scheduler.

8.3.2.1 Replacing the Scheduler in Legacy Code

In this example case, we replace the scheduler of the `DemoPrinter` using the Scheduling Workbench. The first step is to write a `scheDL` specification describing the kind of jobs and resources exist in the system. We also define a scheduler in the specification. In Listing 8.1 the definition of a `Job` is shown. A `Job` with the

¹In listings, `scheDL` language keywords are shown in bold.

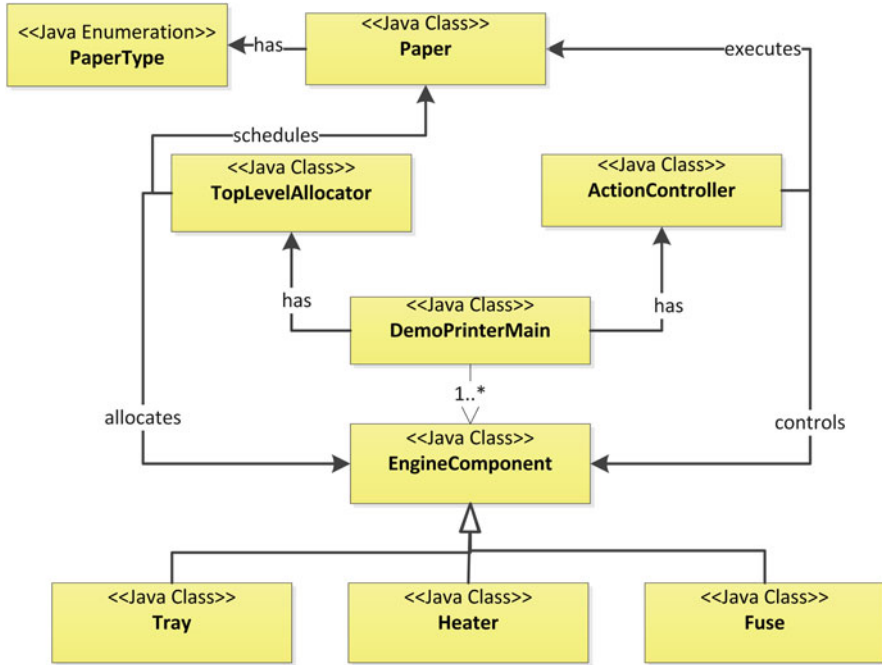


Fig. 8.9 The static structure of DemoPrinter

name `Paper` is defined. This job has an integer, the so-called release date, which is defined as the time a job becomes available. It is also possible to define properties like deadline, or priority. In the definition we also reference `Executor` resources as a `DEMAND`, which means: In order to be able to execute a `Paper` job, we need a `Tray`, a `Heater`, and a `Fuse`. The `newpath` property is a *path declaration* which defines an order between the demanded resources. The scheduler has to consider processing times of the involved component such that paper jams are avoided and it can be ensured that components are ready when the paper arrives.

Listing 8.1 Job definition.

```

Job Paper{
    RELEASE release_date Integer
    DEMAND Tray
    DEMAND Heater
    DEMAND Fuse
    newpath
}
    
```

```

PathDeclaration newpath{ Tray->Heater->Fuse }
    
```

In Listing 8.2 the definition of an `Executor` is shown. Every structure defined in `schedL` must have a unique name. Here we have defined the executor `Tray` as an executing resource with single access, which means it can only execute one job at a time. The same definition is made for `Heater` and `Fuse`.

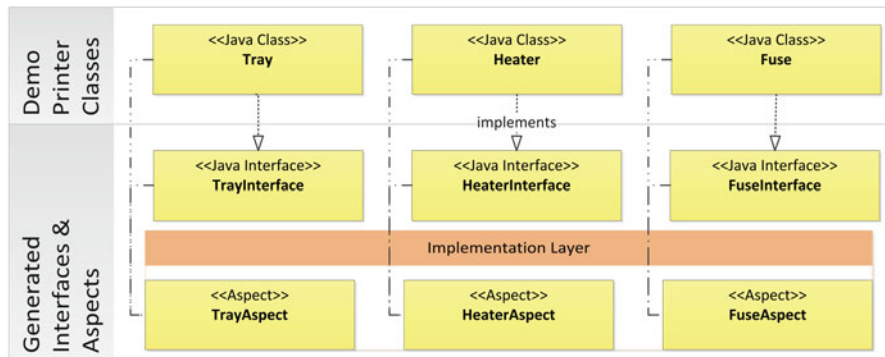


Fig. 8.10 Mapping of system classes and generated interfaces

Listing 8.2 Executor definition.

```

Executor Tray {
    ACCESS single
}

```

The next step is to define the scheduler which will include references to the structures defined before. **Scheduler** is the central structure in `schedL`; this is, only the structures referenced by the scheduler will be generated. Therefore, it is possible to define multiple jobs or resources in the same file even if they are not used. In Listing 8.3 the scheduler definition for our example is shown. This scheduler definition basically describes a scheduler which assigns the three resources to sheets applying the *First-Come First-Served* policy.

Listing 8.3 Scheduler definition.

```

Scheduler Myscheduler {
    PolicyDefinitions {
        builtin FCFS
    }
    JobReference {
        Paper
    }
    ResReference {
        Tray
        Heater
        Fuse
    }
}

```

This is all the code needed for defining a scheduler for the `DemoPrinter` example. The jobs and resources are automatically mapped to classes with the same name in the `DemoPrinter` application. Figure 8.10 illustrates this mapping for the engine components of the printer. Printer classes are extended by generated interfaces, but this relationship is encapsulated in aspects (the implementation layer shown in the figure). This way we can add behaviour to the `DemoPrinter` without altering its implementation.

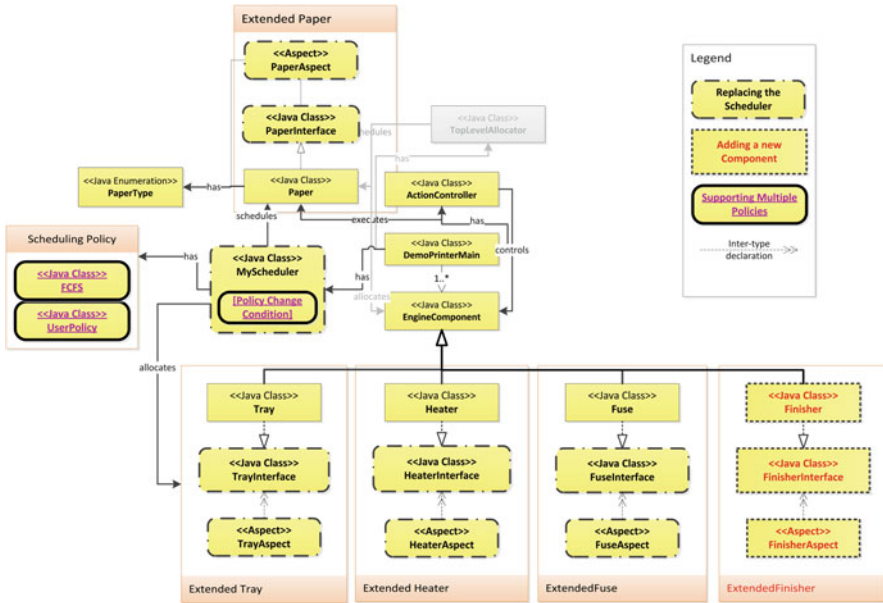


Fig. 8.11 The final view of the system after all example cases

Given a `schedL` specification, the Scheduling Workbench outputs code consisting of the user-defined components, the scheduler and its helper classes, a library of scheduling policies declared in the `schedL` specification, and abstract classes that capture domain concepts like job, resource, etc.

Since the `DemoPrinter` application already contains a scheduler, the user has to find and disable its code after the generated code is imported, e.g. by removing the `TopLevelAllocator` class in the example. Then the new scheduler can be added to the base system, ready to work. A concise and fixed interface is provided with the scheduler to make it intuitive to use in the base system. The system’s state after making this change can be seen in Fig. 8.11. If a scheduler in an application is developed in the Scheduling Workbench in the first place, switching the scheduler becomes even easier: Only the `schedL` specification has to be replaced.

8.3.2.2 Adding a Component

In the second example we will illustrate what happens if a new component, namely a `Finisher`, is added to the system. In order to modify our scheduler software we only need to add a few lines of code to our `schedL` specification. The applied changes are shown in Listing 8.4.

Listing 8.4 Extra code for adding a component.

```
Scheduler Myscheduler{
Job Paper{
    ...
    DEMAND Finisher
}
Executor Finisher{
    ACCESS single
}
PathDeclaration newpath{
    Tray->Heater->Fuse->Finisher
}
Scheduler Myscheduler{ ...
    ResReference{ ...
        Finisher
    }
}
```

We only had to add five lines to our `schedL` specification to make sure that a new class for `Finisher` and the necessary dependencies are generated connecting this class to the scheduler. After altering the specification the scheduler code needs to be regenerated and imported into the system code. Then the system is able to use the new component. The system's state after making this change can be seen in Fig. 8.11.

8.3.2.3 Supporting Multiple Policies

In the last example case, we add support to `MyScheduler` for supporting multiple policies. This means, the base system can switch policies when a condition changes, for example if the power is low it may choose a power-aware policy. This is important since it enhances system adaptivity greatly. In Listing 8.5 we show what must be added to our `schedL` specification to support multiple policies. Firstly we define a resource called `Power` with limited capacity.

Listing 8.5 Extra code for supporting multiple policies.

```
Resource Power{
    CAPACITY limited
}
Scheduler Myscheduler{
    PolicyDefinitions{
        builtin FCFS
        new UserPolicy
        Condition Policychange{
            resource: Power
            FCFS = "Power.getCapacity() >= 100";
            Userpolicy = "Power.getCapacity() < 100";
        }
    }
    ...
}
```

Secondly, in the `PolicyDefinitions` block we declare two policies, one is First-Come First-Served, provided by the Scheduling Workbench's policy library, hence we distinguish it with the keyword `builtin`. The second one is a *new* policy, meaning it will be implemented/provided by the user. If more than one scheduling policy is declared then a *condition* for policy change needs to be defined.

We can define a condition in `schedL` with the `Condition` keyword, then referencing the structure triggering the policy change we define when each policy is valid. According to the condition defined in Listing 8.5 the FCFS policy will be applied when the available power is more than or equal to 100 W, and `UserPolicy` will be applied when available power is less than 100 W. Figure 8.11 illustrates the system after applying this change.

8.3.3 Conclusion

In these example cases we have demonstrated how to create a scheduler with `schedL`. We have also discussed the possibility to use custom scheduling policies allowing the user to customise certain aspects of their specification offering enhanced expressiveness. The complete list of benefits provided by Scheduling Workbench are:

- Benefit 1. Domain concerns can be expressed intuitively from a higher level of abstraction.
- Benefit 2. Complex domain constraints can be defined and checked; modelling errors are reduced.
- Benefit 3. Software maintainability is increased through concise and understandable DSL code.
- Benefit 4. Software evolution is eased by the underlying rich domain model.
- Benefit 5. Integrated editor and IDE support make programming in DSLs easy.
- Benefit 6. Code generation highly reduces manual labour spent on customising general-purpose tools.
- Benefit 7. Easy system integration of generated code is provided through non-invasive software engineering techniques.

8.4 Multi-Objective Optimisation of System Qualities in Embedded Control Software

As discussed in Sect. 8.1, to optimise the system behaviour with respect to system qualities, the right values for certain parameters have to be selected. Such controllable parameters are often also called *decision variables*. Examples of decision variables in high-end printing systems are the paper transportation speed of the

system and the temperature setpoint of a heating device. Optimising multiple system qualities (or system objectives) by influencing a set of decision variables within the boundaries of the system constraints is known as multi-objective optimisation (MOO) [20].

In current design practise, the control logic implemented in *embedded control software* is decomposed into many different controllers, each controlling a part of the system. Such a decomposition makes the control logic easier to comprehend and maintain, as opposed to, for example, a single (black-box) controller that controls all variables. As the software decomposition usually follows the control decomposition, the different controllers are implemented in several software modules in the embedded control software. Certain system qualities, such as power consumption and productivity, are not controlled by a specific controller, but emerge from the behaviour of the system as a whole. So, if we want to influence these system qualities, we have to manipulate and coordinate many controllers, scattered through the embedded control software. This manipulation and coordination of controllers introduces additional structural complexity within the embedded control software.

To the best of our knowledge, there is a lack of systematic methods to design and implement multi-objective optimisation of system qualities in embedded control software. We have observed that, in practise, state-of-the-art attempts to realise multi-objective optimisation leads to:

- Solutions that are tailored to the specific characteristics of the embedded system, and therefore are inflexible in case the physical system changes or evolves.
- Solutions that are tightly integrated into and coupled with the control software modules, making the embedded control software difficult to comprehend and hard to maintain.
- Solutions that are sufficient but sub-optimal, because the implementation of stronger optimisations would be too complex.

As such, the lack of systematic methods to design and implement multi-objective optimisation in embedded control software leads to higher development and maintenance costs [2]. It may also prevent the implementation of multi-objective optimisation all together, if the performance improvement does not outweigh the reduction in software quality and the increase in development and maintenance costs.

In this section we present the *MO2 method*, a systematic method to design and include multi-objective optimisation and dynamic trade-off making in embedded control software. The MO2 method includes an architectural style to specify and document a multi-objective optimisation solution within the architecture of the embedded control software. The architectural style is supported by techniques, implemented in a tool chain to (1) validate the consistency of the solution, (2) include general optimisation algorithms, and (3) generate code that implements the optimisation algorithm and coordination of the control modules.

8.4.1 MO2 Method Overview

Multi-objective optimisation algorithms try to optimise a given set of utility functions (i.e. objectives) for a given set of decision variables. This type of problem was introduced in works on decision making in economy by Edgeworth [12] in the late nineteenth century. Pareto extended the work with the concept of Pareto optimality [26].

Definition 1 (Multi-objective optimisation problem [6]). A multi-objective optimisation problem exists of the following elements:

- A set of decision variables that can be influenced.
- A set of constraints on the decision variables.
- A set of objectives, expressed as functions of the decision variables.

The solution of a multi-objective optimisation problem is the valuation of the decision variables that satisfies the constraints and provides the (Pareto) optimal value for the objective functions (i.e. there is no other possible valuation for the decision variables that satisfies the constraints and provides a better value for the objective functions).

Note that, if there is more than one utility function, there may not be a single optimal solution for this problem. Instead, the solution is a set of Pareto-optimal points [26].

We developed the MO2 method to design and implement control software with multi-objective optimisation. Figure 8.12 shows an overview of the MO2 method. The MO2 method applies two DSLs: the *MO2 architectural style* and the *SIDOPS+ language* for physical models.

Step 1: The MO2 Architectural Style

The *MO2 architectural style* – i.e. a notation for architectural design as well as a methodology how to structure the architecture – (abbreviated as *MO2 style*) supports architecting control software. We call an architectural model created with the MO2 style an *MO2 architectural model* or *MO2 model*. MO2 models are specialisations of Component-and-Connector models [5]. Besides the different control components and their interfaces, MO2 models include the elements input/output variables, decision variables, constraints, and objective functions. As such, the MO2 style supports the design and documentation of embedded control software that includes multi-objective optimisation functionality.

An MO2 model specifies the architecture of the control software, which consists of software components that implement control logic (i.e. *control components*). The interfaces of these software components consist of input and output variables. The control components are composed into a control architecture by connecting output variables to input variables. Furthermore, an MO2 model specifies which

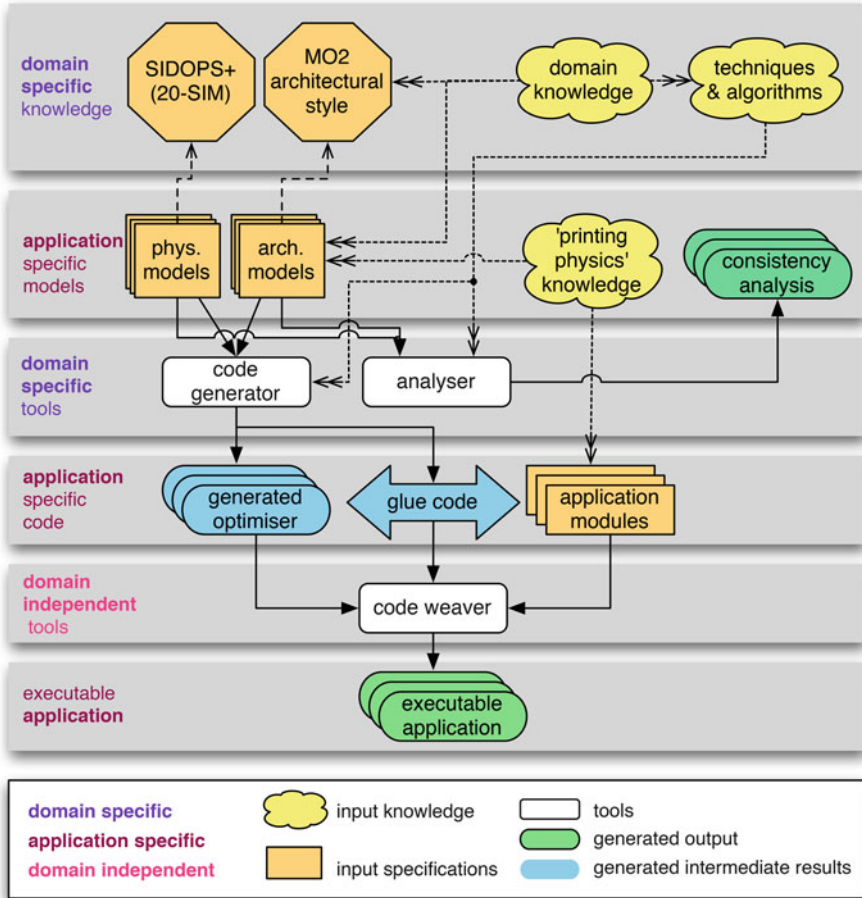


Fig. 8.12 Overview of the MO2 method

variables are decision variables, which variables have constraints (*constrained variables*), and which variables represent the outcome of objective functions (*objective variables*). An MO2 model serves two purposes:

1. It is design documentation of the embedded control software applying MOO.
2. It is an input model to generate an optimiser.

Step 2: Application of the SIDOPS+ Language

The computational logic that is implemented in the software components creates a mathematical relationship between decision variables and input/output variables

used in constraints and objective functions. To be able to analyse a MO2 model and generate an optimiser module, the mathematical relationship between the decision variables and the other variables should be specified. Therefore, the MO2 method provides the possibility to refer to models that specify the computational logic (e.g. control logic, or implemented physical characteristics) of components in the architecture. These models can be specified in any language in which computational logic can be mathematically specified. In our implementation of the MO2 method, we apply the SIDOPS+ language of the 20-sim tool set [1, 3, 22], because of the suitability of this language to model control logic and physical characteristics, which are two common domains of computational logic in embedded control software.

Step 3: Analysis and Code Generation

The MO2 method includes a tool chain taking an MO2 architectural model and the referenced 20-sim specifications (defined in the SIDOPS+ language) as input. The tool chain contains a graphical editor, which is an extension of the ArchStudio 4 tool set [8], to create and edit MO2 architectural models. The *MO2 consistency validator* checks the consistency of the MO2 model. For example, it checks whether each variable that is used in constraints and objective functions has a mathematical relationship with the decision variables. Such a relationship should have been specified using 20-sim models. If the MO2 architectural model is consistent, it can be provided to the *MO2 code generator*, to generate an optimiser module specific for the given architecture and MO2 model. The software modules that implement the basic control architecture are provided to the *MO2 code weaver*. The code weaver introduces the interaction between these software modules and the generated optimiser module by weaving instrumentation code in the software modules. The result is embedded control software that includes multi-objective optimisation functionality.

The following section introduces an industrial case study, which will be used in subsequent sections to illustrate the three steps.

8.4.2 Industrial Case Study

In this section, we present an industrial case study where we have applied our MO2 method to a digital printing system, in particular to the subsystem related to the *Warm Process* of professional printers.² This process, schematically shown in Fig. 8.13, is responsible for transferring a *toner image* to paper: The *paper path*

²A video demonstrating the application of our method in the case study can be found at the Octopus homepage of the University of Twente: <http://www.utwente.nl/ewi/trese/research-projects/Octopus.doc/>.

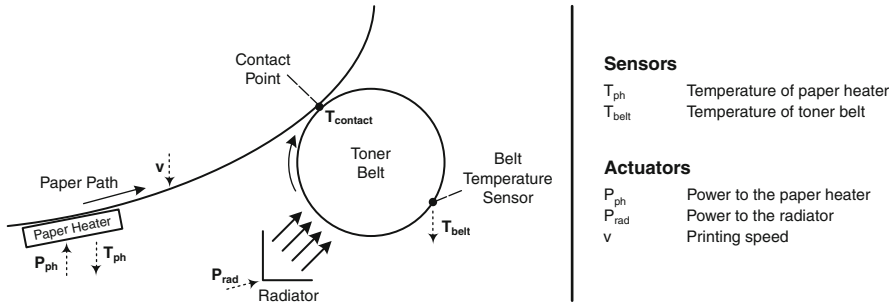


Fig. 8.13 Schematic view of the *Warm Process*

transports sheets of paper and a *toner belt* to transport toner images. For correct printing, the sheets of paper and the toner belt must have a certain temperature at the contact point. Therefore, the warm process contains two heating systems; a *paper heater* to heat the sheets of paper and a *radiator* to heat the toner belt.

8.4.2.1 Multi-Objective Optimisation in the Warm Process

In this case study, engineers aim to introduce the possibility to make run-time trade-offs between the two conflicting objectives *power consumption* and *productivity* of the printing system. They have identified the decision variables that can be used to influence the objectives, the different constraints in the system, and the objective functions:

- **Decision Variables:** Paper transportation speed (v).
- **Constraints:**
 1. $60 \leq v \leq 120$ (Speed between 60 and 120 pages/min).
 2. $P_{rad} \leq 800$ (Maximum power to the radiator is 800 W).
 3. $P_{ph} \leq 1,200$ (Maximum power to the paper heater is 1,200 W).
 4. $P_{total} \leq P_{avail} - 100$ (Total power consumption should not exceed the amount of power that is available to the system, and a margin of 100 W is taken into account).
 5. $40 \leq T_{ph}^{sp} \leq 90$ (Setpoint for the paper heater is between 40 and 90° C).
- **Objective functions:**
 - Minimisation of total power consumption: $P_{total} = P_{ph} + P_{rad}$.
 - Minimise the inverse of the speed (for an increased productivity): $1/v$.

A trade-off between these objectives is made using a weighted trade-off function. Since the trade-off function can change, e.g. due to user requests, this problem is different from single-objective optimisation.

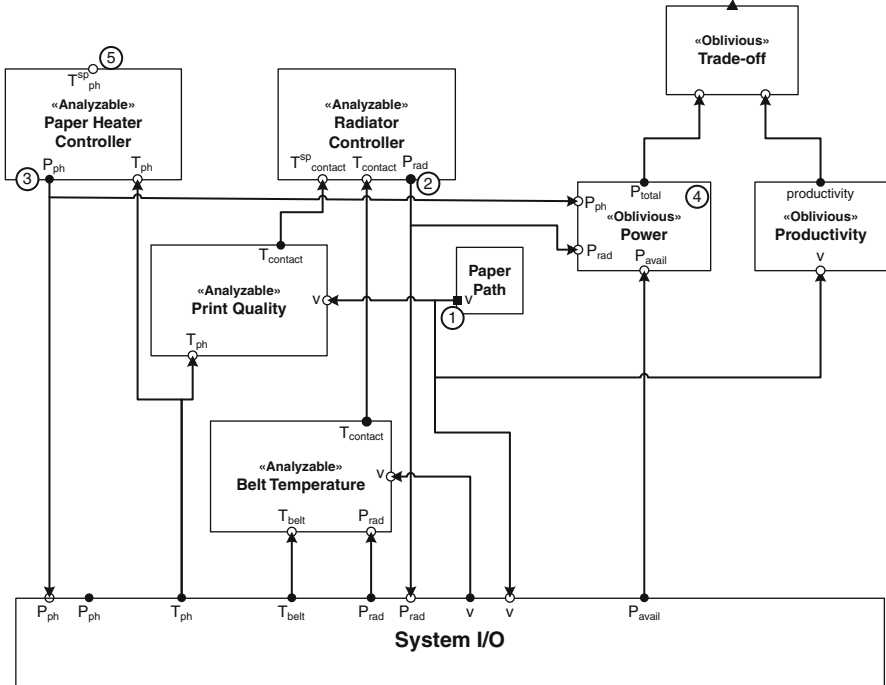







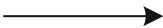

Fig. 8.14 MO2 model of the case study

8.4.2.2 MO2 Architectural Model

Control software implements the control logic for this system. To implement multi-objective optimisation, an MO2 architectural model of the control software has been created. Figure 8.14 shows the graphical representation of this MO2 model. The notation, i.e. our domain-specific modelling language for the MO2 style, is briefly explained in Table 8.1.

Basic control logic is implemented in several components in the control software. The Paper Heater Controller controls the temperature of the paper heater (T_{ph}) to a certain setpoint (T_{ph}^{sp}), by adjusting the power given to the paper heater (P_{ph}). The Radiator Controller controls the temperature of the toner belt at the contact point ($T_{contact}$) to a certain setpoint ($T_{contact}^{sp}$), by adjusting the power given to the radiator (P_{rad}). The value of $T_{contact}^{sp}$ is provided by the Print Quality component. The Print Quality component implements a model of print quality, which relates the value of $T_{contact}$ to the values of v and T_{ph} , to ensure sufficient print quality. Because there is no sensor in the system to measure the value of $T_{contact}$, the value of $T_{contact}$ is derived from the values of T_{belt} , v , and P_{rad} using a physical model of the belt temperature that is implemented in the Belt Temperature component.

Table 8.1 Model elements of the MO2 style notation

Notation	Description
	Component with a <i>stereotype</i> and a <i>name</i> . Here, the stereotype <code>Analyzable</code> indicates that there is a mathematical model of (part of) the semantics of the component. The stereotype <code>Oblivious</code> indicates that the component is only present to support the modelling of the multi-objective optimisation problem; it is not implemented.
	In-port/out-port
	In-port/out-port that represents a decision variable.
	In-port/out-port that represents an objective value.
	Usage of a port: The ports that belong to a component are attached to the edge of the component. The port may be labelled with its <code>variableName</code> .
	Connector
	Informal label indicating constraints attached to the component or port.

The `Paper Path` component provides the default speed (v). The `System I/O` component provides an interface to the sensors and actuators in the system.

The modelled components `Power`, `Productivity`, and `Trade-Off` have the stereotype `Oblivious`, which means that they are not implemented in the control software; these components are modelled to specify parts of the multi-objective optimisation problem. The specification for the example in Fig. 8.14 is as follows:

- The out-port v of the `Paper Path` component is a *decision variable port*, specifying that v is a decision variable.
- The labels 1–5 indicate constraint ports; the corresponding constraints are not shown in the figure.
- The two objective functions for power consumption and productivity are modelled using the `Power` and `Productivity` components. The trade-off function is modelled using the `Trade-Off` component. The out-port of this component is the only *objective port* in the system, indicating that the optimiser should minimise this value.

8.4.2.3 20-Sim Models of Control Logic

To solve an multi-objective optimisation problem, the mathematical relationship between the decision variables, and the constraints and objective functions must be known. The reason for this is that only when this mathematical relationship is known, it can be analysed which values should be selected for the decision variables

to satisfy the constraints and optimise the objective functions. The mathematical relationship between the decision variables and the different constraints and objective functions in an MO2 model is defined by the semantics of the control components in the MO2 model. To analyse the specified MO2 model and to generate an optimiser, this semantics should be available to the MO2 tool chain. Therefore, the components in an MO2 model can reference a 20-sim [1, 3] model that specifies that part of the component's semantics that is relevant for creating the mathematical relationship.

Listings 8.6 and 8.7 show two example 20-sim models (using the SIDOPS+ language [22] of the 20-sim tool set) for respectively the components Radiator Controller and Power. Similar 20-sim models are referenced by the other components with stereotype Analyzable or Oblivious in the MO2 model.

Listing 8.6 SIDOPS+ specification referenced by the RadiatorController component.

```
constants
    real Kp_rad = #some value#;
    real Ki_rad = #some value#;
variables
    real global Tcontact;
    real global TcontactSP;
    real global Prad;
equations
    Prad = Kp_rad * (TcontactSP - Tcontact) + Ki_rad *
        int(TcontactSP - Tcontact);
```

Listing 8.7 SIDOPS+ specification referenced by the Power component.

```
variables
    real global Pph;
    real global Prad;
    real global Ptotal;
    real global Pavailable;
equations
    Ptotal = Pph + Prad;
```

8.4.2.4 Analysis and Code Generation

The MO2 tool chain can analyse whether for all constraints and objective functions there is a mathematical relationship with the decision variables. For example, for the MO2 model in Fig. 8.14 the tooling detects that there is no mathematical relationship between the decision variable v and the constraints on the ports P_{ph} and T_{ph}^{SP} (labelled 3 and 5) on the component Paper Heater Controller. This means that the optimiser cannot influence these constraints, and they are ignored.

The code generator in the MO2 tool chain generates an optimiser component and interaction with the implementation of other control modules. Figure 8.15 shows the structure of the implementation, including the generated optimiser. Note the absence of the Oblivious components Power, Productivity, and Trade-off: The only function of these components was to model the multi-objective optimisation problem. Therefore, they are not present in the implementation of the system; their semantics is part of the generated optimiser component.

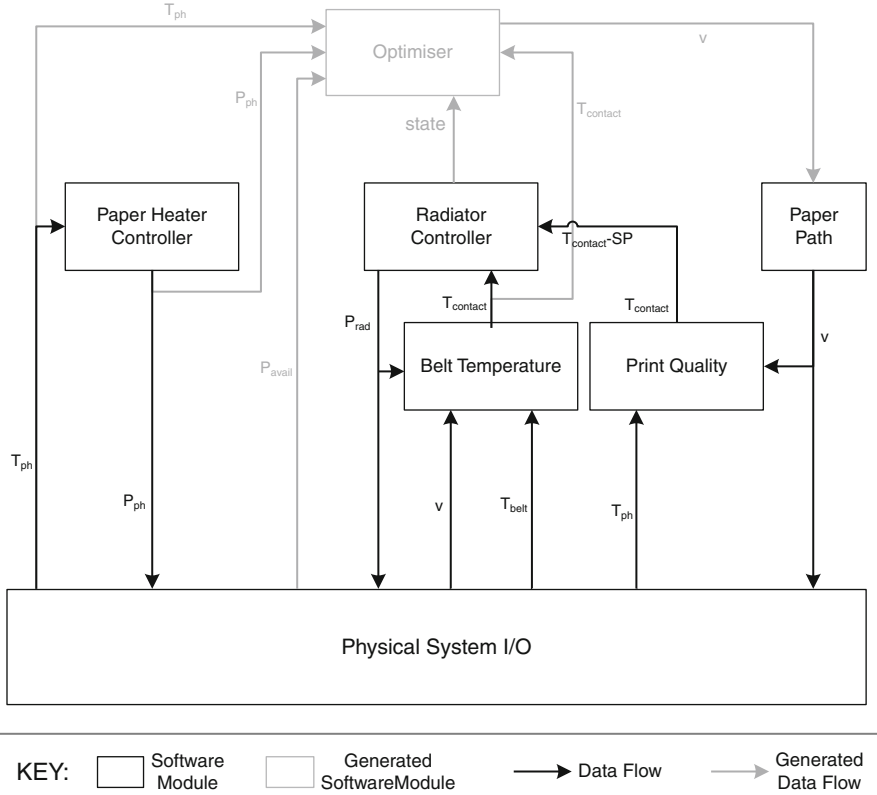


Fig. 8.15 Software structure with generated optimiser

8.4.3 Evaluation

In this section we compare the control implementation that contains multi-objective optimisation (MO2 implementation), as presented in the previous sections, with the control implementation that contains a state-of-the-practise algorithm to optimise productivity, called *Intelligent Speed*.

8.4.3.1 Intelligent Speed Algorithm

Algorithm 8.1 shows the Intelligent Speed algorithm. The Intelligent Speed algorithm works as follows. The amount of power that is not utilised (P_{margin}) is calculated. When P_{margin} is too low, the speed is decreased with 20 ppm. When P_{margin} is higher than a certain boundary (200 W), then the speed is increased with an amount that is proportional to the actual value of P_{margin} .

Algorithm 8.1: Intelligent speed algorithm

```

1  $P_{\text{total}} := P_{\text{ph}} + P_{\text{rad}}$ 
2  $P_{\text{margin}} := P_{\text{avail}} - P_{\text{total}}$ 
3 if  $P_{\text{margin}} \leq 0$  then
4   |  $v_{\text{new}} := v - 20$ 
5 end
6 else if  $P_{\text{margin}} \geq 200$  then
7   |  $v_{\text{new}} := v + 0.05 \cdot P_{\text{margin}}$ 
8 end
9 else
10  |  $v_{\text{new}} := v$ 
11 end
    // Ensure that new speed is within limits:
12  $v_{\text{new}} := \min(120, \max(60, v_{\text{new}}))$ 

```

Note that this algorithm does not optimise the power margin P_{margin} to 0 W, but maintains a certain amount of margin, which varies between 0 W and 200 W. This margin is used in real printer systems to cope with a delay in which the speed can be changed; in this experiment we assume that speed can be changed instantaneously, but in real printer systems there is a delay of a number of seconds before the speed can be adapted. To cope with sudden drops in the amount of power available during this delay period, the algorithm maintains a margin.

To make a fair comparison between the MO2 implementation and the Intelligent Speed implementation, the MO2 implementation maintains a power margin of 100 W. This is implemented by specifying the constraint $P_{\text{total}} \leq P_{\text{avail}} - 100$ on the P_{total} out-port of the Power component, as was demonstrated in Sect. 8.4.2.

8.4.3.2 Comparison

The performance of the two control software implementations regarding productivity is measured in an experimental setup. This setup uses a MATLAB/Simulink model of the Warm Process thermodynamics, to simulate the Warm Process part of the physical printer system. This Simulink [23] model has been provided by our industrial partner Océ, and is a realistic model of a printer system.

Several scenarios have been simulated. Each scenario has a fluctuating, but limited, amount of power available. Scenarios with different power fluctuation intervals (i.e. the interval after which the amount of available power changes) have been simulated. Each scenario runs for 20,000 time steps (simulated seconds).

Figure 8.16 provides the average printing speed obtained by the two implementations for each power fluctuation interval. This figure clearly shows that the MO2 implementation provides higher productivity than the Intelligent Speed algorithm.

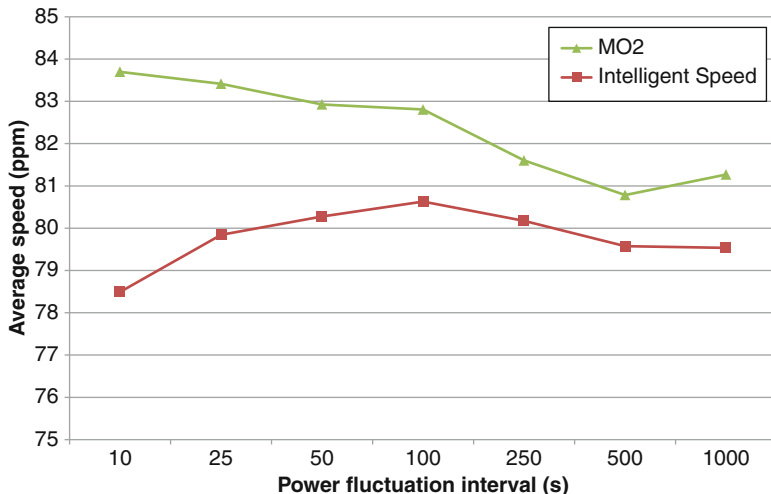


Fig. 8.16 Average printing speed

Figure 8.17 shows the average power margin obtained by the two implementations for each power fluctuation interval. As explained before, the Intelligent Speed algorithm maintains a power margin between 0 W and 200 W, while the MO2 implementation tries to maintain a power margin of 100 W.

The figure shows that the Intelligent Speed implementation generally results in an average power margin above 100 W. The MO2 implementation results in a power margin that is slightly below 100 W. Especially for larger power fluctuation intervals, the average power margin of the MO2 implementation approaches 100 W. The average power margin of the MO2 implementation is generally below 100 W, because there are situations in which there is insufficient power available to print at the lowest speed and still maintain a 100 W margin. In this case, the constraint to maintain a 100 W margin is dropped.³ These situations happen more often with higher power fluctuation intervals, as Fig. 8.17 shows.

Figure 8.18 shows the power margin of the Intelligent Speed algorithm during one simulated scenario. The figure shows that the power margin is not constant, but varies between 0 W and 200 W. This is inherent in the design of the Intelligent Speed algorithm (Algorithm 8.1 on page 274): the speed is increased when there is more than 200 W power margin and decreased when there is less than 0 W power margin. A problem with this design is that sometimes there is a low power margin

³This is a property of the chosen optimisation algorithm: When the solution space of the MOO problem is empty, this algorithm drops constraints until there is a solution. Additional configuration of this algorithm is done so that it drops the 100 W power margin constraint first. This property is not part of the MO2 method; the method leaves open what should happen when the solution space of the MOO problem is empty.

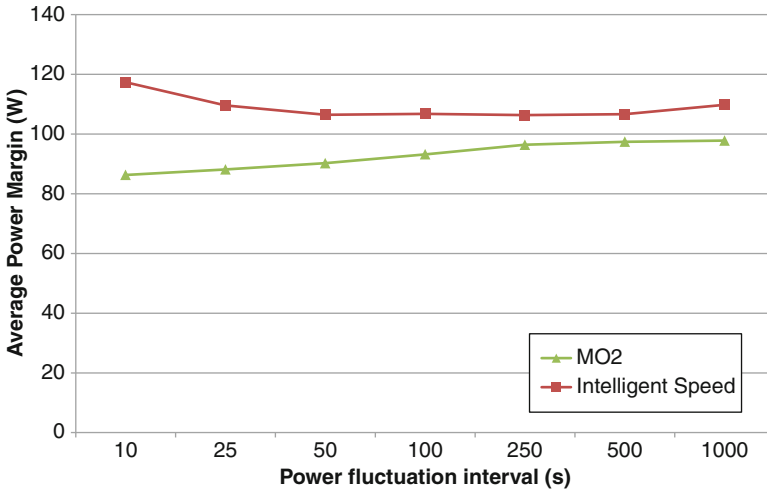


Fig. 8.17 Average power margin

available, making it harder to cope with sudden drops in the amount of power available. At other times there is a too high power margin available, which reduces productivity.

Figure 8.19 shows the power margin of the MO2 implementation for the same scenario as was used to demonstrate the power margin of the Intelligent Speed implementation in Fig. 8.18. Figure 8.19 demonstrates that the MO2 method is able to provide a precise and stable power margin of 100 W, as opposed to the Intelligent Speed algorithm which gives an unpredictable power margin between 0 W and 200 W. The short peaks and drops visible in Fig. 8.19 are caused by large changes in the amount of power available that the system cannot directly adapt to. The two longer drops in the power margin (around $0.65 \cdot 10^4$ s and $1.65 \cdot 10^4$ s) are caused by the fact that during this period there is not enough power available to print at the lowest speed and maintain a 100 W power margin. In this case, the power margin is used to continue printing at the lowest speed.

8.4.4 Discussion: Computational Performance of Multi-Objective Optimisation

Applying multi-objective optimisation algorithms results in a better performing system, but can also lead to a considerable computational performance overhead in software. This concern is particularly relevant for embedded systems, as these systems generally tend to have limited processing power available. In the end, this concern is an engineering trade-off: The engineer has to decide whether to

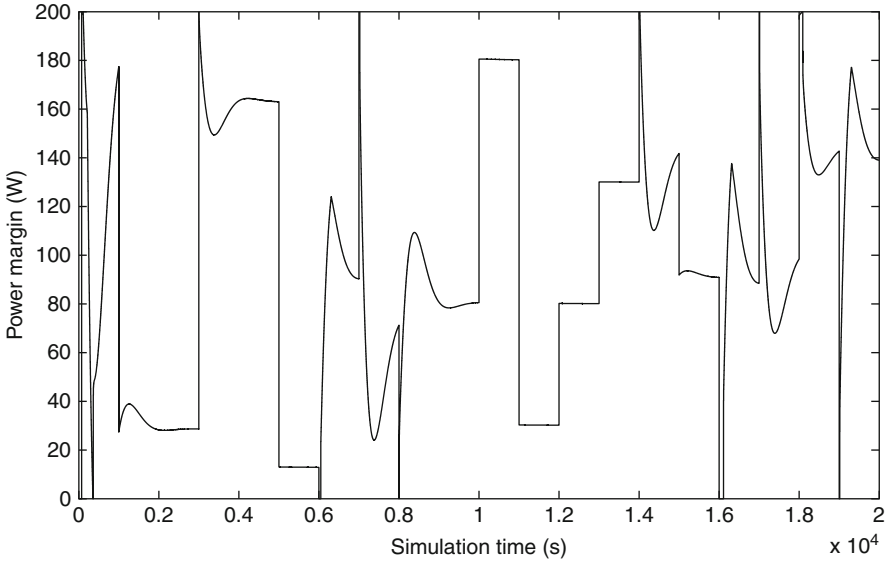


Fig. 8.18 Power margin during one scenario for the Intelligent Speed implementation

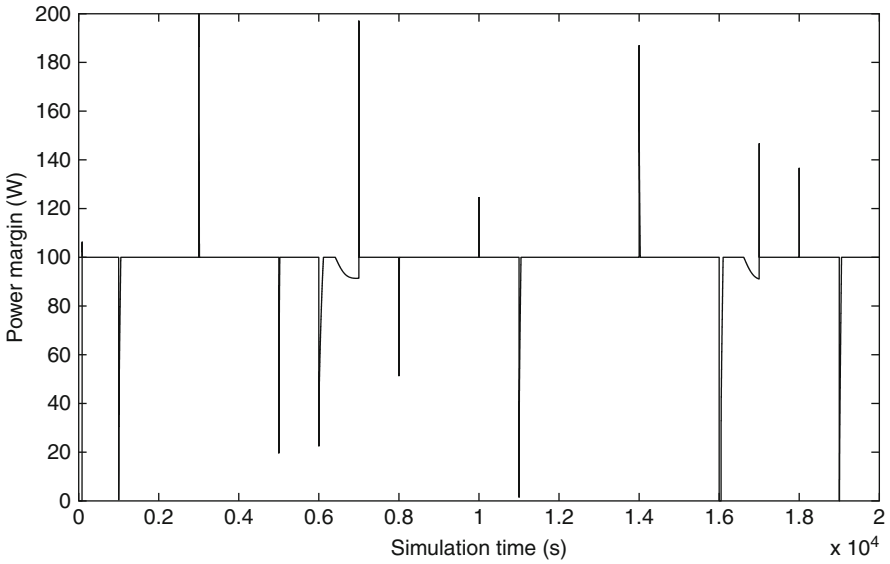


Fig. 8.19 Power margin during the same scenario for the MO2 implementation with 100 W margin

apply multi-objective optimisation algorithms for a better performing system, or not to apply them for more efficient software that can execute on limited processing hardware.

The computational complexity of solving a multi-objective optimisation problem depends on the characteristics of the problem. Certain multi-objective optimisation problems can be solved (deterministically) in polynomial time (e.g. linear programming problems [31]), while other problems are NP-hard [16]. As such, the trade-off can be influenced by careful selection of the multi-objective optimisation algorithm and adapting the multi-objective optimisation problem in such a way that it can be solved efficiently. For some systems an approximation of the optimal solution would be sufficient. In this case, using an approximation algorithm, instead of an algorithm that gives an exact result can reduce the processing power required by the optimisation software.

Many different multi-objective optimisation algorithms have been designed, both algorithms that give exact results and algorithms that give approximations. An extensive overview of multi-objective optimisation can be found in [13].

8.5 Conclusion

In this chapter we have argued that making the control component of embedded system software adaptive can lead to a competitive advantage. Without adaptive control, we typically have to statically constrain the control parameters conservatively to ensure that system constraints are always obeyed. In contrast, adapting control to the current context, e.g. environmental conditions, peer systems, or user requests, allows to reach the physical boundaries of the controlled system under specific dynamic conditions.

The research challenge here was to provide software engineers with a systematic approach to develop adaptive behaviour *without* compromising the quality of the software: While being able to manage the complexity, software must be able to adapt and evolve. To achieve this, software should be modular with limited dependencies between the modules. From a bird's eye view, our proposed approach consists of the following building blocks:

- Using domain-specific languages (DSLs) or domain-specific modelling languages (DSMLs) for expressing domain-specific behaviour efficiently. This reduces the complexity of components because developers can focus for one particular concern on *what* must be done by a component instead of *how* something is done.
- The declarative nature of DSLs and DSMLs also facilitates automatic processing of component definitions by tools; for example, such definitions can be analysed to check them for correctness, feasibility, or performance characteristics. Another example is automatic code generation; this is also suitable to establish

links between domain-specific components and the rest of the system without introducing strong dependencies between the source code of those components.

We have demonstrated our approach by means of two industrial case studies. The first example was performed in the domain of schedulers which compute an ordering in which to execute tasks to optimise the system performance according to a given objective. In a case study we have demonstrated our Scheduling Workbench for implementing schedulers in a domain-specific language separately from the system which is to be scheduled. The code generation of the workbench allowed later integration of the scheduler into a system. We have demonstrated that our approach reduces the development effort of evolving the scheduler definition and the development effort of replacing the scheduler definition completely.

In the second case study we have made the control in the Warm Process adaptive, i.e. the continuous control of a physical subsystem in a high-quality digital printer. We have presented the MO2 method and tool chain to dynamically optimise control according to multiple objectives, i.e. different quality characteristics. In the concrete example, we have applied this approach to optimise the printer system with respect to the trade-off between throughput and energy consumption. We have shown, that our approach indeed can lead to a better performing control system; in addition, our approach gives engineers the opportunity to replace (also at a later stage) the optimisation algorithm itself.

The key messages of our contribution are:

1. The software evolvability of a system is increased by separating the implementation of a (control) problem from other functionality.
2. The complexity of such an implementation can be kept low by using domain-specific (modelling) languages for expressing the (control) component.
3. The usage of DSLs and DSMLs enables static analysis as well as automatic generation of run-time support for the component and its integration with the rest of the system.
4. The integration of a component with the system can be achieved through so-called code weavers, a technology adopted from aspect-oriented programming. This technology links the execution of the component but shields the developer from having to deal with strong dependencies in the source code.
5. A methodology to get software specifications that are (relatively) cleanly modularised in the implementation, resulting in systems which are easier to understand and to maintain.

Acknowledgments This work has been carried out as part of the Octopus project with Océ-Technologies B.V. under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs, Agriculture, and Innovation under the BSIK program.

References

1. 20-sim tooling. <http://www.20sim.com>. Accessed Aug 2012
2. Banker, R.D., Datar, S.M., Kemerer, C.F., Zweig, D.: Software complexity and maintenance costs. *Commun. ACM* **36**, 81–94 (1993)
3. Broenink, J.F.: Modelling, simulation and analysis with 20-sim. *Journal A* **38**, 22–25 (1997)
4. Brucker, P.: *Scheduling Algorithms*, 3rd edn. Springer, Berlin (2001)
5. Clements, P., Bachman, F., Bass, L., Ivers, D.G.J., Little, R., Nord, R., Stafford, J.: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston (2002)
6. Collette, Y., Siarry, P.: *Multiobjective Optimization: Principles and Case Studies*. Springer, Berlin (2003)
7. Czarnecki, K.: Overview of generative software development. In: J.P. Banâtre, P. Fradet, J.L. Giavitto, O. Michel (eds.) *Unconventional Programming Paradigms*. Lecture Notes in Computer Science, vol. 3566, pp. 326–341. Springer, Heidelberg (2005)
8. Dashofy, E., Asuncion, H., Hendrickson, S., Suryanarayana, G., Georgas, J., Taylor, R.: Archstudio 4: An architecture-based meta-modelling environment. In: *Companion to the Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, pp. 67–68 (2007)
9. de Roo, A., Sözer, H., Akşit, M.: An architectural style for optimizing system qualities in adaptive embedded systems using multi-objective optimization. In: *Proceedings of the 8th Working IEEE/IFIP Conference on Software Architecture (WICSA 2009)*, Cambridge, pp. 349–352 (2009)
10. de Roo, A., Sözer, H., Akşit, M.: Runtime verification of domain-specific models of physical characteristics in control software. In: *Proceedings of the Fifth IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2011)*, Dallas, pp. 41–50 (2011)
11. de Roo, A.J.: *Managing software complexity of adaptive systems*. Ph.D. thesis, University of Twente, Enschede (2012)
12. Edgeworth, F.Y.: *Mathematical Psychics: An Essay on the Application of Mathematics to the Moral Sciences*. C. Kegan Paul, London (1881)
13. Ehr Gott, M., Gandibleux, X. (eds.): *Multiple criteria optimization: state of the art annotated bibliographic surveys*. International Series in Operations Research & Management Science, vol. 52. Kluwer Academic, Dordrecht (2002)
14. Elrad, T., Fillman, R.E., Bader, A.: Aspect-oriented programming. *Commun. ACM* **44**, 29–32 (2001)
15. Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.): *Aspect-Oriented Software Development*. Addison-Wesley, Boston (2005)
16. Glaßer, C., Reitwießner, C., Schmitz, H., Witek, M.: Approximability and hardness in multi-objective optimization. In: *Programs, Proofs, Processes*. Lecture Notes in Computer Science, vol. 6158, pp. 180–189. Springer, Heidelberg (2010)
17. Hatley, D.J., Pirbhai, I.A.: *Strategies for real-time system specification*. Dorset House, New York (1987)
18. Hatun, K., Bockisch, C., Sözer, H., Akşit, M.: A feature model and development approach for schedulers. In: *Proceedings of the 1st Workshop on Modularity in Systems Software (MISS 2011)*, Porto de Galinhas, pp. 1–5 (2011)
19. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: *Feature-oriented domain analysis (FODA) feasibility study*. Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA (1990)
20. Keeney, R.L., Raiffa, H.: *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley, New York (1976)
21. Kent, S.: Model driven engineering. In: M. Butler, L. Petre, K. Sere (eds.) *Integrated Formal Methods*. Lecture Notes in Computer Science, vol. 2335, pp. 286–298. Springer, Berlin (2002)
22. Kleijn, C.: *20-sim 4.1 Reference Manual* (2009)

23. MATLAB/Simulink (2010). <http://www.mathworks.com/products/simulink/>. Accessed May 2012
24. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. *ACM Comput. Surv.* **37**, 316–344 (2005)
25. Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2 (2007). <http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF>. Accessed Aug 2012
26. Pareto, V.: *Cours D'Économie Politique*. F. Rouge, Lausanne (1896)
27. Selic, B.: Using UML for modeling complex real-time systems. In: F. Mueller, A. Bestavros (eds.) *Languages, Compilers, and Tools for Embedded Systems*. Lecture Notes in Computer Science, vol. 1474, pp. 250–260. Springer, Berlin (1998)
28. van de Laar, P., Punter, T. (eds.): *Views on Evolvability of Embedded Systems*. Springer, Dordrecht (2011)
29. van Engelen, R., Voeten, J. (eds.): *Ideals: Evolvability of Software-Intensive High-Tech Systems*. Embedded Systems Institute, Eindhoven (2007)
30. Ward, P.T., Mellor, S.J.: *Structured development for real-time systems: Introduction & tools*. Yourdon Press, Englewood Cliffs (1985)
31. Winston, W.L.: *Operations research: applications and algorithms*, 4th edn. Thomson Brooks/Cole, Stamford (2004)