Generic components: a symbiosis of paradigms

M. Becker

System Software Research Group
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
mbecker@informatik.uni-kl.de

Abstract. Software reuse and especially the paradigm of software components are promising approaches to increase the efficiency of software development. One of the basic problems of software reuse, however, is the tradeoff between the abstraction from project-specific aspects on the one side, and the actual contribution of a reusable artifact during the realization of a new system on the other side. Stringent resource constraints further complicate the application of these approaches in domains, where efficient and therefore specialized solutions are required, e.g. in the domain of embedded systems. Generic components, which are designed and implemented to be adaptable to new application scenarios, allow to overcome these limitations, esp. if they somehow automate the essential modifications. This paper presents a concept of generic components that has been developed to facilitate the construction of highly specialized embedded operating systems. Besides the illustration of the underlying concept, the paper discusses the external representation of generic components and the internal realization of the provided variability. Futhermore it shows how generic components may be automatically instantiated by tools, and reflectes some of our experiences in constructing generic components.

1 Introduction

The highly competitive and dynamic field of software development implies the need to continuously increase the efficiency of the construction and evolution of software products. For a long time reuse of software artifacts in general and of standard components in particular have been promising approaches [McI76][Sch94] to achieve this goal through the consequent and efficient exploitation of the commonalities among produced software artifacts. However, there are inherent problems with both techniques. Firstly, software artifacts have to be made sufficiently abstract to be reusable, i.e. to be applicable in different deployment situations, while at the same time need to be sufficiently concrete to bear a reuse benefit at all. With component-based software development this gets even worse, since the components are typically intended to be reused in a black box fashion and thus often do not support their customization at compile-time in regard to individual demands. Secondly, the evolution of prefabricated software artifacts – e.g. caused by changing requirements – in most cases results in a rise of complexity that further degrades the performance and thus often prohibits

their reuse. This holds especially if tight resource constraints have to be met, e.g. in the domain of embedded systems.

To deal with the required variability, generative programming techniques, e.g. as presented in [Cza00], are sound approaches, since they also pursue reuse on the level of the production process instead solely on the product level. The variability among the artifacts thus can be more effectively managed and even be formulated in the context of the problem space. But there are problems with pure generative approaches, too. Domain-specific default assumptions have to be applied, otherwise the complexity of the problem specification and the underlying generator would be of the same magnitude as the implementation complexity of the whole problem itself. Again, the tradeoff between default solutions and the need for individual optimizations is likely to appear. As a consequence manual specializations have to be considered within the reuse activities as well, if the efficiency of the results is essential. The challenge is to combine different reuse strategies in a way that exploits the benefits of each approach. Code generation can be used to supplement the reuse of software components by automating their otherwise time-consuming and error-prone customization. Pre-planned manual optimizations within the reusable artifacts can be supported by providing adequate implementation constraints and knowledge, etc.

We have developed an approach based on the component and the generation paradigm in the domain of embedded operating systems to address the aforementioned problems. It has been coined by the stringent resource constraints, such as scarce memory and low cost CPUs, in combination with varying non-functional requirements, like robustness and timeliness. The resulting efficiency constraints – often the most crucial criterion for the success or failure of a software project within this domain – make conventional reuse approaches especially problematic in this context, since they necessitate even unique tailor-made solutions that are very hard to reuse efficiently in following projects. To address this problem, the concept of *generic components* has been developed. Generic components provide means to ease their tailoring according to individual requirements; they combine the technique of reusing pre-fabricated software artifacts with the goal of producing specialized software. Most of the variability provided by them can be automated by *generators* accomplishing even fine-grained optimizations on the level of source code, but also unique customizations are supported. Obviously, the combination of the component and the generation paradigm only becomes possible through a weakening of the conventional component notion. With generic components also grey-box reuse has to be considered.

Although this integrative concept has been developed for the domain of embedded operating systems, it should also be applicable in other domains – provided that a known and manageable set of component variants is likely to cover the majority of possible requirements. This paper presents our concept of generic components and focuses on their external representation and some general facts concerning the internal realization of the provided variability. The rather process-related questions of mapping requirements to appropriate component selections and configurations in our approach is covered by companion papers [Bau00][Bau00/2].

The remainder of this paper is structured as follows: Section 2 introduces the concept of generic components in-depth. Section 3 covers the external representation of generic components. In Section 4 several approaches to implement generic components are discussed. Finally, in Section 5, we draw some conclusions.

2 The concept of generic components

Conventional component-based software development not considering customizations at compile-time reaches its limits where tailor-made solutions are required to meet stringent nonfunctional requirements. To address this, the concept of generic components [Bau99] has been developed, an approach for delivering software solutions that – despite of relying on the reuse of components – are customized to the settings of a specific application domain. Generic components combine the techniques of reusing pre-fabricated software artifacts with the goal of producing specialized software.

A *generic component* can be defined as a software module allowing to choose its properties to a certain degree without necessarily having to write or change code manually. The choice of properties may concern both functional and nonfunctional aspects, e.g. the choice of algorithms or data structures and the adaptation of the component's interface. To support this kind of customization, so-called *generic parameters* together with their respective range of supported values are identified at the component's *customization interface*. These are the externally visible setscrews to control the instantiation of the generic component into a conventional one (cf. Fig. 1). Within the generic components so-called *variation points* – placeholders for the variable parts of code – can be identified that are replaced with concrete code fragments produced by *generators* in accordance to the actual values chosen for the generic parameters during the instantiation. Typically only few parameters have an impact on a variation point; they form the input to the specific generator that is associated with this variation point. To subsume, customizing a generic component hence means to choose actual values for the supplied generic parameters, and let appropriate generators accomplish the necessary modifications. Besides a pure reduction of customization effort, customizations on a higher level of abstraction can be supported through the application of generative techniques, e.g. template meta programming, or sophisticated code compilers. The application developer, for instance, applying generic components to construct embedded control systems does not need to know in detail what changes within the components are required to provide thread preemption or a `delay()` functionality. He only has to decide if the functionality is needed and has to be informed about the additional resource requirements.

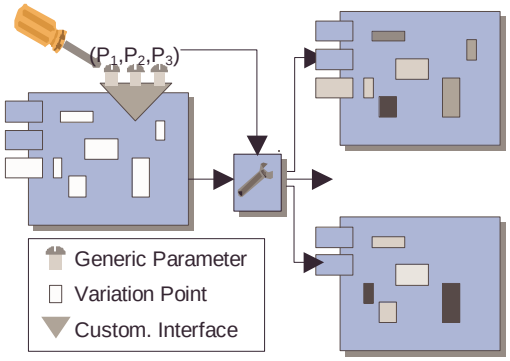


Fig. 1. Conceptual view on generic components.

From the conceptual point of view a generic component is a kind of abstract representative of a set of closely related software components with slightly different properties. Since it is only sensible to integrate such components in a generic component that share a considerable amount of common code, generic components – according to [Par76] – can be considered as a component family. The different family members can be distinguished by the tuple of actual values, chosen from the subset of the Cartesian Product over the generic parameters that contains the consistent parameter

selections. As with program families, a thorough analysis of the presumably needed properties, and the resulting commonalities and variabilities should take place before generic components are implemented. Domain Analysis methods [Cza00][Sch94] or the Multi-Paradigm Design [Cop98] can be used to this end.

2.1 Generic parameters

Generic parameters – the pendants of run-time configuration parameters at compile-time – allow to tailor the component's properties by simply choosing one of the supported parameter values. Trivial properties like the value of a constant can be instantiated as well as code fragments realizing special data structures or algorithms. Specific adaptations concerning the runtime environment of the component, e.g. the type of used hardware, can thus be handled within the component, instead of providing external wrappers causing additional costs. Especially in the case of directly interacting components, using function calls to communicate with each other, adaptations of the calling interface are very useful. Furthermore, the interfaces of generic components can be the target of the customization. Superfluous complexity [Big94] can be reduced in this way by providing only the needed functionality. Finally, a tailoring of nonfunctional properties of the component can be supported, e.g. an optimization concerning the processing time or memory consumption at run-time. This tailoring is usually driven by the wish to construct software components that meet the essential given requirements and moreover support as many as possible of the desired properties, e.g. little resource demands, reduction of the interface complexity, etc.
Three major approaches to tailor code that is used to replace a variation point can be identified (cf. Fig. 2). Variation points are filled by *selection* if they are simply replaced with discrete entities that are selected from a finite set of alternatives. Entities in this sense are pre-built parts of a component implementation, e.g. data structures, algorithms, or other code fragments. In the case of *generation*, the variation points are placeholders for code fragments produced by associated generators depending on the provided actual values of the generic parameters. The range of generator functionality can reach from the simple insertion of values for constants over the expansion of small macros to sophisticated code generation, where abstract descriptions in form of the actual values are translated into concrete artifacts. With this type any generative technique can be utilized in generic components. Where the aforementioned tailoring types do not provide sufficient flexibility, variation points can be replaced by parameter *substitution*, i.e. a variation point is substituted with the actual value of the respective parameters. In this way, user-defined code can be inserted into the component. This type of tailoring offers the highest degree of flexibility to the user and is typically used to handle code

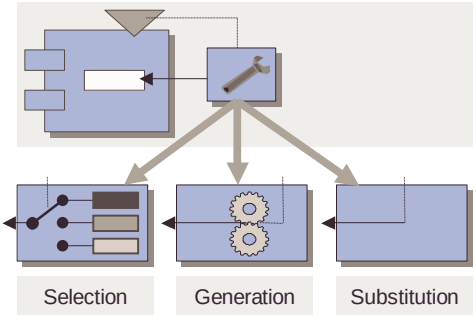


Fig. 2. Different types to tailor code that replaces a variation point.

fragments that are unique in each component instantiation, e.g. individual optimizations. Besides the pure replacement of the parameter an important and nontrivial task during the instantiation of a generic component can be the monitoring of the insertion. In this case, the generator checks the provided code against some constraints or predefined limits formulated by the implementer of the component. If a generic parameter only controls customizations of the same type, esp. in the case of substitution, it is sensible to speak of a *selection*, *generation* or *substitution parameter*. The mentioned tailoring types represent different kinds of reuse. While a selection reuses concrete artifacts, generation reuses the production methods, and substitution supports the reuse of knowledge and constraints about prospective solutions.

In principle, it is reasonable to perform the customization hierarchically, i.e., new generic parameters can be introduced with the code fragments that replace variation points. However, a recursive introduction of new parameters while choosing parameter settings adds a considerable amount of complexity to the configuration process. Therefore, it is sensible to restrict the introduction of new generic parameters to the tailoring by selection as illustrated in Fig. 3. A typical scenario where hierarchical parameter introductions are used is the following: a fixed number of implementations exist for a variation point, e.g. different hardware drivers , but the opportunity to add a new implementation has to be given to the user, too. This can be realized with a selection between the existing code fragments and a frame that provides a substitution parameter. If the user decides to provide new code, he selects the frame and afterwards supplies the new code by substituting the new parameter.

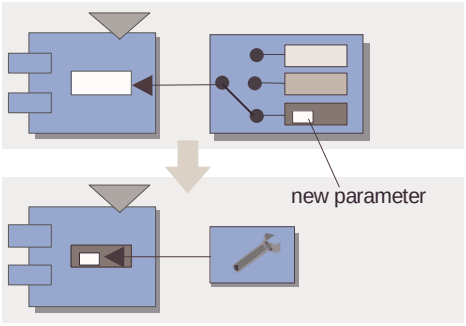


Fig. 3. Hierarchical tailoring: introduction of a new parameter though selection.

2.2 Objectives of the approach

The concept of generic components has been developed to make the component paradigm applicable even in domains where the black-box reuse of components typically will not work, i.e. that tailor-made solutions are required for some reason. Through the identification of variation points and of generic parameters the customization at compile time is supported allowing the tailoring of components in respect to stringent requirements. Doing so, the generic components help to overcome the inherent tension between reusability and usefulness of conventional components. If feasible, the tailoring complexity can be captured in pre-build fragments or sophisticated generators providing implementations for the variation points' instantiation. Otherwise substitution parameters can be used to support the creation of new specializations through the clear identification of the relevant variation points and the specification of implementation constraints and knowledge. Some kind of abstract components can be realized in this way that are very useful in the case of highly individual and therefore not

reusable aspects, or yet unknown requirements and solutions. The clear identification and the factorization of the common and variable parts is advantageous for the following reasons: Firstly, the evolution of the generic components gets easier, e.g. as common code is not replicated in several component variants or as new variants created by the user can be easily integrated into the respective generators. Secondly, the number of related components in reuse pools and therefore the complexity of their retrieval can be reduced in case of orthogonal variabilities. If a thread manager component, for instace, offers generic parameters for the number of supported threads and the kind of the used stacks, then a lot of different thread manager variants would have to be in the reuse pool in order to meet stringent requirements. In the case of a generic component only one component with corresponding generators is needed. With the ability to flexibly adapt the implementation of the reused components, conventional approaches to improve component reusability, which typically interfere with the overall system performance, can be avoided, e.g. the strict uncoupling of components, and large inheritance hierarchies. To subsume, the concept of generic components approximates the advantages of manual customization and optimization, while on the other hand sidestepping the obstacles of time-consuming and error-prone coding by reusing pre-fabricated and tested implementations.

3 The user's perspective on generic components

The reusability of software components strongly depends on their external representation. Comprehensive descriptions of the provided functionality, the interfaces, the dependencies on other artifacts and their architectural embedding, etc are essential, especially as the components are typically reused in a black-box fashion. This also holds for generic components. Due to the variability provided by the generic components, however, further information to represent the generic component in a comprehensive manner is required. From an external point of view, generic components mainly differ from conventional ones in the provision of an additional customization interface and in the fact, that some of the components' properties are kept variable. Furthermore, some mechanism is supplied that supports the automatic tailoring of the component according to the actual values chosen for the generic parameters. In the following, however, we will concentrate on the customization interface and elaborate the required information that has to be presented to the user of the generic component to support its consistent configuration and application.

3.1 Customization Interface

A generic component in our sense is a kind of abstract representative of a family of software components. It can be put into concrete shapes with the assignment of actual values to the provided generic parameters. Therefore, at least the generic parameters must be clearly identified to give the user an idea what can be tailored at all – but that will not suffice. To scope the variability of the parameters, the range of supported values for each of them has to be specified as well as the interdependencies among them, typically restricting the number of consistent parameter selections. As the actual

values of substitution parameters are directly inserted into the generic component, it is impossible to enumerate all supported values for this kind of parameter – otherwise the substitution parameter should be replaced by a selection parameter. However, at least the language and an abstract description of the expected code fragment can be stated to facilitate the consistent parameter assignment in that case.

Since the users of generic components are naturally more interested in the results of their tailoring than the tailoring itself, further information regarding the variable properties should be given too. In our approach, this information is structured in so-called *feature types* and *features*. A feature type comprises clearly distinguishable features and represents a variable property of the component, e.g. scheduling strategy, memory management strategy, or synchronization mechanism. Feature types thus span the space of components that can be instantiated from a generic component. This information, together with the description of the common features, is typically used to determine in a very early stage, whether a generic component is applicable for the specific problem or not.

Obviously the feature types of a generic component are somehow correlated with its generic parameters. If this correlation is explicitly represented in the customization interface, it can be used to determine the concrete features of the tailored component before the final instantiation step. This feedback can be used for composability tests and for further adjustment of the component, and therefore facilitates the selection of an appropriate actual parameter set. For each feature type, a functional dependency on a subset of the generic parameters can be identified. In our approach, this set of feature-dependency functions is used to represent the correlation between a feature type and its parameters. One special case is worth to be mentioned: feature-dependency functions that are invertible can be used favorably to determine parameters based upon the desired features. This significantly facilitates the customization of a component. Fig. 4 subsumes the aforementioned points into a formal representation of the customization interface. As the comprehensible description of the listed aspects is non-trivial and expensive, it may be reasonable to provide less information at the cost of user support, e.g. the feature-dependency functions could be omitted.

Representation of the customization interface CI:

$$CI =_{df} (P, (P_i), VS, F, (F_i), (FD_i))$$

With:

P	$=_{df}$	$\{P_1, \ldots, P_n\}$, the set of generic parameters
P_i	$=_{df}$	$\{p_1, \ldots, p_{m(i)}\}$, supported values for each parameter
PP	$=_{df}$	$P_1 \times \ldots \times P_n$, the space of parameter selections
VS	$=_{df}$	$\{pp \in PP \mid pp \text{ is a valid selection}\}$, the set of valid selections
F	$=_{df}$	$\{F_1, \ldots, F_k\}$, set of feature types
F_i	$=_{df}$	$\{f_1, \ldots, f_{m(i)}\}$, variety of features for the feature type F_i
PF	$=_{df}$	$F_1 \times \ldots \times F_k$, the space of varying features
FD_i	$=_{df}$	$VS \rightarrow F_i$, feature-dependency function

Fig. 4. Representation of the customization interface.

3.2 Parameter dependencies

After the discussion of the information that has to be presented at the configuration interface, we take a closer look at the configuration step itself. In practice, the generic

parameters often are not independent of each other. Typically, it is possible to identify hierarchical dependencies between them; i.e. the presence of some generic parameter depends on the choice of actual values for others. For instance, a generic parameter that is only supplied in the context of a distinct data structure does not have to be considered if the data structure is not used in an actual configuration. Although this information is already represented by the set of valid selections, the customization complexity for the user can be further reduced if these parameter dependencies are explicitly considered during the configuration step. Instead of a *flat configuration*, where all parameters are visible throughout the customization, although a part of them is not really needed, a *hierarchical configuration* should be preferred. Here only those parameters that bear a meaning for the current level of choices are presented. This approach typically results in less parameters and a more intuitive way of configuration, but requires more support in the customization interface and the instantiation of the component. Eventually some backtracking has to be done in the case that a lower level of configuration reveals the insufficient selection of a parameter on a higher configuration level. In Fig. 5 an example to illustrate the different configuration approaches is given. The arrow between A and B denotes that B only has to be considered in the case A is set to 2. Obviously, the hierarchical approach facilitates the configuration, e.g. no values have to be selected for parameters B and E unless the value 2 is chosen for A. The example additionally points out the two major reasons for hierarchical dependencies. They are either caused by an implicit requires-dependency as shown between A and B, or hierarchical tailoring (cf. Section 2) as with A and C.

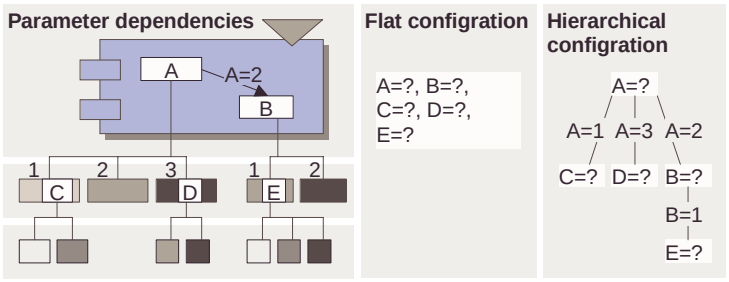


Fig. 5. Parameter dependencies and the different configuration approaches.

3.3 Tool support

Several approaches to represent the customization interface and to support the selection of actual values in a user-friendly way are reasonable. However, it becomes clear that tool support, esp. for the feature-dependency functions and hierarchical configuration, is indispensable. Therefore, we apply tools [Bau00] based on the technique of extended design-spaces, a multi-dimensional classification scheme that allows to express such correlations between features and parameters as well as hierarchies of parameters in a very natural and concise way [Bau98][Lan90][Lan96]. The required information is kept in so-called component spaces that are used by our tools to control the customization process. Besides the customization of generic components, a mechanism has to be supplied that supports the automatic tailoring of the component in correspondence with the chosen configuration. These two mechanisms can favora-

bly be integrated into a single tool, thus automating the customization and the instantiation of generic components. Our *D-Space-1* tool [Bau00], for instance, facilitates the application of the customization interface of our generic components through a graphical interface and controls the tailoring of the generic components by calling the appropriate generators, which accomplish the specified modifications.

4 Internals of generic components

From the implementers' point of view, the descriptive elements of generic components – especially their code – significantly differ from those of conventional ones. Generic components comprise a set of variable implementations and generic parameters to determine them. In addition to the conventional description elements, such as code, header files, and resources, the required mechanisms to automate the tailoring must be supplied in some way by the implementer of the generic component. A wealth of well-known possibilities addressing different facets like code organization and modification, customization of structure and behavior, extensibility of existing implementations, or code generation are available for this kind of meta programming, e.g. appropriate language constructs, pre-processors, external generators. All of these viable implementation approaches have to somehow integrate the common and the variable description parts that form a generic component. Before discussing the different tailoring techniques themselves, we want to look at the preconditions needed to implement the tailoring facilities in the generic components.

4.1 Integration of commonalities and variabilities

Different techniques can be applied to tailor the description fragments that replace the variation points in the concrete component implementation. All of these approaches have at least two concepts in common. Firstly, the variable description parts have to be associated with the common ones in some way. Secondly, links between the internally used generic parameters and their externally visible pendants at the customization interface have to be established to propagate the external settings. Besides the mere description of the variability, the following aspects thus have to be addressed in some manner: the identification of the variation points, the association of tailoring transformations, and the representation of the generic parameters. Although the clear identification of the variation points might seem to be dispensable in some cases, esp. if the same language is used to describe the common and variable parts of a generic component, it is extremely helpful if the variability of the generic component has to be revised or new variants have to be added. With this little overhead, tool support for the localization of variable implementations throughout the generic component can easily be provided. The variation points can be identified by a special markup that obviously must not be used within the description of the common parts. For this reason, the number of different markups should be kept as small as possible. The coherent use of unique markups further eases the construction of tools for the identification of the variation points within the generic component in that they do not have to be aware of the actual techniques applied to describe the variability.

Concerning the association of the tailoring transformations with the variation points, two major methods are conceivable: *inline* and *link*. The tailoring is specified inline, if it is described directly in the variation point. Since this results in a mixture of different kind of description types in most the cases, doing so is only advisable if the transformation is not too complex and cannot be applied elsewhere. More flexibility is provided through an external tailoring description or a stand-alone tool that is linked with the variation points. This link can be realized with an entry in the variation point that is used to call the external generator, or an external identification of the variation point that can be handled with a unique variation point identifier. In the latter case, several variation points can be substituted with one transformation. As generic parameters typically affect multiple variation points, the required elementary transformations sometimes can be combined into one comprehensive transformation.
As stated above, the generic parameters that control the instantiation of a variation point have to be represented in a way that allows the propagation of the respective actual values provided at the customization interface. If a transformation is specified inline no further representation of the generic parameters is required, since they can be identified and replaced within the transformation descriptions if a uniform naming scheme is used. If a transformation is associated with a variation point via a link, the required parameters have to be specified along with the link somehow, e.g. as call parameters of the external tool.

4.2 Tailoring techniques

After the discussion of methods to integrate the common and variant parts of generic components, we will now focus on the tailoring techniques applicable to realize the variability. As the concept of generic components is primarily aimed at compile-time tailoring, almost any text transforming or text generating technique can be deployed. Based on our experience, the necessary tailoring can be favorably realized through a combination of the following techniques:
Separate files – separately implemented variant parts of components are selected by choosing the corresponding source file for compilation and linking. This is the most simple approach; neither modifications of the descriptions nor special actions for instantiation are necessary. The selection of the files can be accomplished through external file management tools like RCS or CVS.
Pre-processors – scan the descriptions for particular tokens – e.g. programming language constructs or specific pre-processor statements –, parse their semantic context, and perform associated pre-defined actions. In their full extent, pre-processors are problem-specific compilers similar to those for simple programming languages and can be built using scanner and parser generators like lex/yacc. With special pre-precessors other programming paradigms such as Aspect-Oriented Programming [Kic97] can be applied within generic components, e.g. to realize efficient implementations of variable nonfunctional aspects. However, a great deal of variability can be realized with the pre-processor functionality provided by the programming languages or scripting tools like sed or perl in a very straightforward way.

Generic mechanisms of programming languages – most programming languages and the corresponding compilers somehow support generic programming, e.g. through object-orientation, templates, or compile time parameters and optimizations. Wherever it does not contradict efficiency and comprehensibility, the deployment of such mechanisms is a natural and well-known way to realize the required variability. Examples are the approaches of template meta programming [Vel98][Vel95] and generative programming [Cza99] that both extensively utilize the template mechanism offered by C++ to build variable implementations. One drawback of these mechanisms is that the implementation of the common and the variable parts are mixed up, thus complicating the location of the variable implementations in most cases.

Code generators – the most powerful, but also most complex way to perform the tailoring. Higher level descriptions of the desired functionality are passed through the generic parameters to external generators producing tailored description parts, which afterwards replace the correspondig variation points. The use of generators is especially tempting wherever the modification of pre-built components does not appear to be sufficient to cope with heavily varying and unforeseen requirements. One problem with generators is that they must apply domain-specific defaults for those aspects that are not explicitly specified by their input, mostly resulting in code that in less efficient as manually optimized code.

4.3 A glance at implementations

In the following, two approaches to implement the tailoring of a generic component are exemplarily presented. In the first example given in Fig. 6, the UNIX stream editor sed [Dou97] is employed to tailor the `get` function of a generic dispatcher that supports multi-processor environments. To allow for optimizing the source code for single-processor environments, references to the parameter `n` have been marked by comments of the form `GPx_My`. An appropriate sed script can search for the respective lines of source code and apply whatever

```
TCBptr Dispatcher::get(size t n){// GP2 M10 TP:CHANGE
   TCBptr old;
   old = running[n];    // GP2 M11 TP:CHANGE
   old->setLastCPU(n);  // GP2 M13 TP:REMOVE
   // GP5 M7 TP:INSERT
   ProcContext.All.Save(old->TCB.Register);
   return old;
}
```

source code

```
s/\(Dispatcher\.[hc][hc] -\)/\1 Single Processor -/
...
/GP2 M10/ s/\(size t n\)//
/GP2 M11/ s/\[n\]//
s/\/\/GP2 M13//
...
```

sed script

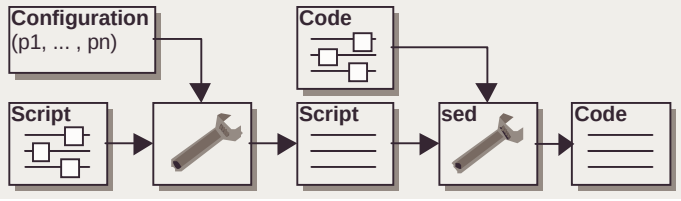


Fig. 6. A small excerpt from a generic dispatcher component together with a sed script allowing to tailor the source code.

changes are necessary. According to the sed command syntax, line 3 for instance searches for lines with the mark `GP2_M10`, then takes the first occurrence of the string "`size_t n`", and replaces it by an empty string. Variation points and the corresponding generic parameters are identified with the `GPX_MXX` markup. The tailoring description is specified externally in the sed script and is therefore associated via links – the markup in this case – with the variation points. It may be questioned, how the actual values of the generic parameters are considered in this approach. Apparently, these values must be propagated into the script. For this, a generic sed script is provided along with the generic component (cf. figure 6). In this script, placeholders for the generic parameters are identified. After the parameter assignment the generic parameters in the generic sed script are replaced by the actual values. The result of this replacement is a sed script that is finally used to tailor the component.

In the next example, a quite different approach is presented. Figure 7 shows an excerpt of a generic scheduler component. Java is used as a meta-programming language to implement the customization. The tailoring is described inline in the source code. All lines that start with a `//.` mark contain instructions of the Java meta-program. In lines 1 – 2, for instance, the generic parameters are declared, and lines 5-7 show an if-block. The meta-programming results in a source code description denoted as pseudo code that can be transformed through a simple generator into an executable Java class. Advantageous in this approach is that, although a kind of pre-processing is used to implement the tailoring, the complete expressiveness of a full-fledged programming language is available, and that finally an executable class is generated, which encompasses the generic component and automates the required tailoring. Any external generator can be called from within the pseudo code, and interactive support for substitution parameters becomes feasible. Conceptually, this approach is closely related with the work presented in [Kam99] in the way that a general purpose programming language is used to automate variability within components. The both approaches differ in the scope of the meta programming, since we use this technique only to implement the generic components not to compose them. Our composition approach can be found in [Bau00].

pseudo code

```
//. generic parameters:
//.            String schedStrategy;
[...]
SCHEDULER C::~SCHEDULER C() {
//.  if ( schedStrategy.equals("Prio") ) {
       delete [] TABLE;
//.  }
}
[...]
void SCHEDULER C::ReadyToRun(THREAD P pclThread){
  int  iPriority = pclThread->GetPriority();
//. if ( schedStrategy.equals("Prio") ) {
//.   includeFile("ReadyToRun prio.cpp");
//. } else if ( schedStrategy.equals("EDFPrio")){
//.   iEntries++; }
```

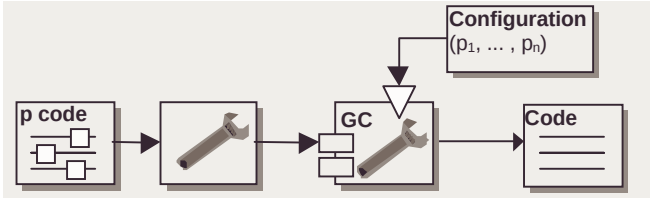


Fig. 7. Generic scheduler component that is transformed into an executable generic component automating its own configuration.

5 Conclusions

The presented concept of generic components aims at improving the reusability of pre-fabricated software components by explicitly planning for specific types of foreseeable adaptations, and by providing mechanisms and tools to perform such adaptations automatically. Generic components are implemented to be easily modified within prescribed bounds and to expose generic parameters, which provide the external handles for choosing and fine-tuning component properties. Tools like configuration management systems, pre-processors, or full-fledged code generators then allow to flexibly customize components for specific application environments with minimized manual intervention. Although the individual techniques used to implement and customize generic components are all well known, their explicit combination within the concept of generic components presents a new and promising way to pursue component reuse, even in the domain of embedded systems. With the ability to flexibly adapt the implementation of reused components, conventional approaches to improve component reusability, which typically interfere with the overall system performance, can be avoided. The concept of generic components approximates the advantages of manual customization and optimization, while on the other hand sidestepping the obstacles of time-consuming and error-prone coding by reusing prefabricated and tested implementations.

We have applied the presented techniques and the notion of generic components to build a number of reusable building blocks for embedded operating systems. As software reuse is especially beneficial – not to say: only works – within one type of architecture [Gar95], these building blocks are designed for specific reference architectures, respectively. For a small operating system kernel for embedded systems, for instance, a total of 12 generic components provide the flexibility to generate more than a hundred different kernel variants covering a wide spectrum of possible requirements. Thanks to the fine-grained modifications that are applied to the generic components during instantiation, the resulting system properties – especially concerning critical nonfunctional aspects such as the timing behavior and the memory consumption – are comparable to manually tailored implementations. On the downside, however, the implementation effort for generic components is significantly higher than for conventional components. According to our experience, making component implementations generic approximately accounts for additional 50 100% of the development effort as compared to conventional implementations. And for most cases, this is just half the way: in order to further support the selection and configuration of components by tools as presented in chapter 3 or in [Bau00], comprehensive component descriptions are required. Such descriptions roughly make up another 100% of the initial effort for conventional implementations, ultimately summing up to tripled development costs for generic components – an investment, after all, that is typically compensated by the increased probability of reuse. To further reduce this overhead, we are currently extending our research to the questions of how to specifically support the development of generic components. The possibilities range from ergonomic editors smartly handling the variant parts of the source code, to automated detection of component interdependencies and automatic deduction of properties of composed artifacts.

6 References

[Bau98] Baum, L.; Geyer, L.; Molter, G.; Rothkugel, S.; Sturm, P.: *Architecture-centric Software Development Based on Extended Design Spaces*, Proc. of the 2nd ARES Workshop (Esprit 20477), Las Palmas de Gran Canaria, February 1998

[Bau99] Baum, L.: *Towards Generating Customized Run-time Platforms from Generic Components*, Proc. of the 11th Conference on Advanced Information Systems Engineering (CAISE'99), 6th DC, Heidelberg, Germany, June 1999

[Bau00] Baum, L.; Becker, M.; Geyer, L.; Molter, G.: *Mapping Requirements to Reusable Components using Design Spaces*, Proc. of the IEEE Int'l Conference on Requirements Engineering (ICRE-2000), June 19-23, Schaumburg/Chicago, USA, 2000

[Bau00/2] Baum, L.; Becker, M.; Geyer, L.; Gilbert, A.; Molter, G.; Tamara, V.: *Supporting Component-Based Software Development Using Domain Knowledge*, 4th World Multiconference on Systemic, Cybernetics and Informatics (SCI2000), 2000

[Big94] Biggerstaff, T.: *The Library Scaling Problem and the Limits of Concrete Component Reuse*, Proc. of IEEE Int'l Conference on Software Reuse, November 1994

[Cop98] Coplien, J.O.: *Multi-Paradigm Design for C++*, Addison Wesley Publishing Company, 1998

[Cza99] Czarnecki, K.; Eisenecker, U.: *Components and Generative Programming*, Software Engineering Notes, vol. 24, no. 6, 1999

[Czar00] Czarnecki, K.; Eisenecker, U.: *Generative Programming*, Addison-Wesley, 2000

[Dou97] Dougherty, D.; Robbins, A.: *sed & awk*, 2nd Edition, O' Reilly, 1997

[Gar95] Garlan, D.; Allen, R.; Ockerbloom, J.: *Architectural Mismatch: Why Reuse is So Hard*, IEEE Software, 12(6), 1995

[Jac97] Jacobson, I.; Griss, M.; Jonsson P.: *Software Reuse - Architecture, Process and Organization for Business Success*, ACM Press/Addison-Wesley, 1997

[Kam99] Kamin, S.; Callahan, M.; Clausen, L.: *Lightweight and Generative Components I: Source-level Components*, 1st International Symposium on Generative and Component-Based Software Engineering (GCSE'99), 1999

[Kic97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopez, C.; Loingtier, J. M.; Irwin, J.: *Aspect-Oriented Programming*, European Conference on Object-Oriented Programming (ECOOP), 1997

[Lan90] Lane, T. G.: *Studying Software Architecture Through Design Spaces and Rules*, Technical Report CMU/SEI-90-TR-18, Carnegie Mellon Univ., 1990

[Lan96] Lane, T. G.: *Guidance for User-Interface Architectures*, in: Garlan, D.; Shaw, M.: Software Architecture – Perspectives on an Emerging Discipline, Prentice Hall, 1996

[McI76] McIlroy, M.D.: *Mass-produced software components*, Software Engineering: Report on a Conference by the NATO Science Committee, (Naur, P.; Randell, B., eds.). NATO Scientific Affairs Division, Brussels, 1976

[Par76] Parnas, D.L.: *On the Design and Development of Program Families*, IEEE Transactions on Software Engineering, SE-2:1-9, March 1976

[Sch94] Schäfer, W.; Prieto-Diaz, R.; Matsumoto, M (eds.): *Software Reusability*, Ellis Horwood, New York, 1994

[Vel95] Veldhuizen, T.: *Using C++ template metaprograms*, C++ Report, vol. 7, no. 4, May 1995

[Vel98] Veldhuizen, T.: *Template Metaprograms*, http://www.cs.rpi.edu/~musser/ap/blitz/meta-art.html, 1998