The Use of Parameterised Contracts for Architecting Systems with Software Components

Ralf H. Reussner

Universität Karlsruhe (TH), Am Fasanengarten 5, D-76128 Karlsruhe, Germany
reussner@ira.uka.de

Abstract. We discuss relations between component design and software architecture and how CASE tools and generators can support software architectural design with components. More specific, we present the concept of parameterised contracts and show how they influence the granularity of software components. Parameterised contracts link the providers- and the requires-interface of a component. This allows to perform automatically a certain class of component adaptations without changing the code. These adaptations also do not have to be foreseen and programmed by the component developer in advance. We show how parameterised contracts can be used to enhance the reusability of software components in software architectures.

1 Introduction

In component oriented programming (COP) software components are regarded as units of late composition [14]. This view is justified by the anticipated benefits of reusing software components (considering time-to-market, costs, and quality of software). Due to that view, a lot of technical-oriented research in component-oriented programming was concerned with composition-time interoperability checks (e.g., [3, 5, 15]). Several new interface models have been proposed which allow to detect more interoperability errors than the currently used signature-list-based interfaces do.

Most work on software architectures (e.g., [1, 6, 13]) looks at structural interactions (i.e., connectors) between components and abstracts away from concrete interface models. More recent work in software architecture (like [4]), also consider dynamic reconfigurations of composed systems, or take a more process-oriented view to apply software architectures to create product families [2]. Most architectural description languages (ADLs) model for components the explicit separation of the provides-interface (i.e., the services offered by a component) and the requires-interface (i.e., the external services required by a component) and intentionally do not restrict themselves to a certain interface model.

Bringing together software architecture research and component oriented programming can have several benefits:

- a higher degree of reuse of existing components in new software architectures
- better support of dynamically reconfigurable software architectures
-- (in the long term) a better engineering foundation of architecting software systems, if we understand how local component properties interact with global system properties.

To yield these benefits, we have to ease the application of components not only as a unit of late composition, but also as a unit of architectural composition. This support of component integration relates to a hierarchy of research problems.

1. When to accept / reject a component for integration into a new architecture. This question relates to well-known problems of interoperability research: (1) How to ensure the correct usage of a component by the rest of the system (i.e., other components, a framework, etc.), and (2) how to ensure that all resources required by a component are available?

2. How can we adapt a component in case it does not fit exactly into the environment? On the one hand, practical experience shows that existing components usually cannot be reused in a new context as they are. On the other hand, users of components cannot or are not willing to change components. So what we need is a adaptation support by the infrastructure.

3. How to conclude from local component properties to global system properties. (Recently one concentrates on non-functional properties or quality attributes.)

This paper concentrates on question two and tackles question three. The contribution of this paper lies in the formulation of parameterised contracts and in demonstrating their relevance for software architectural problems. Parameterised contracts [9] are a generalisation of classical contracts, as defined by Meyer [7]. We use parameterised contracts to restrict a component’s provides- and requires-interface, hence adapting the component to specific reuse contexts. Not touching the code allows us to apply this concept to black-box-components. Implemented in the infrastructure of a component system, neither the component developer nor the system developer has to foresee and manually program these kind of adaptations. This reduces component and system development costs.

In section 2 we further discuss the links between COP and software architecture, motivate the importance of parameterised contracts and show their relevance for component design and system architecture. In section 3 we present parameterised contracts as an approach for component adaptation. In section 4 we show the application of parameterised contracts with examples of two different interface models. Section 5 concludes and sketches the application of parameterised contracts as a more general concept to conclude from local component properties to global system properties (question three).

2 Linkage between Component Design and System Architecture

2.1 Use of Components and Generators in Software Architecture

Interface information of a component can be used to decide whether to reuse a component or not. Hence, we can utilise interface information for the important
issue of software architecture evaluation in early stages of development. When
to decide between two alternative software architectures, we can choose the ar-
chitecture, which achieves a higher degree of reuse. This degree of reuse strongly
depends on the effort one has to spend to adapt existing components into the
new architecture's context.

Without concrete interface information of a component, we usually cannot
evaluate the effort of the component's adaptation. Generally, we see two ways
how CASE tools can use interface information of software components to ease
the adaptation and to estimate the effort of adaptation: (a) Semi-automatic gen-
eration of adaptors out of interface information (e.g., [12, 16]). The more complex
the adaptor is, the higher the effort of integration (and the likelihood of pro-
gramming errors). (b) The automatic adaptation of a component by a generator. In
the case the result of that adaptation satisfies the requirements, the component is
integrateable. In case of not, we can try to generate adaptors. Hence, the automatic
adaptation tries to combine a component oriented approach with an generator-
oriented approach to achieve a higher degree of reuse of components (generation
of adapted components and generation of adaptors). In our context the deployment
of generators during system design is sufficient, but also the deployment of generators for composition-time adaptation has its applications.

The class of adaptations which can be performed with parameterised con-
tracts is the class, where components restrict their requires-interface (in case
the component itself has not to offer all functionality), and vice versa, where components restricts its provides-interface (in case the environment offers not
all requested functionality).

In the following we motivate why this class of adaptation is practically rele-
vant and relates to the design of components and system architectures.

### 2.2 Component Design and Reuse in Software Architectures

The granularity of a component is an issue of component design, which has a
strong effect on the component's reusability. On the one hand, large grained
components are justified by the observation that most users of a computer pro-
gram only use a subset of the program's functionality. Hence, translated to com-
ponents, this means the more functionality a component offers (i.e. the larger
grained the component is), the more users will find their requested functionality
as a subset of the component's functionality and will reuse the component.

On the other hand, adding new functionality does not improve a component's reusability in general, since the added functions translate into new requirements
to the environment where the component is to be embedded. Thus, the compo-
nent becomes less reusable, contrary to the original intention. This we call the
'granularity-reuse problem'. For example, imagine you are designing a printer
management component. If you restrict its functionality to handle only local
printers, it will not be very reusable because it will not handle network print-
ers. However, if you design the component for network printers, it will require
a network even for managing the local printer, and that will not make it very
popular with users.
So the component’s granularity (which is an issue of component design) has a strong influence on system design, since the system has to provide all functionality the component requires. Components of a large granularity usually require more external components than components of a small granularity. But building only fine-grained components is no solution. Although they have less strong external dependencies, they introduce many unnecessary interfaces into the system (which degrades its performance). Furthermore, they are very specific, so less reusable and less attractive for component manufacturers than larger-grained components.

The granularity-reuse-problem arises because components have static provides- and require-interfaces. In all future reuse contexts the component will offer all functionality, which was specified in its provides-interface at design time. Regardless, whether this functionality is completely used or not, also the component will always require all functionality, which is specified in its requires-interface. Hence, what we need, is a mapping of the functionality, which is actually requested from a component by a system, to the functionality, which the component really requires, to provide this requested functionality. For reengineering existing systems, the reverse mapping is also useful (i.e., adapting new components to existing systems). These kinds of mappings are provided by parameterised contracts.

3 Parameterised Contracts

3.1 Interoperability Checks and Classical Contracts

Interoperability checks between components relate to “classical” contracts for software components. A contract for a component specifies, under which pre-condition a component $A$ has to fulfil the post-condition. Translated to interoperability checks, we check if the environment (represented by component $B$) offers all functionality $A$ expects. Thus we check the inclusion of $A$’s requires-interface (precondition) in $B$’s provides interface (see point (1) in figure 1). Is this check successful, $A$ fits into the environment and will offer all services of its provides-interface (postcondition). Hence, interoperability checks have a boolean outcome: a component fits into system or not. But for practical reasons the more interesting and relevant question is, what can we do, if a component does not fit into its environment.

3.2 Parameterised Contracts as Generalisation of Classical Contracts

While interoperability tests check the requires-interface of a component against the provides-interface of another component, parameterised contracts link the provides-interface of one component to the requires-interface of the same component (see points (2) and (3) in figure 1). Classical contracts [7] do not reflect this connection between provides- and requires-interfaces. Classical contracts, once formulated statically during component development, cannot change the post-
Interoperability checks (1) and Requires-parameterised Contract (2) and Provides-parameterised Contract (3)

or precondition according to a new reuse context. This motivates the formulation of two kinds of parameterised contracts:

- provides-parameterised contracts map the provides-interface to a requires-interface.
- requires-parameterised contracts map the requires-interface to a provides-interface.

Technically spoken, parameterised contracts are a mapping which is bundled with the component and computes the interfaces of the components on demand. The requires-parameterised contract takes as arguments the requires-interface of the component and the provides-interface of the environment. Hence, parameterised contracts are isomorphic mappings between the domain of preconditions and the domain of postconditions. Both domains can be modelled as partially ordered sets (posets). (A mathematical discussion of parameterised contracts can be found in [11].) The intersection of a component's requires-interface and the environment's provides-interface describes the functionality which is required by the component and provided by the environment. Out of that information the requires-parameterised contract computes the new provides-interface of the component. Analogously, a provides-interface computes the new requires-interface out of the provides interface of the component and the requires-interfaces of its clients. In our prototypical system [11] we implement parameterised contracts within the component's reflection system.

The following scenarios demonstrate the application of parameterised contracts. Provides-parameterised contracts are useful when designing a new system. A software architect states the functionality a component has to fulfil. Then she finds several candidate components in a repository, which deliver at least the required functionality. For all these candidate components one can compute the functionality they really need in this context via their provides-parameterised contracts. This can be done because the functionality the component has to provide in this context is known.

Requires-parameterised contracts are useful for integrating components into existing systems. Imagine an existing system should be reconfigured with a new component, or an existing system architecture should be enhanced by an existing component. One question in this situation is which functionality the new component will deliver without changing the environment of the component. For creating fault-tolerant system architectures requires-parameterised contracts can be applied at run-time. This is useful because in fault-tolerant systems it is important to know which functionality a component can offer when other components fail.
Parameterised contracts are an abstract concept, which is not tied to a specific interface model. Like most ADLs, parameterised contracts only need the explicit separation of provides- and requires-interfaces. Further, parameterised contracts can be applied at design-time (as strongly proposed in this paper), but also during composition-time (e.g., to integrate components into existing systems) or at run-time (e.g., for graceful system degradation). Like classical contracts, parameterised contracts can be applied for a number of different software units, such as methods, modules, objects and components. (But parameterised contracts may prove most useful when applying to components.)

Applying parameterised contracts to software components means, that the interfaces of the component are recomputed dynamically. The code has not to be manipulated. For practical reasons is is important, that the programmer does not have to foresee and program all possible component adaptations in advance, which are computable by parameterised contracts.

For components with strongly connected provided and required services parameterised contracts do not make sense. E.g., for a screen-saver without access to a graphic-device a requires-parameterised contract cannot compute a meaningful provides-interface.

4 Example

As mentioned, parameterised contracts are not tied to a specific interface model. Here we will show the application of parameterised contracts for a simple interface model (signature lists) and a more sophisticated one (including protocol information, modelled with finite state machines).

As an example, regard a multi-media video-mail component. This mail not only contains the video itself, but also offer functionality to present the video. This design is useful, if you want to abstract away from specific video file formats or if you want to handle different media (like text, sound, and video) in the same manner. The videoMail component makes use of two other (system-specific) components: videoPlayer and soundPlayer. The functionality offered by video-player contains in its provides-interface the methods start, stop, pause, volumeUp, volumeDown, and certain methods to adjust the picture, like brightnessUp, brightnessDown, etc. In case videoMail arrives on a system without sound support (e.g., due to hardware reasons) a require-parameterised contract computes a restricted provides-interface without the methods volumeUp and volumeDown.

Additionally to signature-lists, it is often valuable to include more information in components’ interfaces, such as a description of valid call sequences to the component’s services [5, 8]. This description of supported call sequences for the provides protocol of a component. The unrestricted provides protocol of videoMail (figure 2, left) models the allowed sequences of method calls to videoMail with a finite state machine. In practice, the set of allowed call sequences depends on the specific reuse context (i.e., the call sequences the underlying videoPlayer supports). So one can imagine that a requires-parameterised
Fig. 2. The provides protocol of the videoMail component. Unrestricted version (left), restriction to specific reuse context by require-parameterised contract (right).

contract restricts the functionality of videoMail in a specific reuse context to the protocol shown with the finite state machine in figure 2 (right). Note that modeling the protocol reveals functionality changes which are not expressible with signature lists, such as the availability of method increaseUp.

The technical realisation of parameterised contracts including the linkage between interfaces is described in [10] for an interface model with finite state machines. [11] extends this algorithms to an interface model of greater practical relevance which also can model common container classes like stacks, which cannot be modelled exactly with finite state machines.

5 Conclusions

We presented parameterised contracts as a generalisation of interoperability checks between components (i.e., classical contracts of components). Applying parameterised contracts to software components allows to perform adaptations of components automatically, which restrict the component’s provides- or requires-interface. The base of parameterised contracts lies in the observation that in most practical cases the provides-interface and the requires-interface of a component are not isolated: A component will offer less functionality if its environment offers not all functionality the component requires (to provide all programmed functionality). And, a component will require less functionality, if not all offered functionality of the component is to be used by its clients. Hence, it enhances the component’s reusability to compute the component’s provides-interface out of its required-interface and vice versa.

The linkage between a component’s provides- and requires-interface is independent from the concrete interface model. So we can apply parameterised contracts to simple signature-list-based interfaces, but also to protocol-based interfaces, like demonstrated in the examples.

To revisit question three of the introduction (how to conclude from local component properties to global system properties), we can apply parameterised contracts also to interface models which describe quality attributes. So we can compute quality attributes of combined components like we can compute the
functionality of components adapted to others, as demonstrated in this paper. For example, as parameterised contracts relate provided and required functionality of a component, they also can relate the reliability offered by a component to the reliability of the external services used by the component.

References