Adapting Components and Predicting Architectural Properties with Parameterised Contracts

Ralf H. Reussner

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

Abstract. [1] While interoperability tests check the inclusion of a component's requires interface in the environmental provides interfaces, parameterised contracts link the provides- 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. Combining parameterised contracts themselves results again in a parameterised contract, describing the properties of a component assembly. We present to common connection styles, which can be used to conclude from local component properties to global architectural properties.

1 Introduction

In component oriented programming (COP) software components are regarded as units of late composition [17]. 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., [4, 6, 18]). 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., [7, 16]) looks at structural interactions (i.e., connectors) between components and abstracts away from concrete interface models. More recent work in software architecture (like [5]), also consider dynamic reconfigurations of composed systems, or take a more process-oriented view to apply software architectures to create product families [1]. 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.

[1] This paper is an extension of [14].

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 [11] are a generalisation of classical contracts, as defined by Meyer [8]. 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. In section 5 we show how we can compute a parameterised contract for simple forms of component assemblies (pipelined and bundled components). There we present two ways to combine parameterised contracts (corresponding to pipelining and bundling) and mention some inherent properties of these combinations. These combinations of parameterised contracts can be used to predict the properties of the component assembly out of the component's properties. Section 6 concludes and discussed the application of parameterised contracts as a concept to conclude from local component properties to global system properties in general (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 architecture, 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 estimate 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 adaption and to estimate the effort of adaption: (a) Semi-automatic generation of adaptors out of interface information (e.g., [15, 20]). The more complex the adaptor is, the higher the effort of integration (and the likeliness of programming errors). (b) The automatic adaptation of a component by a generator. In case the result of that adaptation suffices the requirements, the component is integratable. 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 contracts 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 relevant 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 program only use a subset of the program's functionality. Hence, translated to components, 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 component 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 printers. 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). Furtheron, 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 reenginering 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 precondition a component A has to fulfil the postcondition. 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 [8] do not reflect this connection between provides- and requires-interfaces. Classical contracts, once formulated statically during component development, cannot change the post- or precondition according to a new reuse context. This motivates the formulation

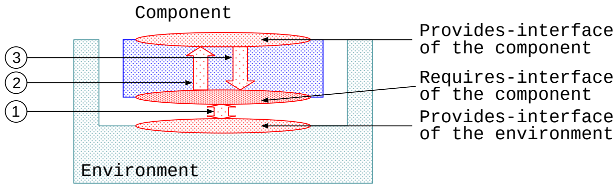

Fig. 1. Interoperability checks (1) and Requires-parameterised Contract (2) and Provides-parameterised Contract (3)

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, parametersised 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 [13].) The intersection of a components requires-interface and the enivoronment prevides-interface describes the functionality which is requires 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 proviodes-interface computes the new requires-interface out of the privides interface of the component and the requires-interfaces of its clients. In our prototypical system [13] 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 system 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. Furtheron, 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 degration). 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.

More generally, parameterised contracts reflect the fact, that the properties of a component cannot be regarded in reality as absolutely fixed attributes of the component. Much more, component properties often strongly depend on the specific context, the component is deployed in. For example, the timing behaviour of a component cannot be given by some fixed numbers of miliseconds, because it depends very muchon the timing behaviour and guarantees of the underlying environment (middleware, OS, hardware). Similarily, the reliability of a component depends on the reliability of its environment.

For components with strongly connected provided and required services parameterised contracts may 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 [6, 9]. 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 se-

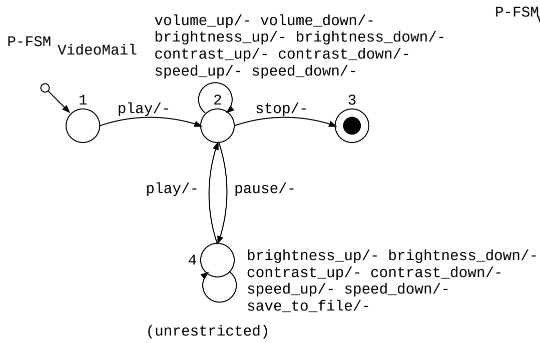

Fig. 2. The provides protocol of the `videoMail` component. Unrestricted version (left), restriction to specific reuse context by require-parameterised contract (right)

quences depends on the specific reuse context (i.e., the call sequences the underlying `videoPlayer` supports). So one can imagine that a requires-parameterised 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 `constrastUp`.

The technical realisation of parameterised contracts including the linkage between interfaces is described in [12] for an interface model with finite state machines. [13] 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 Combining Parameterised Contracts to Predict Architectural Properties

To revisit question three of the introduction (how to conclude from local component properties to global system properties), we can combine parameterised contracts to reflect the combination (assembly) of components. As argued before, parameterised contracts are well-suited to describe properties (functional and non-functional) of components, since a parameterised contract makes the existing relations between provides- and require interface explicit.

When regarding the composition of components as a component itself (and the benefits of hierarchical decomposition during design suggest us to do so), we would like to describe the composition of components with a parameterised contract again. Hence, the question is, how to combine parameterised contracts of components in a way, that the combination of parameterised contracts itself is again a parameterised contract which describes the assembly of these components. Therefore we have to consider, that the combination of parameterised contracts must reflect the way the components interact (i.e., the architectural pattern: e.g., pipe-and-filter [3, 16], master-worker, clients-and-server, etc.).

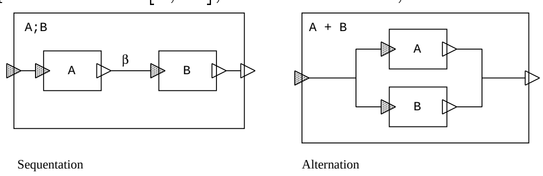

Fig. 3. Combinations of parameterised contracts: sequentiation (left) and alternation (right)

Two common ways to combine components are (a) pipelining of components and (b) bundling components (meaning that several components together provide their services directly to clients, for example behind a facade). For both combinations of components corresponding combinations of parameterised contracts exist. For pipelined components we combine parameterised contracts by sequentiation (figure 3 (left)), for bundled components parameterised contracts are combined by alternation (figure 3 (right)). Both kinds of combinations of parameterised contracts have their specific properties: The provides interface of the sequentiated parameterised contract will never be stronger than the provides interface of component A. But it can be restricted if B's provides interface does not include A's require interface completely. Analogously, the requires interface of the sequentiated component will never be stronger than the requires interface of component B. But it can be weaker, if component A requires only a true subset of B's requires interface. In contrast, an alternated parameterised contract is the sum of the parameterised contracts of the inner components, hence it enhances the functionality of the inner component's provides interfaces and requires interfaces. Therefore it can be used the describe the effects of plug-ins. A mathematical treatment of these intuitive arguments is given in [13].

Sequentiation and alternation are two very simple forms of component assemblies with respect to the combination of parameterised contracts. When looking at concurrent usage of one component (a server) by several other components (clients), issues of synchronisation must be considered when predicting properties of these component assemblies. In some cases synchronisation can be handled by the architectural style, in other cases specific generated connectors can ensure certain synchronisation properties (like fairness or absence of deadlocks) [15].

6 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.

Future work will be concerned with models of more complicated combinations of parameterised contracts beyond sequentation and alternation. This includes communication patterns like publisher-subscriber and others. To conclude from local component properties to global architectural properties two issues are of concern: (a) the inclusion of the properties to reason about in the interface model of a component, and (b) the proove of certain additional global system properties. For example, for talking about the timing behaviour of an architecture it is not sufficient to reason about the timing of the single components, but also it must be sure, that no deadlocks of the component interaction destroy any timing properties of the system. Here transactional systems (e.g., [19]) and other modelling techniques, appropriate for global concurrency analysis (such as Büchi-Automata [2] or Petri-Nets (e.g., [10])), may be useful.

References

1. Jan Bosch. *Design and Use of Software Architectures – Adopting and evolving a product-line approach.* Addison-Wesley, Reading, MA, USA, 2000.
2. J. R. Buechi. On a decision problem in restricted second order arithmetic. In E. Nagel et al., editor, *Proceedings of the second international conference on logic, methodology and philosophy of science*, pages 1–11. Stanford University Press, 1960.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley & Sons, New York, NY, USA, 1996.

4. Jun Han. Temporal logic based specification of component interaction protocols. In *Proccedings of the 2nd Workshop of Object Interoperability at ECOOP 2000*, Cannes, France, June 12.–16. 2000.
5. Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, Reading, MA, USA, 2000.
6. Bernd Krämer. Synchronization constraints in object interfaces. In Bernd Krämer, Michael P. Papazoglou, and Heinz W. Schnmidt, editors, *Information Systems Interoperability*, pages 111–148. Research Studies Press, Taunton, England, 1998.
7. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. *Lecture Notes in Computer Science*, 989:137–155, 1995.
8. Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.
9. Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, October 1993.
10. Wolfgang Reisig. *Petrinetze*. Springer-Verlag, Berlin, Germany, 2 edition, 1986.
11. Ralf H. Reussner. Parameterised Contracts for Software-Component Protocols. Presentation given at Oberon Microsystems, Zürich, http://liinwww.ira.uka.de/~reussner/zuerich00.ps.gz, December 2000.
12. Ralf H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In *34th Hawaiin International Conference on System Sciences*. IEEE, January 3–5 2001.
13. Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Dissertation, Fakultät für Informatik, Universität Karlsruhe, Am Fasanengarten 5, D-76128 Karlsruhe, Germany, July 2001. to appear.
14. Ralf H. Reussner. The use of parameterised contracts for architecting systems with software components. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001. to appear.
15. Heinz W. Schmidt and Ralf H. Reussner. Automatic Component Adaptation By Concurrent State Machine Retrofitting. Technical Report 25/2000, Fakultät für Informatik, Universität Karlsruhe (TH), Am Fasanengarten 5, D-76128 Karlsruhe, Germany, 2000. This report appeared simultaneously as Technical Report No. 2000/81 of the School of Computer Science and Software Engineering, Monash University, Melbourne, Australia.
16. Mary Shaw and David Garlan. *Software Architecture*. Prentice Hall, Englewood Cliffs, NJ, USA, 1996.
17. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press, Addison-Wesley, Reading, MA, USA, 1998.
18. A. Vallecillo, J. Hernández, and J.M. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *ECOOP '99 Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, 1999.
19. Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 2001.
20. D. Yellin and R. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.