# Component Composition with Parametric Contracts

Ralf H. Reussner, Steffen Becker, and Viktoria Firus
ralf.reussner@informatik.uni-oldenburg.de
steffen.becker@informatik.uni-oldenburg.de
viktoria.firus@informatik.uni-oldenburg.de

Software Engineering Group, University of Oldenburg
OFFIS, Escherweg 2, 26121 Oldenburg, Germany

**Abstract.** We discuss compositionality in terms of (a) component interoperability and contractual use of components, (b) component adaptation and (c) prediction of properties of composite components. In particular, we present parametric component contracts as a framework treating the above mentioned facets of compositionality in a *unified* way.
Parametric contracts compute component interfaces in dependency of context properties, such as available external services or the profile how the component will be used by its clients. Under well-specified conditions, parametric contracts yield interfaces offering interoperability to the component context (as they are component-specifically generated). Therefore, parametric contracts can be considered as adaptation mechanism, adapting a components provides- or requires-interface depending on connected components. If non-functional properties are specified in a component provides interface, parametric contracts compute these non-functional properties in dependency of the environment.

## 1 Introduction

"Software components are for composition." [1, pp. 3]. Technically, compositionality of software components includes interoperability checks, adaptations to make components interoperable and compositional reasoning on properties of the component assembly. To support interoperability checks, automated component adaptation and compositional reasoning on properties, appropriate information has to be given by the components' interfaces.

Current object based interface models are not appropriate for software components for (at least) three reasons:

1. Object interfaces model only provided services, not the required services. As argued in section 2, contractual use of components is only possible, if a component not only specifies the services offered, but also the services required

for proper operation. Interoperability checks between two components $A$ and $B$ depend on the specification of both: the services $A$ requires from $B$ and the services $B$ offers (to $A$, or to any other component).

2. Object interfaces include only signatures. Adding behavioural specifications and quality attributes significantly increases the power of interoperability tests (i.e. the class of statically detectable errors). Errors due to wrong service call sequences or insufficient Quality of Service (QoS) can be detected (and hence excluded) before using the software.

3. Objects have fixed interfaces. An object interface corresponds to the functionality implemented by the object. This also holds for component interfaces and components. However, the deployment context of a component is variable. This deployment context heavily influences (a) the functionality offered (or effectively required) by the component, (b) the protocol and (c), most obviously quality attributes such as performance or reliability [2].

Consequently, to support composition, besides provides interfaces a component needs (a) additional information in interfaces, (b) requires interfaces and (c) an abstract model of the relation between provided and required services. The later is given in this paper by parametric contracts. Parametric contracts are formally discussed in [2]. They are used to compute context dependent interfaces and have been deployed for predicting the component reliability in dependency of its context [2]. In the following we assume a component ($C$) has a provides interface ($\mathrm{P}_C$) and a requires interface ($\mathrm{R}_C$).

The contribution of this paper is twofold: (a) to discuss the role of parametric contracts for software component composition and (b) to provide a syntax for specifying *parametric contracts* within component specification. Further on this paper discusses the importance of tool support for generating parts of the specifications.

The structure of this paper is as follows. The term "contractual use" of software components and its relation to interoperability checks is clarified in section 2. Parametric contracts for signatures, protocols and quality attributes are introduced thereafter. In section 4 we discuss their role for software composition, in particular to the above mentioned facets of composition, interoperability, adaptation and prediction. The importance of tool support for the specification of parametric contracts is discussed in section 5. There we also highlight why the specification of component behaviour by parametric contracts is no contradiction to a black-box view on components. After the presentation of related work (section 6), we conclude with a summary and the discussion of open issues and future work in the last section.

## 2 Contractual Use of Components and Interoperability Checks

Much of the confusion about the term "contractual use" of a component comes from the double meaning of the term "use" of a component. The "use" of a component often refers to the following:

1. the usage of a component during run-time. This is, calling services of the component, like calling `TransferFunds` on a payment component.
2. the usage of a component during composition time. This is, placing a component in a new reuse-context, like it happens when architecting systems, or reconfiguring existing systems (e.g., updating the component).

Depending on the above case, contracts play a different role. Contracts are assumed to be known, as they are well known in software engineering literature [3]. Instead of explaining the design by contract paradigm again the essence is summarized here by the following sentence in a general form:

> If the client fulfils the precondition of the supplier, the supplier will fulfil its postcondition.

Let's get back to the two types of the "use" of a component. It is clear, that the component plays the role of a supplier in both cases. In order to specify contracts the pre- and postconditions as well as the clients additionally need to be identified. The use of a component during run-time is obviously simply calling the components services. Hence, the clients of the component are all the components calling a service of the supplying component, which are all those components connected to the provides interface of the supplier. The pre- and postconditions involved in this case are simply those specified for the affected service itself. Therefore it should be evident that this type of use is nothing different as using a method. Thus this case should be called the use of a *component service* instead of the use of a *component* and is being disregarded in the following.

The other case of component usage (usage at composition time) is the actual important case, when talking about the contractual use of components. This is the case, when architecting systems out of components or deploying components within existing systems for reconfigurations. Consider a component $C$ acting as supplier, and the environment acting as client. The component offers services to the environment (i.e., the components connected to $C$'s provides interface(s)). According to the above discussion of contracts, these offered services are the postcondition of the component, because it is that, what the client can expect from a working component. Also according to Meyers above description of contracts, the precondition is that, what the component expects from its environment (actually all components connected to $C$'s requires interface(s)) to be provided by the environment, in order to enable $C$ to offer its services (as stated

in its postcondition). Hence, the precondition of a component is stated in its requires interfaces.

Analogously to the above single sentence formulation of a contract, we can state:

> If the user of a component fulfils the components' requires interface (offers the right environment) the component will offer its services as described in the provides interface.

Note that checking the satisfaction of a requires interface includes checking if the contracts of required services (the service contracts specified in the requires interface(s)) are sub-contracts of the service contracts stated in the provides interfaces of the required components. A detailed description of subcontracts can be found in [4, p. 573]. The contractual use of components enables interoperability checks that can be performed when architecting new systems or replacing components during system maintenance [2].

There is a range of formalisms used for specifying pre- and postconditions, defining a range of interface models for components (see for extensive discussions and various models e.g., [5–7]).

## 3 Parametric Contracts

**Motivating Scenarios**
In daily life of component reuse, a component rarely fits directly in a new reuse context. For a component developer it is hard to foresee all possible reuse contexts of a component in advance (i.e., during design-time). One of the severe consequences for component oriented programming is that one cannot provide the component with all the configuration possibilities which will be required for making the component fit into future reuse contexts. Coming back to our discussion about component contracts, this means, that in practice one single pre- and postcondition of a component will not be sufficient. Consider the following two scenarios:

1. the precondition of a component is not satisfied by a specific environment while the component itself would be able to provide a meaningful subset of its functionality.
2. a weaker postcondition of a component is sufficient in a specific reuse context (i.e., not the full functionality of a component will be used). Due to that, the component will itself require less functionality at its requires interface(s), i.e., will be satisfied by a weaker precondition.

Hence, what we need are not static pre- and postconditions, but *parametric contracts*. In the first case a parametric contract computes the postcondition which

is computed in dependency of the strongest precondition guaranteed by a specific reuse context (hence the postcondition is parametric with the precondition). In the second case the parametric contract computes the precondition in dependency of the postcondition (which acts as a parameter of the precondition). For components this means, that provides- and requires-interfaces are not fixed. A provides interface is computed in dependency of the actual functionality a component receives at its requires interface, and a requires interface is computed in dependency of the functionality actually requested from a component in a specific reuse context. Hence, opposed to classical contracts, one can say:

> Parametric contracts link the provides- and requires-interface(s) of the same component. They have a range of possible results (i.e., new interfaces).

Interoperability is a special case now: if a component is interoperable with its environment, its provides interface will not change. If the interoperability check fails, a new provides interface will be computed.

Mathematically, parametric contracts are modelled by a function $p$ mapping a provides interface $P$ to the minimal requires interface $R = p(P) = \mathrm{R}_C$ specifying the needs of $P$. Hence $p$ is a function of the set $\mathrm{Prov}_C$ of all possible provides interfaces of $C$ to the set $\mathrm{Req}_C$ of all possible requires interfaces of $C$. A possible provides interface is any interface offering a subset of the functionality implemented in $C$, the set of all possible requires interfaces is the image of $\mathrm{Prov}_C$ under $p$. Note that $p$ not necessarily is an injective function: several different provides interfaces may be mapped to the same requires interface. Consequently, the inverse mapping, associates to each requires interface of $R \in \mathrm{Req}_C$ a *set* of supported provides interfaces. To yield a single provides interface, we use the "maximum" element of this set. Formally, this element is the smallest upper bound of the set $p^{-1}(R)$. This smallest upper bound is the join of the elements of $p^{-1}(R)$ which exists because if provides interfaces $P_1$ and $P_2$ are elements of $p^{-1}(R)$ each of their elements (i.e., services, service call sequences, services with QoS annotations) is supported, consequently, the interface describing the set of all these elements is also itself element of $p^{-1}(R)$ $(P_1, P_2 \in p^{-1}(R) \Rightarrow P_1 \cup P_2 \in p^{-1}(R))$. For later use, we define the shorthand inv-$p$ : $\mathrm{Req}_C \rightarrow \mathrm{Prov}_C$ as inv-$p(R) := \bigcup_{E \in p^{-1}(R)} E$. (Note that we use the more intuitive set-oriented notion of $\cup$ for the join-operator which is commonly referred to as $\sqcap$ in literature on lattices, etc.)

Like for a classical contracts, the actual parametric contract specification depends on the actual interface model and should be statically computable. In any case, there's no need for the software developer to foresee possible reuse contexts. Only the specification of a bidirectional mapping between provides- and requires-interfaces is necessary.

Resulting from this there is a need for the component supplier to specify the parametric contract in the component's specification to enable anyone trying

to reuse the component to determine the component's capabilities in a certain environment or to learn about the environment one has to provide to get the needed functionality. To achieve this we propose in the following syntactical notations and the according semantics which allows the specification of parametric contracts.

**Signatures**
In addition to the list of provided services and the list of required services we need a mapping between every provided service and the respective required services. This means that for each provided service a list of required external services must be provided by the component developer or has to be extracted by code analysis tools. When computing the actual provides interface a service is only included in the provides interface, if all its required services are provided by the environment.

Hence, the specification of the interface of a specification framework to be enhanced should be expanded by the specification of a parametric contract using the following syntax. The contract specification can simply be appended to the definition of the respective interface (i.e. an IDL description).

The syntax of the proposed specification can be taken from the following extended BNF:

```
parametricContract ::= "parametric contract {"
                          (ServiceEffectSpecification)+ "}"
ServiceEffectSpecification ::= SerivceID "{"
            ((ServiceIDExtern ", ")* ServiceIDExtern | "") "}"
ServiceID ::= Identifier
ServiceIDExtern ::= Identifier "::" Identifier
```

**Grammar 1:** Parametric contract specification for signature lists

For the rest of this paper a payment component should be regarded which makes use of parametric contracts. The example component is capable of performing funds transactions either by requesting a bank transfer or by the use of the customers credit card information. As a matter of fact, the supplied credit card information needs to be validated before a transaction will be accepted. For this, a remote component hosted by the credit card company is being called. As the usage of the validation component is dependent on the payment of a monthly fee the component will not be available in every environment for economic reasons. Further on the component needs a database connection for caching purposes. The database connection is also needed for performing bank transactions because it contains a mapping between the bank codes and the names of the respective banks. Nevertheless bank transactions can be offered without the credit card validation services so that the component might still be useful in environments not providing these services. Hence, the component producer decided to

use a parametric contract to reflect this scenario in the components implementation in order to offer the component to a larger group of customers.

Using this example the payment component and its parametric contract may be specified as depicted in example 1.

```
interface Payment
{
  void CreditCardPayment (in double amount, in CreditCardInformation inf);
  void BankTransferPayment (in double amount, in BankAccountInformation inf);
}
interface extern
{
  void CreditCardValidator::ValidateInformation (in CreditCardInformation inf);
  DBConnection DB::GetConnection ();
}
parametric contract
{
  CreditCardPayment { CreditCardValidator::ValidateInformation,
                      DB::GetConnection }
  BankTransferPayment { DB::GetConnection }
}
```

**Example 1:** Example parametric contract

It can be seen that the component only offers the service `CreditCardPayment` if the required service `CreditCardValidator::ValidateCardInformation` is available as it is described in the use case above. If no database is available the component is unable to offer any services any more.

**Protocols**

If a component producer specifies the interface of the component by the protocol the component provides, one has to specify (supported by existing tools) for each offered service which call *sequences* are required for its correct execution [7]. The component specification therefore has to include the so called service effect specification which is a description of every possible control flow of a specific service call. The requires protocol is not stated explicitly. It is being calculated dynamically out of the service effect information at configuration time as it is depending on the actual part of the provides protocol being used by the component's clients [2]. Notice that the service effect needs to be a sequence of calls leading from a defined start state to a defined end state, e.g. if there is the need for housekeeping functions they have to be requested as well.

In general, protocols can be specified using two different approaches. One possibility is to specify all permitted sequences of service calls, the other one is to describe which sequences are prohibited. Using the first approach a description of all eligible call sequences has to be specified. The set of allowed call sequences describes the provides protocol of the component. If the second approach is being favoured the specification needs to state under which conditions a preceding service call is prohibited.

As stated before we focus on the support of static interoperability checking. Hence, the used notation for the component protocol has to enable the deployment of efficient algorithms for checking if a given protocol is included in an other protocol. Although finite state machines are limited regarding their expression power they enable inclusion checks to be evaluated efficiently. For this reason we use finite state machines to express permitted call sequences for the rest of this paper.

There are additional reasons for preferring the specification of allowed call sequences over the specification of prohibited sequences. The reasons are summarized in the following enumeration.

1. Its easier for the producer of the component to specify the allowed order of service calls. In case of incomplete specifications, a missing allowed order is not causing harm (although it restricts the usability of the component). However, missing a prohibited sequence in a specification can lead to unexpected behaviour.
2. Tools can be used to perform automatic analysis of message sequence charts (MSCs) or workflows specified in similar languages. Further on it is possible to perform control flow analysis on the provided byte code to extract the service effect specification as mentioned below.
3. As we aim at predicting QoS properties of component configurations there is the need to determine the call sequences the component performs on external component services. This information is needed to estimate the probability of a call to a specific external service.

The following is a proposal for a syntax which may be used to specify the needed FSMs. An example is depicted and specified in figure 1 where a requires and a provides protocol was specified by the use of UML state charts. States are specified by their identifier and the additional information if the state is a start or a final state. Only one state is allowed to be a start state but any number of final states may be used. For complexity reasons the automaton we expect to be supplied should be deterministic. Transitions are described by their start and final state and the operation triggering the given transition.

```
FSM ::= "fsm { states { " (STATE ", ")* STATE
          " } transitions { " TRANS* " } }"
STATE ::= IDENTIFIER STARTSTATE FINALSTATE
TRANS ::= "(" IDENTIFIER "," IDENTIFIER "," IDENTIFIER")"
STARTSTATE ::= " ISSTARTSTATE" | ""
FINALSTATE ::= " ISFINALSTATE" | ""
```

**Grammar 2:** Grammar for specifing protocols

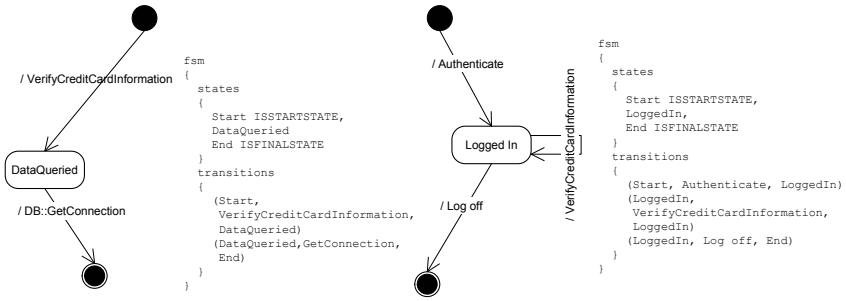


**Fig. 1.** An example requires and provides protocol

A component supplier has to specify the provided component protocol as given in figure 1. Additionally one has to specify the service effect specification. We propose to use a FSM according to the given grammar for these specifications as well.

**Quality of Service**

For extra-functional properties, the application of parametric contracts is crucial. For example, one cannot specify the timing behaviour of a software component as a fixed number. Much more, the timing properties of a component offered in its provides interface is always a function of the environment's timing behaviour, as received at its requires interfaces. The same argument holds for reliability as empirically validated in [2]. By sequencing parametric contracts of single components which form a component architecture (without cyclic dependencies) one can compute the overall architectural properties by sequencing the single parametric contracts and applying them (as a function) to the properties of the underlying environment where the system is deployed. (How to deal with multiple provides- or requires-interfaces is discussed in detail in [7, 8].)

For QoS specifications we assume the usage of QML [9] at interface level. QML adds quality attributes to single services. There are two possible cases:

1. A component is depending on the services of other components. In this case the provides interface needs to be calculated from the actual selection of the components providing the needed services. The specification should therefore provide enough information to allow the estimation of the quality attributes at the provides interface if all required services are known.
2. A component is independent of other services or at least they are unknown at the time the system is assembled (e.g. consider a WebService, internally it might consist of further components but that is unknown or unchangeable by the configurator). In this case the needed quality attribute values need to be determined by monitoring or other techniques. Remember that some kind of reference architecture needs to be specified in this case as well to get comparable figures.

The specification in the second case is clear. It is simply an interface description with additional QML constructs specifying the needed quality attributes. In the first case we need a specification of the service effect interface specified as FSM as described in section 3. Additionally we now need probabilities describing - given a certain state - which transition might be performed next. As a result the specification schema given in the previous section needs to be expanded by those probabilities which results in the following modification to the BNF given before:

```
TRANS ::= "(" IDENTIFIER "," IDENTIFIER "," IDENTIFIER "," PROBABILITY ")"
PROBABILITY ::= [0..1] AS DOUBLE
```

**Grammar 3:** Extension for reliability specifications

This information is sufficient for predicting accurately the components reliability [2].

## 4 Component Composition and Parametric Contracts

To discuss compositionality with parametric contracts more concretely, we introduce the following shorthand for the simple, yet exemplary case where $n$ components $C_i$ are composed to a composite component $C = C_1 \circ \cdots \circ C_n$. As the components $C_i$ can be either basic or composed, and we assume the composition operator as associative, we will in the following only consider the case $C = A \circ B$ with $A := C_1$ and $B := C_2 \circ \cdots \circ C_n$. Obviously, we neglect the existence of different composition operators and the discussion whether they are commutative, etc. However, note that a component $C_i$ itself can be composed by several sub-components by other composition operators.

As we consider components as black-box entities (i.e., solely described by their interfaces), the composition operator $\circ$ computes the interfaces of $C$ as a function of $A$'s and $B$'s interfaces $((\mathrm{P}_C, \mathrm{R}_C) = (\mathrm{P}_A, \mathrm{R}_A) \circ (\mathrm{P}_B, \mathrm{R}_B))$. As a shorthand for this, we write $C = A \circ B$.

In the following we discuss the afore mentioned different facets of compositionality, using this shorthand.

**Interoperability**
The well-formdness of $C$ requires that $A$ and $B$ interoperate. Checks for interoperability are boolean functions checking for the composition operator $\circ$ the interoperability of two components $(A, B)$: $\mathrm{Interop}_\circ(A, B) \in \{\mathrm{false}, \mathrm{true}\}$. In case of `false` as a result, $C$ does not exist, i.e.,

$$A \circ B = \begin{cases} (\mathrm{P}_A, \mathrm{R}_B) & \text{if } \mathrm{Interop}_\circ(A, B) = \mathrm{true} \\ \bot & \text{else} \end{cases} \quad (1)$$

In principle, interoperability is the absence of any interoperability error. However, statically (i.e., by analysing interface information but not executing the components) one can only prove the absence of *some classes* of interoperability errors. Hence, interoperability often is defined in respect to a class of absent interoperability errors (i.e., no signature matching errors, no protocol related errors, etc.) The importance of *static* interoperability checks arises because, (a) one is interested in interoperability while constructing / reconfiguring systems, i.e., before systems are operated by the user (for obvious reasons), (b) because one does not want to restrict oneself to executable components but also wants to make use of abstract components solely described by interfaces [1] (c) execution-based interoperability *tests* (opposed to static *checks*) also cannot prove the absence of all interoperability errors (assuming the number of ways using a component is unlimited).

**Adaptation**
As mentioned above, the composition operator is undefined in case of a negative outcome of the interoperability check. Adaptation is concerned with bridging interoperability errors, i.e., adapting a component in a way, that $\text{Interop}_\circ$ is `true`. (This adaptation of a component may happen by an adapter used by clients instead of the original component and leaving the original component unchanged). However, this view of adaptation is equivalent to a function $\text{Adapt}_\circ$ taking the two components $A$ and $B$ as arguments, and returning the composed component $C$ if adaptation was possible or $\perp$ if adaptation is not possible. At least $\text{Adapt}_\circ$ can adapt one (or both) of its argument components $A$ and $B$ and define $C$ in terms of $A$'s and $B$'s interfaces. Except from that, $\text{Adapt}_\circ$ can also have the possibility of changing $A$'s and $B$'s interfaces (for making them interoperable) and define $C$ in terms of these altered interfaces.

Using the parametric contracts $p_A, p_B$ as introduced in section 3,

$$A \circ B = (\text{inv-}p_A(\mathrm{R}_A \sqcap \mathrm{P}_B), p_B(\mathrm{R}_A \sqcap \mathrm{P}_B)) \tag{2}$$

Here $\mathrm{R}_A \sqcap \mathrm{P}_B$ refers to the "intersection" of $\mathrm{R}_A$ and $\mathrm{P}_B$, i.e., the interface, describing what $B$ provides of that what $A$ demands. (Again, the actual $\sqcap$ operator depends on the interface model used: it is a simple set-intersection for signature-lists, while a cross-product for protocols modelled by finite state machines or the maximum operator for a quality measure.) This refers to the steps $1a$ and $2a$ in Figure 2. The first scenario in section 3 where a component $B$ does not provide all functionality another component $B$ requires, refers to step 1 in Figure 2, where a weakened provides interface of $C$ is computed, as $A$ cannot provide all its implemented functionality, because its requires interface is not fulfilled completely. The second scenario in section 3 where a component (in our example $A$) uses not all functionality another component ($B$) provides would

[1] Note, that the usage of abstract components conflicts with the widely accepted notion of a component by Clemens Szyperski, requiring a component to be executable (possibly on an abstract virtual machine) by definition [1].

result in our example in a weakened requires interface of $C$, as component $B$ will not require its strongest requires interface in this specific context where $A$ does not request $B$'s complete functionality. This corresponds to step 2 in Figure 2.

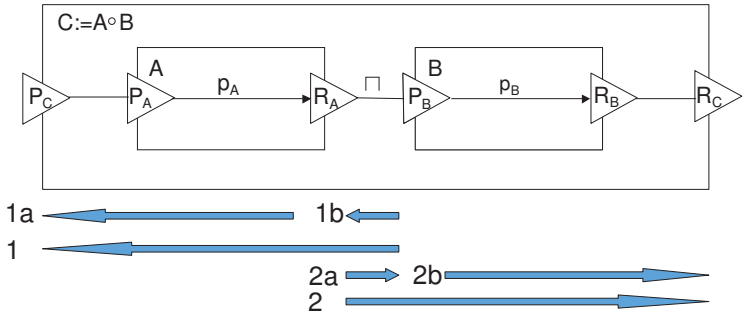


**Fig. 2.** Compositional reasoning and parametric contracts

Of particular interest are approaches for *automated* adaptation, for at least two reasons: (a) generation is usually cheaper, faster and less error prone than manual programming of adapters if the domain is well understood and the adapter description language is less complex than a "normal" programming language. (b) Support of one of the hallmarks of component-based software development: the ability to know component properties before deployment in a new system as the component has already been deployed (and observed) in different contexts previously. If a manually written adapter is used together with a component having well-known properties, most likely, the adapted component will behave differently to the original component without adapter. This annihilates the benefit of using a component with well-known properties. Opposed to a manual adapter, for a generated adapter it is much easier to predict its changes on a component's properties. This is because the standardised generation process eases the formal modelling of the adapters transformation of component properties.

Note that adaptation by parametric contracts is a special case of boolean interoperability checks: if $A$ and $B$ are interoperable, $(\mathrm{P}_A, \mathrm{R}_B) = \mathrm{Adapt}_\circ(A, B)$.

**Property Prediction / Compositional Reasoning**
Reasoning about the properties of a composition by the properties of its parts is a crucial ability of any engineering construction process. As mentioned above, adaptation can be seen as computing context dependent interfaces of the inner components $A, B$ and using them as interface parts of the outer component $C$. Most interestingly, exactly this reasoning on the properties of the composition in terms of the properties of the single components and the composition-operator

used, is done by the above function $\text{Adapt}_\circ$. Therefore, it is more a matter of the kind of properties considered whether to speak on adaptation or prediction of properties. For various reasons the propagation of changed functional properties (such as different signatures, omitted services, different protocols) from inner interfaces (such as $P_A$ or $R_B$) to the outside composed component is called adaptation, while the propagation of changed quantitative quality attributes (such as various performance metrics, reliability and availability) is called prediction. However, in both cases compositional reasoning is applied.

## 5 Tool Support

Parametric contracts with their specification of internal dependencies between provides- and requires-interfaces do not conflict with the black-box delivery of components. This is because, tools can generate the specifications of parametric contracts by byte-code analysis. Parametric contracts for signature-list interfaces are yielded by a classical use-analysis where each implemented service's code is analysed for external methods called. Parametric contracts for protocols-modelling interfaces can be yielded by control-code analysis [7]. Parametric contracts for reliability prediction base on those for protocol-modelling interfaces and add transition probabilities. In [2] it is shown, that educated guessing these transition probabilities results only in small prediction errors under specific conditions.

## 6 Related Work

The grounding work of Bertrand Meyer on contracts forms the basis for parametric contracts, as cited in section 2. Besides the already cited QML, related work on component specification mainly deals with specifying component protocols. As the FSM approach used in this paper is limited in expression power, other ways of describing component behaviour have been researched. These descriptions have been expressed in different formalisms, each having specific advantages and drawbacks [10–12, 2], such as linear-timed logic (LTL) ([13, 14]) or Petri-nets [15, 16]. When considering finite call sequences, the use of (QP)LTL is equivalent to the FSM approach in term of power of specification and the analysis one can perform [17, p. 1024]. However, when transforming LTL expressions into finite state machines one has to consider the possible state explosion. The Petri-net approach is in general more powerful in modelling and the set of feasible analyses differs from finite state machines. Successful research was also undertaken to make the state-machine approach more powerful without loosing its possibilities of efficient analyses [7]. However, in general there is a trade-off between a formalism's expression power and the set of feasible analyses which can be performed within this formalism.

## 7 Conclusions

The use of parametric contracts has been presented as a general framework describing different facets of compositionality such as component interoperability (as a special case), adaptation and property prediction in a unified way. The concrete specification of parametric contracts was given for three kinds of interface models, namely a signature-list based interfaces, protocol-modelling interfaces and interfaces containing reliability values as QoS information. Currently, parametric contracts as a means of component specification are included in the specification framework of the working group 5.10.3 of the German Computer Science Society (G.I. e.V.) [18]. Future work will be directed to the specification of parametric contracts of additional QoS attributes like performance. Therefore, the use of different specification and analysis formalisms for time are investigated, such as timed automata [19].

## References


1. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. 2 edn. ACM Press and Addison-Wesley, New York, NY (2002)
2. Reussner, R.H., Poernomo, I.H., Schmidt, H.W.: Reasoning on software architectures with contractually specified components. In Cechich, A., Piattini, M., Vallecillo, A., eds.: Component-Based Software Quality: Methods and Techniques. Number 2693 in LNCS. Springer-Verlag, Berlin, Germany (2003) 287–325
3. Meyer, B.: Applying "design by contract". IEEE Computer **25** (1992) 40–51
4. Meyer, B.: Object-Oriented Software Construction. 2 edn. Prentice Hall, Englewood Cliffs, NJ, USA (1997)
5. Krämer, B.: Synchronization constraints in object interfaces. In Krämer, B., Papazoglou, M.P., Schnmidt, H.W., eds.: Information Systems Interoperability. Research Studies Press, Taunton, England (1998) 111–141
6. Vallecillo, A., Hernández, J., Troya, J.: Object interoperability. In Moreira, A., Demeyer, S., eds.: Object Oriented Technology – ECOOP '99 Workshop Reader. Number 1743 in LNCS. Springer-Verlag, Berlin, Germany (1999) 1–21
7. Reussner, R.H.: Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten. Logos Verlag, Berlin (2001)
8. Schmidt, H.W., Reussner, R.H.: Generating Adapters for Concurrent Component Protocol Synchronisation. In: Proceedings of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems. (2002)
9. Frolund, S., Koistinen, J.: Quality-of-service specification in distributed object systems. Technical Report HPL-98-159, Hewlett Packard, Software Technology Laboratory (1998)
10. Campbell, R., Habermann, N.: The Specification of Process Synchronization by Path Expressions. In: Proc. Int. Symp. on Operating Systems. Volume 16 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (1974) 89–102
11. Nierstrasz, O.: 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. (1993) 1–15

12. Yellin, D., Strom, R.: Protocol Specifications and Component Adaptors. ACM Transactions on Programming Languages and Systems **19** (1997) 292–333
13. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, New York, USA (1992)
14. Han, J.: Temporal logic based specification of component interaction protocols. In: Proccedings of the 2nd Workshop of Object Interoperability at ECOOP 2000, Cannes, France (2000)
15. Petri, C.A.: Fundamentals of a theory of asynchronous information flow. In: Information Processing 62, IFIP, North-Holland (1962) 386–391
16. Ling, C., Schmidt, H.W.: A concept of time in workflow modelling and analysis. Technical Report 2000/72, School of Computer Science and Software Engineering, Monash University, VIC 3168 Australia (2000)
17. van Leeuwen, J.: Formal Models and Semantics, Handbook of Theoretical Computer Science. Volume 2. Elsevier Science Publishers, Amsterdam, The Netherlands (1990)
18. Overhage, S.: Towards a standardized specification framework for component discovery and configuration. In Weck, W., Bosch, J., Szyperski, C., eds.: Proceedings of the Eigth International Workshop on Component-Oriented Programming (WCOP'03). (2003)
19. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco, CS (1996)