Specifying Contractual Use, Protocols and Quality Attributes for Software Components

Steffen Becker, Ralf H. Reussner, Viktoria Firus
Software Engineering Group, Department of Computing Science
University of Oldenburg, Germany
becker|reussner|firus@informatik.uni-oldenburg.de

August 28, 2003

Abstract

We discuss the specification of signatures, protocols (behaviour) and quality of service within software component specification frameworks. In particular we focus on (a) contractually used components, (b) the specification of components with variable contracts and interfaces, and (c) of quality of service. Interface descriptions including these aspects allow powerful static interoperability checks. Unfortunately, the specification of constant component interfaces hinders the specification of quality attributes and impedes automated component adaptation. This is because, especially quality attributes heavily depend on the components context. To enable the specification of quality attributes, we demonstrate the inclusion of *parameterised contracts* within a component specification framework. These parameterised contracts compute adapted, context-dependent component interfaces (including protocols and quality attributes). This allows to take context dependencies into account while allowing powerful static interoperability checks.

1 Introduction

A specification framework for components has to provide information for several purposes, such as component retrieval and assessment, component deployment, interoperability checks, automated component adaptation, etc.

Current specification frameworks include signatures of the services offered by a component (e.g., UDDI [1]) or comprise additional metadata to classify components (easing the retrieval) and to specify additional component attributes (such as quality) [2]. In any case, specification frameworks for components include the "classic" interface models (signature-list based interface models), stemming from object based middleware, such as the CORBA-IDL [3].

However, using object based interface models is 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 3, the 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 checks (i.e. the class of detectable errors). Errors due to wrong service call sequences or insufficient quality of service 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. It 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 [4, 5].

The contribution of this paper is a syntax for including *parameterised contracts* into the specification framework of the working group 5.10.3 of the German Informatics society (G.I. e.V.) [2]. Parameterised contracts (as introduced in [6] and formally discussed in [4, 7]) compute context dependent contracts (i.e., provides- and requires-interfaces) and have been deployed for predicting the component reliability in dependency of its context [7]. Further on this paper discusses the importance of tool support for generating parts of the specifications proposed in this paper.

The structure of this paper is as follows. In the following section some prerequisites are mentioned. The term "contractual use" of software components and its relation to interoperability checks is clearified in section 3. Parameterised contracts for signatures, protocols and quality attributes are introduced afterwards. The importance of tool support for the specification of parameterised contracts is discussed in section 5. 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 Prerequisite

In contrary to the specification framework which we choose to base our specification tasks on, we differentiate strictly between a component and its interfaces because of the possibility to differentiate between the specification (interface) and the actual implementation. Those interfaces can be further divided into two categories: provided interfaces and required interfaces.

The concepts presented in this paper rely on the separation of a component on the one hand side, represented by the actual implementation of the component, and on the other hand on one or more interfaces specifying the services offered by the component to potential external users. As we will show in section 4 the provides-interface is calculated dynamically during composition time. Therefore it should be seen not tiedly coupled to the component as the same implementation may expose different interfaces depending on the reuse context in which the component is deployed.

Further on it is important for our work to distinguish between provided and required interfaces. The first describe services offered by a component, the later services needed by the component (i.e., services from other components). Components connected to any interface of the component form the *environment* of the component.

Although the need of requires-interfaces is obvious for static interoperability and substitutability check (and well-known in literature [8, 9]), current component models like Sun's EJB or Microsoft's .NET only contain provides-interfaces (one notable exception is CORBA 3.0). As we aim at providing tools to support those interoperability checks during component configuration we focus our work on supplying the necessary specifications needed to perform this task.

A sub task of checking the interoperability of components is the matching of signature names. Imagine one component producer calling a provided function `StartFundsTransfer` and an other one simply calling it `TransferFunds`. We assume that the matching of those names can be done by the means of the normative language specified on the terminological layer of the components specification (e.g. if all the components use the same normative language) or it has to be done manually during composition time by the configurator. Therefore we assume for the rest of this paper that signature names have been matched already.

The specification framework we utilize in this paper already fulfils the stated requirements except that a component may not have multiple interfaces. Interface specifications are separated from other specification attributes by the use of a dedicated layer for this technical information. The provides-interface is modelled on this layer as well as required services of external components. Parameterised contracts specify a link between the required and provided interfaces.

Finally it is worth mentioning that we concentrate solely on the technical aspects of the component specification. The impact of this work on the remaining layers of the specification framework is outside the scope of this paper.

3 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 refers often 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 [10] and are also included in the regarded specification framework on the behavioural layer. 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 which is acting as a 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’ required interface (offers the right environment) the component will offer its services as described in the provided interface.

Note that checking the satisfaction of a requires-interface includes checking whether 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 [11, p. 573]. The contractual use of components enables interoperability checks that can be performed when architecting new systems or replacing components during system maintanance [4].

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., [12, 13, 14]). This leads naturally to different kinds of contracts for components [15].

4 Parameterised Contracts

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 cases:

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 *parameterised contracts* [4, 14]. In the first case a parameterised contract computes the postcondition which is computed in dependency of the strongest precondition guaranteed by a specific reuse context (hence the postcondition is parameterised with the precondition). In the second case the parameterised 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, but a provides-interface if 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:

> Parameterised 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.

Like classical contracts, parameterised contracts depend 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 parameterised contract in the components specification to enable anyone trying to reuse the component to determine the components 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 parameterised contracts in the initially mentioned specification framework.

4.1 Signatures

The specification framework utilizes CORBA-IDL to specify interfaces as primary notation. As mentioned earlier the provided interface and a list of external services (required interface) are already included in the specification. Therefore there is only the need to add information about the parameterised contract of the component.

CORBA-IDL uses signature lists to specify the services of a component. 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 on the interface layer of the utilized specification framework should be expanded by the specification of a parameterised contract using the following syntax. The contract specification can simply be appended to the IDL definition of the respective interface.

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

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

For the rest of this paper a payment component should be regarded which makes use of parameterised 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 parameterised 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 parameterised contract may be specified as follows on the interface layer of the specification framework.

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

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.

4.2 Protocols

If the 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 [14]. 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 [7]. 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 Quality of Service 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 required and a provided 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" | ""
```

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.

4.3 Quality of Service

For extra-functional properties, the application of parameterised contracts is crucial. For example, one cannot specify the timing behaviour of a software component as a fixed number. Much more, the tim-

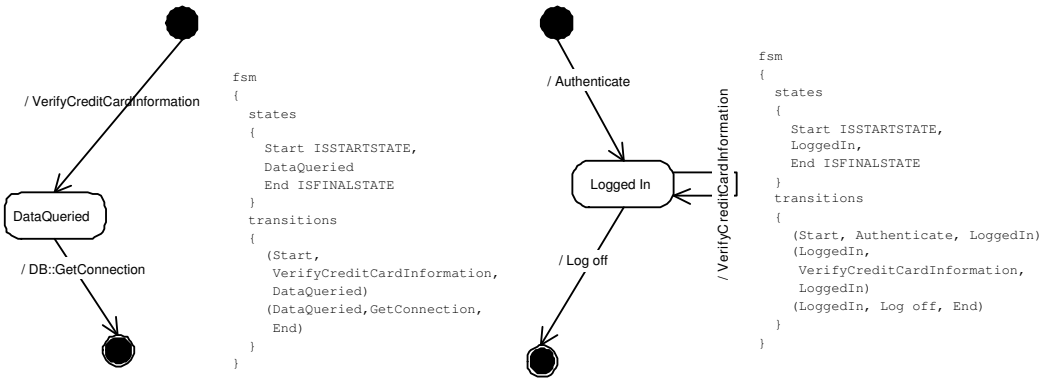

Figure 1: An example required and provided protocol

ing 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 [16]. By sequencing parameterised contracts of single components which form a component architecture (without cyclic dependencies) one can compute the overall architectural properties by sequencing the single parameterised contracts and applying them (as a function) to the properties of the underlying environmental where the system is deployed. (How to deal with multiple provides- or requires-interfaces is discussed in detail in [14, 17].)

For Quality of Service specifications we assume the usage of QML [18] 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 in CORBA-IDL 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 4.2. 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
```

This information is sufficient for predicting accurately the components reliability [16].

5 Tool Support

As one can image the needed specifications for parameterised contracts might become quite large and complex but fortunately a lot of information which needs to be specified can be generated by the support of tools [19, 16].

As shown in figure 2 there are several possible sources for the specification data to result from. An important aspect of any specification is to assure the consistency between the specification and the specified object or component in our context. If the specification is generated from the used modelling diagrams or even from the code of the component it's easier to keep the specification and the actual implementation in sync which gives an important cost benefit.

It also raises the question who is responsible for adding the specification data. The utilized specification framework assumes that the component producer/vendor supplies the complete specification. In this case the component is used as black-box. Tools which are able to analyse the code of a component enable the user of the component to generate missing specifications. In this case we speak of grey-box reuse as the client uses additional information (e.g. the code) to determine needed information.

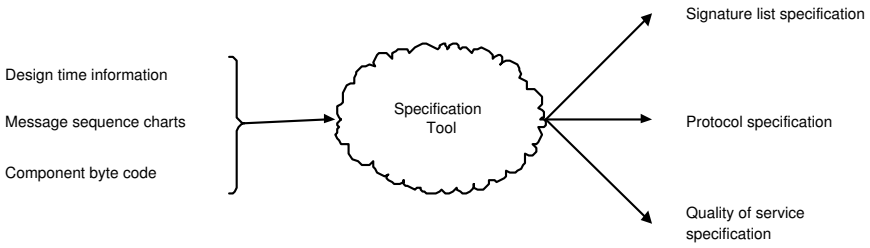

Figure 2: Tool support for the generation of parameterised contract specifications

6 Related Work

We propose the use of FSMs to specify protocol and QoS information. Nevertheless as this approach is limited in expression power, research has been done to find other ways of describing component behaviour. These descriptions have been expressed in different formalisms, each having specific advantages and drawbacks [20, 21, 22, 23], such as linear-timed logic (LTL) ([24, 25]) or Petri-nets [26, 27]. 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 [28, 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 [29]. 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

In this paper we focused on explaining what needs to be specified to add parameterised contracts to an existing specification framework. We distinguished between three levels of specification data whereby higher levels contain lower ones. On the lowest level we introduced signature list based specifications. In a second step we added behavioural information by specifying protocols with finite state machines.

On the highest level Quality of Service attributes were added to the component specification. Finally, the importance of tools in the highlighted context was emphasised.

Future work will be directed to the specification of additional Quality of Service attributes like safety, fairness or liveness. The specification data will then also be used to predict the corresponding properties of complete component architectures.

References

[1] The UDDI standardization consortium, "The UDDI homepage." http://www.uddi.org.

[2] J. Ackermann, F. Brinkop, S. Conrad, P. Fettke, A. Frick, E. Glistau, H. Jaekel, O. Kotlar, P. Loos, H. Mrech, E. Ortner, U. Raape, S. Overhage, S. Sahm, A. Schmieten, T. Teschke, and K. Turowski, "Standardized specification of business components," *Memorandum of the working group 5.10.3, Component Oriented Business Application Systems*, 2002.

[3] Object Management Group (OMG), "The CORBA homepage." http://www.corba.org.

[4] R. H. Reussner and H. W. Schmidt, "Using Parameterised Contracts to Predict Properties of Component Based Software Architectures," in *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002* (I. Crnkovic, S. Larsson, and J. Stafford, eds.), Apr. 2002.

[5] R. H. Reussner, "Contracts and quality attributes of software components," in *Proceedings of the Eigth International Workshop on Component-Oriented Programming (WCOP'03)* (W. Weck, J. Bosch, and C. Szyperski, eds.), June 2003.

[6] R. H. Reussner, "Parameterised Contracts for Software-Component Protocols." Presentation given at Oberon Microsystems, Zürich, http://liinwww.ira.uka.de/∼reussner/zuerich00.ps.gz, Dec. 2000.

[7] R. H. Reussner, I. H. Poernomo, and H. W. Schmidt, "Reasoning on software architectures with contractually specified components," in *Component-Based Software Quality: Methods and Techniques* (A. Cechich, M. Piattini, and A. Vallecillo, eds.), no. 2693 in LNCS, pp. 287–325, Springer-Verlag, Berlin, Germany, 2003.

[8] N. Wirth, *Programming in MODULA-2*. Springer-Verlag, 3rd Edition, 1985.

[9] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *Proceedings of ESEC '95 - 5th European Software Engineering Conference*, vol. 989 of *Lecture Notes in Computer Science*, (Sitges, Spain), pp. 137–153, Springer-Verlag, Berlin, Germany, 25–28 Sept. 1995.

[10] B. Meyer, "Applying "design by contract"," *IEEE Computer*, vol. 25, pp. 40–51, Oct. 1992.

[11] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, USA, 2 ed., 1997.

[12] B. Krämer, "Synchronization constraints in object interfaces," in *Information Systems Interoperability* (B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, eds.), pp. 111–141, Taunton, England: Research Studies Press, 1998.

[13] A. Vallecillo, J. Hernández, and J. Troya, "Object interoperability," in *Object Oriented Technology – ECOOP '99 Workshop Reader* (A. Moreira and S. Demeyer, eds.), no. 1743 in LNCS, pp. 1–21, Springer-Verlag, Berlin, Germany, 1999.

[14] R. H. Reussner, *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[15] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, pp. 38–45, July 1999.

[16] R. H. Reussner, H. W. Schmidt, and I. Poernomo, "Reliability prediction for component-based software architectures," *accepted at Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 2002.

[17] H. W. Schmidt and R. H. Reussner, "Generating Adapters for Concurrent Component Protocol Synchronisation," in *Proceedings of the Fifth IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, Mar. 2002.

[18] S. Frolund and J. Koistinen, "Quality-of-service specification in distributed object systems," Tech. Rep. HPL-98-159, Hewlett Packard, Software Technology Laboratory, Sept. 1998.

[19] G. Hunzelmann, "Generierung von Protokollinformation für Softwarekomponentenschnittstellen aus annotiertem Java-Code," diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, Apr. 2001.

[20] R. Campbell and N. Habermann, "The Specification of Process Synchronization by Path Expressions," in *Proc. Int. Symp. on Operating Systems*, vol. 16 of *Lecture Notes in Computer Science*, pp. 89–102, Springer-Verlag, Berlin, Germany, 1974.

[21] O. Nierstrasz, "Regular types for active objects," in *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, vol. 28, 10 of *ACM SIGPLAN Notices*, pp. 1–15, Oct. 1993.

[22] D. Yellin and R. Strom, "Protocol Specifications and Component Adaptors," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, pp. 292–333, 1997.

[23] R. H. Reussner, "Enhanced component interfaces to support dynamic adaption and extension," in *34th Hawaiin International Conference on System Sciences*, IEEE, Jan. 3–5 2001.

[24] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, USA, 1992.

[25] J. 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.

[26] C. A. Petri, "Fundamentals of a theory of asynchronous information flow," in *Information Processing 62*, pp. 386–391, IFIP, North-Holland, 1962.

[27] C. Ling and H. W. Schmidt, "A concept of time in workflow modelling and analysis.," Tech. Rep. 2000/72, School of Computer Science and Software Engineering, Monash University, VIC 3168 Australia, June 2000.

[28] J. van Leeuwen, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, vol. 2. Amsterdam, The Netherlands: Elsevier Science Publishers, 1990.

[29] R. H. Reussner, *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Dissertation, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, July 2001.