Contracts and Quality Attributes of Software Components*

Ralf H. Reussner
Software Engineering Group, Department of Computing Science
University of Oldenburg, Germany
reussner@informatik.uni-oldenburg.de

Iman H. Poernomo, Heinz W. Schmidt
School of Computer Science and Software Engineering
Monash University, Australia
{ihp|hws}@csse.monash.edu.au

Abstract

We argue that quality attributes of a component are most often not a constant property. Muchmore, the quality of a component heavily depends on the specific usage context. Therefore, we present a specification method for contractually specified components which does not specify quality attributes as constants but as functions to be evaluated at deployment. The contribution of this paper is threefold: (a) We clarify the term "contractual use of components", a term which is often misstated or used inconsistently in current literature. We therefore translate the "design-by-contract"-principle to components. (b) We generalise component contracts to parameterised contracts, taking the mentioned context-dependencies of components into account. (c) We finally demonstrate how parameterised contracts are used to compute the reliability of software components. Results from an empirical evaluation confirm the strong context-dependency of a component's reliability but also show that parameterised contracts for reliability prediction are easy to specify by automated control-flow analysis.

1. Introduction

Quality attributes of components are gaining increasing attraction in the CBSE community. Here we use the term "quality attribute" for extra-functional properties which are externally visible to the user, such as reliability of performance (opposed to extra-functional properties intrinsic to the software as maintainability or reusability). At least two reasons for the recent attraction of quality attributes in component-based software engineering (CBSE) can be identified: (a) Component users wish to use knowledge on the quality of a component to make a purchase decision. Among components with similar functional properties, the knowledge on quality attributes often could be a most significant information, if specified. (b) In the recent past, two of the most promising areas of software engineering, namely software architecture and software components moved closer together. One of the major motivations of software architectures, the aim to reason explicitly on extra-functional properties during software-design may benefit a lot from focusing on *component based* software architectures. Consequently, the problem of predicting quality attributes of the overall architecture by known component-qualities gains more attraction [2, 13].

If a component specifies a certain quality of service in its provides-interfaces, the component also must request a specific quality of service from its context (as the components calls itself services of this environment). The concept "the component offers some quality of service, if the component's expecations are met by its context" naturally leads to contract-principle of B. Meyer [9] with pre- and postconditions. Hence, to specify quality of service for components, we first review the term "contractual use" of software components in section 3 and translate Meyer's design-by-contract principle to components.

*This position paper extends the paper "Using Parameterised Contracts to Predict Properties of Component Based Software Architectures", a position paper presented at the CBSE Workshop at the IEEE ECBS conference in Lund, Sweden, April 2002 and summarises recent results.

Beyond classical contracts, we present in section 4 a generalisation of contracts, called *parameterised contracts* [11], dealing with the prediction of functional component properties as well as component quality attributes by taking the component context into account.

The importance of a component's context for the component's properties becomes clear, when looking at quality attributes like timing behaviour or reliability. Here, the timing behaviour (reliability) of the component clearly depends on the timing behaviour (reliability, resp.) the of environmental services used by the component. In addition, the reliability also depends on the usage profile of the component. The usage profile itself is also clearly not fixed but a part of the component's variable context.

2. Example

Figure 1 shows as an example a composite component (`MobileMailViewer`) which offers the service of displaying mails of various formats to a mobile personal organiser. Internally, the `MobileMailViewer` consists out of a `Controler` (handling the selection of mails, connection to an address book, formatting of strings, etc.) and a `MailServer` (delivering the mails) and a `ViewerSoftwareServer`, which provides the `Controler` with the viewers appropriate to the format of the actual email and its possible attachment. (Since memory is limited on mobile devices, the device itself cannot store viewers for all formats. Also the programmer of the controler cannot foresee all future mail or attachment formats in advance.) Both servers are remote for the `Controler` component. Nevertheless, the personal manager program on the mobile device

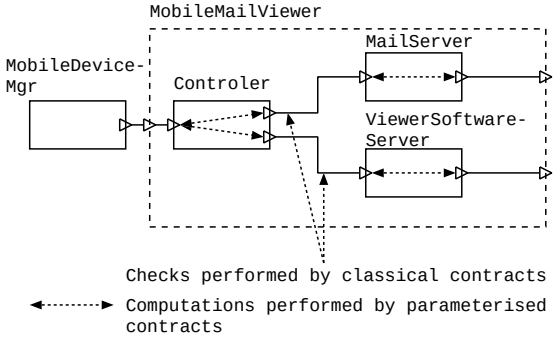

Figure 1. Configuration of a mobile viewer

considers the `MobileMailViewer` as a single component, located locally.

In our figure, rectangles denote components, triangle denote interfaces. Components have two kinds of interfaces: provides- and requires-interfaces. The first describe services offered by a component, the latter services required by the component (i.e., services from other components). Components connected to a component's interfaces form the *environment* of the component. In our example, the `Controler` component requires the `MailServer` and the `ViewerSoftware-Server` and offers services to the `MobileDeviceManager`. Although the need of requires-interfaces is obvious for interoperability and substitutability check (and well-known in literature [15, 8]), current component models like Sun's EJB or Microsoft's .NET only contain provides interfaces. (One notable exception is CORBA 3.0. Within Microsoft's COM-model, one were able to model requires-interfaces via the "ConnectionPoint" pattern).

3. Contractual Use of Components in Software Architectures

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 one of the following:

1. the usage of a component during run-time. This is, calling services of the component, like calling `displayMessage` of the `Controler` 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.

Before actually defining contracts for components, we briefly review the design-by-contract principle from an abstract point of view. According to [10, p. 342] a contract between the client and the supplier consists of two obligations:

- The client has to satisfy the precondition of the supplier.
- The supplier has to fulfil its postcondition, if the precondition was met be the client.

Each of the above obligations can be seen as the benefit for the other party. (The client can count on the postcondition if the precondition was fulfilled, while the supplier can count on the precondition). Putting it in one sentence:

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

It is clear, that a used component plays the role of a supplier. But to formulate contracts for components, we also have to identify the pre- and postconditions and the user of a component. This depends on the above case of usage (run-time of composition-time). Let's first consider the component's use at run-time. The use of a component at run-time is calling its services. Hence, the user of a component C are all components connected to C's provides interface(s).

The precondition for that kind of use is the precondition of the service, likewise the postcondition is the postcondition of the service. Hence, this kind of use of a component is nothing different than using a method. Therefore, the authors do consider this case as the use of a *component service*, but *not* as the use of a *component*. Likewise, the contract to fulfilled here from client and supplier is a *method contract* as described by Meyer already 1992. There is nothing component specific in this kind of contracts!

The other case of component usage (usage at composition time) is the actually 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. Again, in this case a component C is acting as a supplier, and the environment as a client. The component C 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 (as postconditions describe what the client can expect from a working component). Also according to the Meyers above description of contracts, the precondition describes what the component C expects from its environment (i.e., all components connected to C's requires-interface(s)). Only if this precondition is met, C offers 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. The notion of a subcontract is described in [10, p. 573] like contravariant typing for methods: A contract c' is a subcontract of contract c, if (a) the precondition of c' is weaker than or equal to the precondition of c and (b) the postcondition of c' is stronger than or equal to the postcondition of c.

For checking the correct contractual use of the `Controler` component of our example we check, whether the services specified in the requires-interface of `Controler` are included in provides-interface of `MailServer` and the contracts of requires services are subcontracts of provided services (likewise we have to check the binding between the other requires-interface of `Controler` and the provides interface of `ViewerSoftwareServer`). In general, we can state:

> The interfaces involved by contract checking belong to separate components and are connected by bindings (figure 1). Checking contractual use of components are interoperability checks and therefore have a boolean result.

Hence, when architecting systems (i.e., introducing new components), we have to check the bindings between requires-interfaces and the used environmental provides-interfaces. When replacing a component with a newer one, we not only have to check their contract (i.e., the bindings between their requires-interfaces and the used components, like mentioned above), but also the contracts of the using environmental components (i.e., the bindings from the provides-interfaces), because one has to ensure, that by a replacement none of the existing local contracts has been broken. In our example, this means, if we replace the `Controler` component we (a) have to check the contractual use of `Controler` (i.e., we check the precondition of the `Controler` for the interoperability with `MailServer` and `ViewerSoftwareServer`), and (b) we have to check whether the precondition of `MobileDeviceMgr` is still fulfilled (i.e., checking the contractual use of `MobileDeviceMgr`).

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., [6, 14, 11]). This leads naturally to different kinds of contracts for components [1].

Another degree of freedom in the abstract principle of design-by-contract is the time of their deployment. Component contracts as discussed here describe the deployment of components at composition-time. This stresses the importance of contracts which are statically checkable. When a system is architected or reconfigured, one is aware of the possibility of introducing errors. Therefore, the direct feedback about the success of introducing (or replacing) a component into a system is very helpful in practice, because it can assure the absence of composition errors. Opposed to that, run-time checks can only show the presence of composition errors when detecting a contract violation. While this is helpful for debugging, it is particularly bad when reconfiguring existing systems. In this case the person using the system and triggering the error is most commonly not the person reconfiguring or architecting the system. Even worse, in case of several subsequent reconfigurations, the administrtatior finds it hard to trace back the reconfiguration step which introduced the error.

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 the component developer finds it hard to provide the component with 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 praxis one single pre- and postcondition of a component will not be sufficient:

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 (e.g., the `ViewerSoftwareServer` in our example may fail or even being completely absent in a different architecture, but the `Controler` can be still present standard text emails, although cannot display specific attachments).
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* [11]. In case one, 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 case 2 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 (see fig. 1). They have a range of possible results (i.e., new interfaces).

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

5. Applications of Parameterised Contracts

Like classical contracts, parameterised contracts depend on the actual interface model. In any case, the software developer do not have to foresee possible reuse contexts but has to provide a bidirectional mapping between provides- and requires-interfaces. For CORBA-IDL like signature list based interfaces, this means that for each provided service a list of required external services must be provided by the component developer. When computing the provides-interface a service would only be included in the provides-interface, if all its required services are provided by the component linked to the requires-interfaces. If interfaces also describe component protocols, one has to specify for each offered service which call *sequences* are required for its correct execution [11]. This specification task can be simplified by tools (algorithms and a prototypical implementation are described in [4]).

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 timing properties of a component as 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 shown in the following.

6. Context-Dependent Reliability Prediction for Software Components with Parameterised Contracts

Software reliability is defined as the probability that the software operates according to the user's expectations. Hence, reliability is anti-proportional to the MTBF (mean time between failure), i.e., the mean time to a failure (MTTF) plus the mean time to repair (MTTR) [5]. Usually, one assumes a constant MTTR and records (e.g., by statistical testing or system monitoring) the MTTF as a measure of software reliability, as in IBM's cleanroom approach [3]. It becomes clear that reliability (opposed to correctnes) is not relative to a formal system specification but to a given usage profile. Different usage profiles of the same software usually will result in different reliability values.

Determining the reliability of a software component has to take the following two facts into account.

1. The usage profile of the component is not part of the component but of the component's context. Most likely, a component is used differently in different contexts.
2. The component performs calls to external services (via its requires-interface(s)). Consequently, the reliability as percieved by the component user not only depends on the component's code, but also on the reliability of the called external services.

Therefore, the reliability of a software component is not a constant. Muchmore, the reliability of a component has to be modelled as a function having as parameters the usage profile and the reliability of the external services.

In our approach, the parameterised contract is a function computing for the reliability value of each service parameterised with the reliability values of the external (contextual) services. Concretely, the parameterised contract includes for each service provided by the component a Markov-Model which is parameterised with an vector of external services' reliability values. By Markov-Chain analysis one yields the reliability value of each service provided by the component. In a second step, the user can compute the component's reliability, again by Markov-Chain analysis using these reliability values of the provided services and the usage profile of the component.

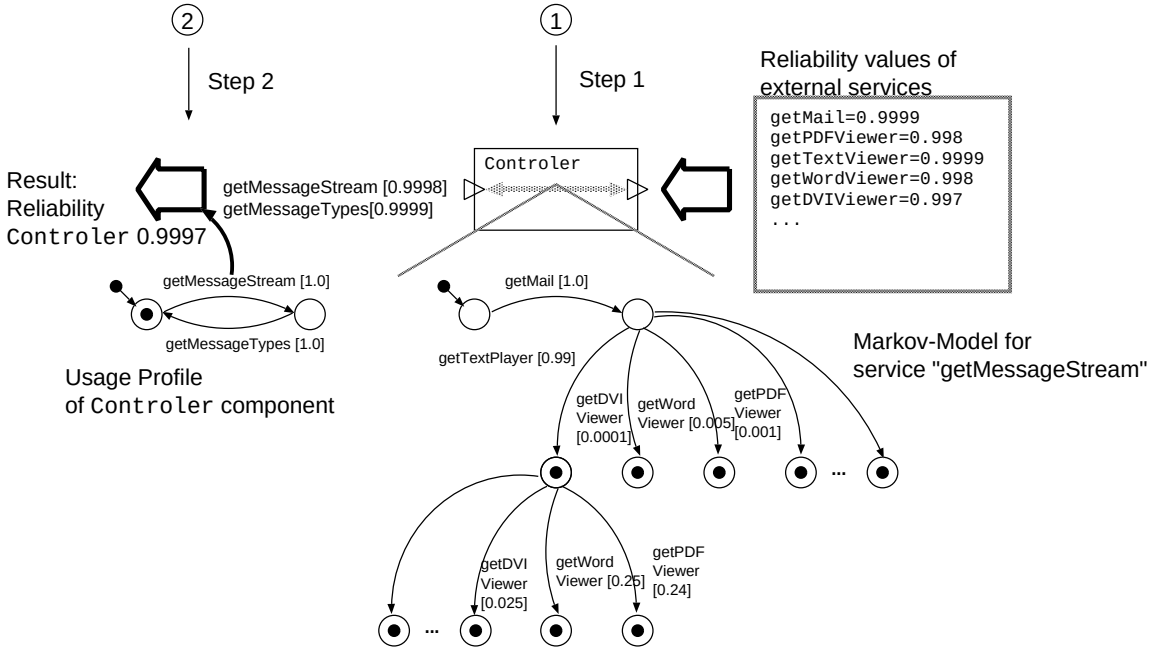

Figure 2. Two-step reliability computation of the `Controler` Component

In figure 2 this two step process is shown for computing the reliability of the `Controler` component of our example system. In the first step, we read the vector of the external services (as given by the components `MailServer` and

`ViewerSoftwareServer`, cf. figure 1) and compute for each service of the `Controler`, namely `getMessage` and `getMessageTypes`, the reliability by Markov chain analysis (details can be found e.g., in [12]). For that we require for each service provided by the `Controler` a Markov Model, specifying the probabilities of the call sequences to external services this service will use. How the explicit specification of such a model can be avoided is discussed in the next section. The result of the first step is the reliability value for each provided service of the `Controler`. This is sufficient information for the most users. However, if the user wants to associate a *single* reliability figure with a component, in the second step the user can compute the reliability of the component, again by Markov Chain analysis using the intended usage profile for this component.

7. Specification of Parameterised Contracts

Even with the benefit of automated analyses, the success of each software engineering technique heavily depends on minimising the costs of its application. As often the main cost arise by additional specification overhead, it is crucial to minimise these specification overhead, e.g., by automated generation of the required specifications.

The application of our technique needs the following inputs:

1. The reliability values for all external services. The information can be gained either by monitoring, or predicting (e.g., by our model itself) or by estimating.
2. The parameterised contract. Here, that means for each provided method a Markov Model describing the usage of external methods.
3. (Only for step two:) The usage profile for the component.

As specifying this information *accurately* may result in considerable costs, in the sequell we discuss the necessity of providing this information accurately.

We used a test-bed for validating our model and for running various experiments [12]. First of all, our experiments show the accuracy of the used Markov models for predicting the reliability. Having accurate inputs, the prediction in the worst case is only 1 % off of the measured reliability values (while on most cases the error is even below 0.1 %).

In one experiment, we changed the reliability of the context and observed the changes in the measured component reliability (cf. figure 3).

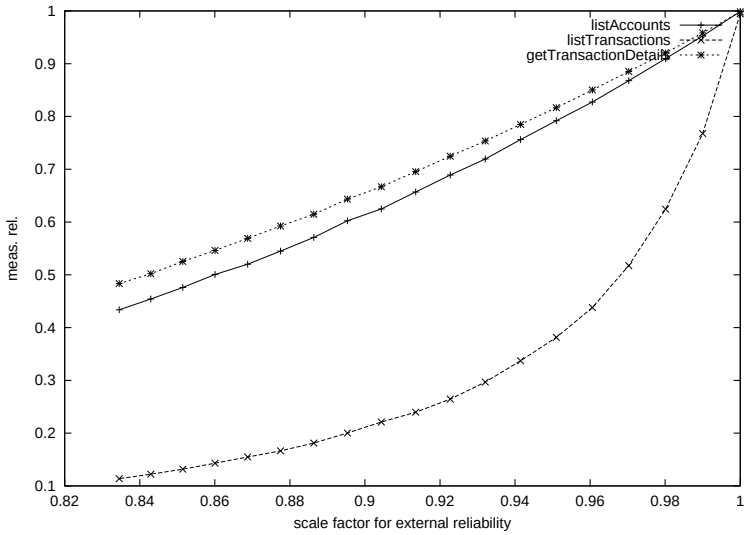

Figure 3. Comparing the reliability of services depending of the external services' reliabilities

These experiments clearly confirm that the reliability of a component can *heavily* depend on the reliability of the environment: only slight changes in the reliability of the services provided by the component context (as shown on the x-axis) result in large changes of the reliability of offered services (especially, if external services are called within loops, as in service `getTransactionDetails`).

The second point in the list of required inputs concerns the prediction model itself, i.e., the Markov models how each provided service uses external methods. In figure 4 we show an experiment within our testbed where the accuracy of the Markov model used for predictions is varied (as shown on the x-axis). Here we are interested in the error our predictions have against the "real" reliability measured (y-axis). In the figure we show the results for two different contexts. "Context one" is most homogenous with respect to reliability, i.e., the reliability of the different services of that context are quite similar (just one order of magnitude difference; details in [12]). Opposed to that, "Context two" includes reliable services and unreliable services (like one encounters when calling services of an intra-net and services of an extra-net). It is to be expected that this is also a quite realistic scene when using webservices from different servers.

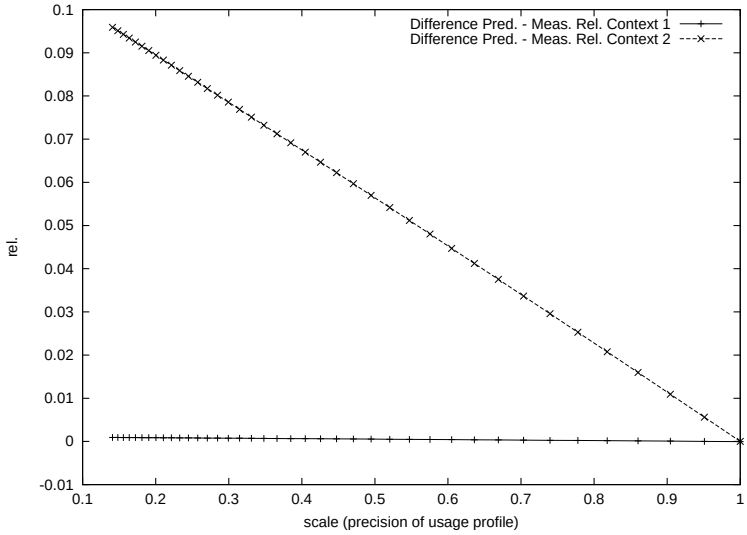

Figure 4. Differences between predicted and measured reliability of varied over the accuracy of the Markov model for two different contexts of external reliability

Most interestingly, in neither context the accuracy really matters. Even without knowing anything of the real Markov-Model (error larger than 50%, as shown on the x-axis) the prediction has less than 10 % error (y-axis). (In "Context one" the prediction is basically completely independent from the accuracy of the input. This is explained by the homogenous context: if all external service have a very similar reliability, is does not matter whether we know exactly which service is called.) So, if the accuracy is of secondary concern, we can yield the Markov models by simply guessing the transition probabilities. For example, by the following heuristic: If a state has n outgoing transitions, we assume (unless knowing better) that each outgoing transition has probability $1/n$. (This can be seen as the application of the "maximum-entropy-principle".) Hence, the only input we actually need is a finite state machine for each provided component service denoting as transitions the calls to internal and external methods. (Later we can inline the internal method calls.) This kind of state machine is basically given by the control-flow graph of a component service which can be yielded by code analysis. Tools for generating these automata by analysing Java source code exist [4] (although this approach is not limited to Java source code).

The last point in the above list of required input, the usage profile for the component, is only needed to compute out of the reliability of the provided services a single reliability figure of the component. As mentioned, mostly the service reliability values are of interest rather than a single component reliability value, hence not having the component usage profile does not hurd. (Although it might be partially derived from Message-Sequence-Charts or from a state machine describing the valid call sequences to provided services. In this context it is of interest that newer industrially used methods, like the BMW reference web architecture applies finite state machines for modelling user inputs [7].)

So the bottom-line is: without information on the context, one cannot say anything about the reliability of the component. But the software component itself can still be delivered as blackbox: the information required to perform a context-sensitive prediction of the component reliability at deployment-time with parameterised contracts can be derived automatically by control-flow analysis from the components code (either the sources or binary code).

8. Conclusion

As a result of discussing the contractual usage of software components, we argued for requires-interfaces as preconditions of components and provides-interfaces as postconditions. Parameterised contracts, linking provides- and requires-interfaces of the same component were motivated by the necessity of computing a component's properties (functional and extra-functional) in dependence of the concrete reuse context.

We emphasise that contracts specifying quality attributes of software components cannot be fixed once and forever by the component vendor. (In case of component reliability our case study clearly demonstrates this.)

In fact, the component contract must be parameterised with contextual properties at deployment-time (or even run-time, if the context changes at run-time). This parameterised contract is to be specified by the component vendor and bundled with the component. Our case study shows that when dealing with the quality attribute "reliability", a parameterised contract automatically generated by control-flow analysis of the code is sufficiently accurate.

Further work in this area includes a formal model for predicting the accuracy of our model in the presence of inaccurate inputs. Beyond that, the application of parameterised contracts for performance prediction seems to be a worthwhile approach, as it is to be expected that also the performance attributes of a component heavily depend on the performance attributes of the concrete component deployment context.

References

[1] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[2] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau. 4th ICSE workshop on Component-Based software engineering: Component certification and system prediction. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 771–772, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[3] M. Dyer. *The Cleanroom Approach to Quality Software Development*. Series in Software Engineering Practice. Wiley & Sons, New York, NY, USA, 1992.

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

[5] A. I. John D. Musa and K. Okumoto. *Software Reliability - Measurement, prediction, application*. McGraw-Hill, New York, 1987.

[6] B. Krämer. Synchronization constraints in object interfaces. In B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, editors, *Information Systems Interoperability*, pages 111–141. Research Studies Press, Taunton, England, 1998.

[7] J. Lind and M. Luber. BMW-Standardarchitektur für web-basierte Anwendungen. *Java-Spektrum*, (11/12):14–20, nov/dec 2002.

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

[9] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.

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

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

[12] R. H. Reussner, H. W. Schmidt, and I. Poernomo. Reliability prediction for component-based software architectures. *Journal of Systems and Software – Special Issue of Software Architecture - Engineering Quality Attributes*, 66(3):241–252, 2003.

[13] J. Stafford and K. Wallnau. Predicting feature interactions in component-based systems. In *Proceedings of the Workshop on Feature Interaction of Composed Systems*, June 2001.

[14] A. Vallecillo, J. Hernández, and J. Troya. Object interoperability. In A. Moreira and S. Demeyer, editors, *Object Oriented Technology – ECOOP '99 Workshop Reader*, number 1743 in LNCS, pages 1–21. Springer-Verlag, Berlin, Germany, 1999.

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