Using Parameterised Contracts to Predict Properties of Component Based Software Architectures

Ralf H. Reussner
CRC for Enterprise Distributed Systems
Technology Pty Ltd
Monash University
900 Dandenong Road,
Caulfield East, VIC 3145, Australia
reussner@dstc.com

Heinz W. Schmidt
Center for Distributed Systems
and Software Engineering
Monash University
900 Dandenong Road,
Caulfield East, VIC 3145, Australia
hws@csse.monash.edu.au

Abstract

This position paper presents an approach for predicting functional and extra-functional properties of layered software component architectures. Our approach is based on parameterised contracts *a generalisation of design-by-contract. The main contributions of the paper are twofold. Firstly, it attempts to clarify the meaning of "contractual use of components" a term sometimes used loosely – or even inconsistently – in current literature. Secondly, we demonstrate how to deploy parameterised contracts to predict properties of component architectures with non-cyclic dependencies.*

1. Introduction

In the recent past, two successful areas of software engineering, namely software architecture and software components moved closer together. In fact, we believe they are just two sides of the same coin, given the intertwining of architectural and detailed design and the need to connect both system-level and component-level reasoning about software.

One of the major motivations of software architecture, the aim to reason explicitly about extra-functional properties during software-design, benefits from focusing on *component based* software architectures. While current research in that area mostly concentrates on component interoperability checks within software architectures and component adaptation in case of incompatibility, predicting properties of the overall architecture from known component properties has gained attraction [2, 11].

From a conceptual point of view, one can consider software architectures as structuring principles and methods for component assemblies. Software architecture and component (re-)configuration are also closely connected. Based on this strong connection, architecture promises help in compositional reasoning, i.e., predicting system properties and qualities from component properties by using the architectural structuring mechanisms in the reasoning process and without recourse to the component internals.

To be able to predict overall architectural properties, local interaction must be correct in terms of the interface model. This means no errors should occur by calling methods with wrong parameters or by calling methods in the wrong order. Due to that necessity of local correctness, we discuss contracts for components first (in section 3). Such contracts specify conditions for correct local interaction. Beyond traditional contracts, in section 4 we present a generalisation called *parameterised contracts* [7, 6]. These deal with the prediction of properties of composite components based on the properties of their basic components. In addition, parameterised contracts depend on the environment in which components are deployed. Some of the environment properties become parameters to these contracts. The importance of this parameterisation becomes clear, when looking at extra-functional properties like timing behaviour or reliability. Here, the timing behaviour (reliability) of the component clearly depends on the timing behaviour (reliability, resp.) of environmental services used by the component.

2. Example

As an example, Figure 1 shows a composite component (`MobileMailViewer`) which offers the service of displaying mails of various formats on a mobile personal organiser. Internally, the `MobileMailViewer` consists of a `Controler` (handling the selection of mails, connec-

tion to an address book, formatting of strings, etc.) and a `MailServer` (delivering the mails) and a `ViewerSoftwareServer`, which provides the `Controler` with the a viewer appropriate to the format of the actual email. Since memory is limited on mobile devices, we assume the device cannot store viewers for all formats. Also the programmer of the controler cannot foresee all future mail or attachment formats. To the `Controler` component, both servers are

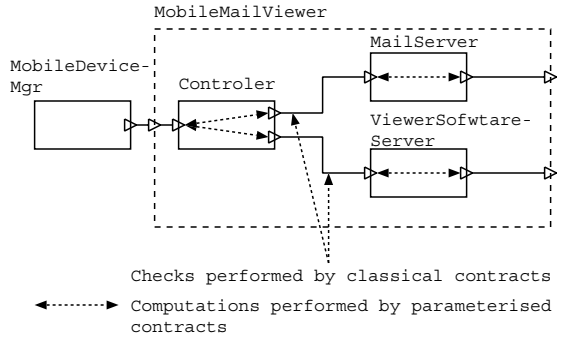

Figure 1. Configuration of a mobile viewer

remote. Nevertheless, the personal manager program on the mobile device considers the `MobileMailViewer` as a single local component.

In our figure, rectangles denote components and triangle denote interfaces. Interfaces are objects themselves not just (meta-)descriptions. Components have two kinds of interfaces: provides- and requires-interfaces. The former describe services offered by a component, the latter services required by the component. Components connected to a components interfaces form the *environment* of the component. In our example, the `Controler` component requires the `MailServer` and the `ViewerSoftwareServer` and offers services to the `MobileDeviceManager`. Although the need of requires-interfaces is obvious for interoperability and substitutability check (and well-known in literature [13, 4]), current component models like Sun's EJB or Microsoft's .NET only contain provides interfaces. (One notable exception is CORBA 3.0).

3. Contractual Use of Components in Software Architectures

Much of the confusion regarding "contractual use" of a component derives from the double meaning of the term "use". It can refer to

1. the *run-time use*, when the methods of a component are called. For example, `displayMessage` of the `Controler` component is used in this way.

2. the *composition-time use*, when a component is placed into a new context or environment. This includes the development time but can also happen when an operational system is reconfigured.

Depending on the above case, contracts play a different role. Before actually defining contracts for components, we briefly review the design-by-contract principle. According to [5, p. 342] a contract between the client and the supplier consists of two obligations for each service or method:

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

Each of the above obligations can be seen as the benefit to the other party. In a nut shell:

> 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 client of a component.

Considering the run-time use first, the clients of a component C are all components connected to C's provides interface(s). The precondition for run-time use is the collection of preconditions of the component's services. Likewise the run-time use postcondition is the collection of postconditions of the provided services. This is still close enough to the traditional design-by-contract. Hence we consider this kind of contract the *service* contract.

The composition-time use or reuse occurs in the architectural design or reconfiguration of a system. From an architectural viewpoint the component C depends on its environment through its requires interfaces. Its correct functioning does not only depend on the preconditions of its provided services. Hence we regard the the requires interfaces as a kind of abstract preconditions at the component contract level. Similarly, from an architectural viewpoint, the provided services are the promised benefits to the (run-time) user. Therefore, we consider those provides interfaces the postconditions of the component-level contract. From the perspective of logical specifiation the story is a little more complex in that each of these conditions is a conditional specification – for instance, the required environment behaviour is only delivered if the component itself satisfies the preconditions of the requires interface services.

Putting it all together we formulate the architecture-by-contract principle as follows:

> If the user of a component fulfils the components' required interface (i.e., the precondition) by offering the right environment the component will offer its services as described in the provided interface (i.e., its postcondition).

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 [5, p. 573] and generalised in [9] using contravariant typing for methods but importantly including invariants as conditions for distributed and hence typically concurrent environments.

For checking the correct contractual use of the `Controler` component in our example we check whether the services specified in the requires interface of `Controler` are included in the provides interface of `MailServer` and whether the contracts of requires services are subcontracts of provides services. Similarly we check the binding between the other requires and provides interfaces.

More generally, when architecting systems (i.e., introducing new components), we have to check the bindings of their requires interfaces to the used environmental provides interfaces in addition to checking the use of the component's provideees interfaces. When replacing a component with a newer one, we not only have to check their contract (i.e., the bindings of their requires-interfaces to 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 non of the existing local contracts have been broken. In our example, if we replace the `Controler` component we have to (a) check the contractual use of `Controler`, i.e., we check the precondition of the `Controler` (i.e., 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., [3, 12, 6]). This leads naturally to different kinds of contracts for components [1].

Another degree of freedom in the abstract principle of design-by-contract and our extension to architecture-by-contract is the time of component 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, errors are common. Therefore, the direct feedback regarding correct component deployment is very helpful in practice, because it can assure the absence of composition errors. In contrast, the run-time checks can detect contract violations at run-time only. In many classes of distributed systems, such late detection of composition errors is unsatisfactory from a quality of service point of view. The problems and costs of late composition error discovery are compounded as the person running the system and triggering the error usually is not the system architect or maintainer, which may now have difficulties reproducing the error or obtaining sufficient information to locate and correct it. Additionally the high costs of hardware component recalls and replacements are well known in other industries. Similarly cost explosions can be expected in a component software industry despite the benefit of electronic recall and delivery.

4. Parameterised Contracts

A component rarely fits directly into a new reuse context. For a component developer it is hard to foresee all possible reuse contexts. Hence, it is also hard for a developer to provide components with reasonable configuration options to 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, because of the following common 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. In the example, the `ViewerSoftwareServer` might fail or even be absent, but the `Controler` could still present standard text emails, although perhaps not display certain attachments.

2. a weaker postcondition of a component is sufficient in a specific reuse context. For example, the component user might not require all functions. Hence the component will itself require less functionality vis its requires interfaces and hence weaken its component precondition.

To model this we need some sort of adaptive pre- and postconditions. We call these *parameterised contracts* [7, 6]. In case 1 a parameterised contract computes the postcondition dependent upon 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 dependent upon the postcondition (which acts as a parameter of the precondition). For components this means, that provides- and requires-interfaces are not fixed but are computed to some extent taking into account the reuse context. Hence, in contrast to classical contracts, one can say:

> Parameterised contracts link the provides- and requires interface(s) of the same component (see fig. 1). They range over many possible actual contracts (i.e., ultimately 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, the parameterised contract tries to compute a new provides interface.

5. Applications of Parameterised Contracts

Like classical contracts, parameterised contracts depend on the actual interface model and should be statically computable. In any case, the software developers do not have to foresee possible reuse contexts but has to provide a bidirectional mapping between provides- and requires-interfaces. For simple interface lists (signatures a la CORBA IDL say), this means, that for each provided service, a list of required external services must be provided by the component developer. When computing the actual provides interface, a service would only be included, if all its required services are provided by the component's environment. If interfaces also describe component protocols, one has to specify a mapping from the provides interface to the requires interface protocol which also identifies the order in which reuquires services are invoked. We have developed tools for such models [6].

For extra-functional properties, the application of parameterised contracts is crucial. For example, one cannot specify the timing behaviour of a software component by some fixed figure. For example the worst-case time of a real-time component is always some function of the time it takes to perform critical system services that are only provided in the deployment environment. The same argument holds for reliability as empirically validated in our recent paper [8].

By connecting parameterised contracts of single components within component architectures and considering critical properties of the deployment environment, one can now compute the overall architectural properties. Our methods are described in [6, 10] in more detail. They necessitate parameterised contracts and are currently limited to non-cyclic architectures (i.e. layers of abstract machines). In our example, we can compute the timing of the composed component `MobileMailViewer` by computing the timing the `Controler` can provide in dependency of the timing `MailServer` and `ViewerSoftwareServer` can provide.

6. Conclusion

This paper discussed contractual usage of software component. We present requires interfaces as precondition of components and provides interfaces as postconditions. Parameterised contracts then link provides and requires interfaces of the same component. They are motivated by the necessity of computing functional and extra-functional component properties dependent upon deployment context. Our methods are supported by an existing tool and discussed using a running example.

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] 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.

[4] 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.

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

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

[7] R. H. Reussner. The use of parameterised contracts for architecting systems with software components. In W. Weck, J. Bosch, and C. Szyperski, editors, *Proceedings of the Sixth International Workshop on Component-Oriented Programming (WCOP'01)*, June 2001.

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

[9] H. W. Schmidt. Compatibility of interoperable objects. In B. Krämer, M. P. Papazoglou, and H. W. Schnmidt, editors, *Information Systems Interoperability*, pages 143–181. Research Studies Press, Taunton, England, 1998.

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

[11] 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.

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

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