Contract Specifications of Software Components

Andreas Sjögren
Department of Computer Engineering
Mälardalen University
andreas.sjogren@mdh.se

28 March 2002

Abstract

Before Component-Based Development, the practice of building applications from pre-existing parts can be trusted in mission critical applications we have to be able to determine how it will behave. The key to be able to determine the behavior when we do not have access to the source code is the specification of the software component. This paper discusses different levels of specifications, with particular focus on what we call contracts.

1 Introduction

Ideally components are black boxes which provides the implementation of a set of named interfaces. A specification of a component is therefore a specification of its interfaces. The specification should provide all the information a client needs to know about the component's services and context dependencies [3].

The properties which we can define for a component can be divided into four different levels of increasingly negotiable properties [1, 3]. At the first level we have syntactic specifications, which are the minimum required to be able to use a component. At the second level we have possibilities to describe sequential behavioral properties between components. At the third level we can describe the synchronization of services. At the fourth level we have possibilites to specify quality of service. In different stages of a system's development process we have different usage and need of specifications.

The reminder of this paper is organized as follows. The next section discusses four different levels of precision of specifications of software components. Section 3 discusses different aspect of the usage of specifications. Section 4 introduces a specification technique based on UML. Section 5 discusses future work and finally, Section 6 summarizes the paper.

2 The Specification Levels

The properties which we can define for a component can be divided into four different levels of increasingly negotiable properties [1, 3]. We will examine these levels in this section.

Sometimes the word *contract* is used to describe a component however we specify them, but most often, as in this paper, the word contract is restricted to mean interfaces specified at the second or higher levels.

2.1 Syntactical Specification

At the first level we have syntactic specifications, the basic way of specifying a component which are the minimum required to be able to use a component. Here we can use Interface Definition Languages (IDL:s), or other typed programming languages. The specifications used in practice today are mostly limited to this level. Different IDL:s are used in component technologies such as Microsoft's COM/DCOM and in Object Management Group's CORBA. In Sun's Enterprise JavaBeans (EJB) and JavaBeans (JB) the ordinary Java programming language are used to define interfaces. In these techniques we can specify the following

- the services (operations) a component provides

- possible input and output parameters for the services

- possible exceptions (or errorcodes as in COM) that might be raised during the execution of the services

The information we can obtain from a syntactic specification is limited to what services the component offers, and the number and types of its parameters. Below is an example of a interface specification of an audioplayer as a EJB component, the audioplayer has two interfaces one for the player functionality and another for the track list (in EJB we are only able to define one interface for the services of a bean, therefore we have to define a super interface which derives from the two interfaces).

```
public interface IAudioplayer
  extends javax.ejb.EJBObject
  ,IPlayer
  ,IPlayList {
}

interface IPlayer {
  void play() throws RemoteException;
  void stop() throws RemoteException;
  void fwd() throws RemoteException;
  void prev() throws RemoteException;
}

interface IPlayList {
  void addTrack(Track t)
    throws RemoteException;
  void delTrack(int id)
    throws RemoteException;
  void moveTrack(int id)
    throws RemoteException;
  void selectTrack(int id)
    throws RemoteException;
}
```

The primary use of these specifications is for static or dynamic type checking that a client uses a component properly, and the limit for enable interoperability between independent components. Of course there is no guarantee that a service offered by a component behaves as expected from the name of it.

2.2 Behavioral Specification

Since syntactic specification does not says anything about the outcome of execution of one of its provided services there is no way to be protected against stupid implementations, such as letting the stop function in an audioplayer component instead change to the next track in the playlist. On the second identified level of specification we can extend the syntactical interface specifications with some semantic information.

Inspired by ideas from axiomatic program theory (see for instance [4]) Bertrand Meyer developed the design by contract theory as a part of the Eiffel language [5]. Here a contract specification can be associated with a software element. If we use these contracts we can specify a component's behaviour by use of boolean assertions, called pre- and post-conditions on the component's services, and constraints on component's state, called invariants. The pre-conditions specify assertions that a client must fulfil before the invocation of a service. The post-conditions are assertions a component guarantee will hold after a service has been invoked, provided the pre-condition was fulfilled. An operation will often depend on the state of the component. Therefore the contract has to include a partial model of the state which may be affected of the operation. Furthermore, a set of invariants may be associated with the specification. An invariant is a constraint on the state model which always will hold. In some techniques there is also possible to specify intra-interface dependencies.

On this level of specifications we can specify conditions that could go wrong and assign explicitly responsibilities to the client, or the component (the contractor). If a client respects the contract it can only call a service from a component in a state where the pre-condition and the invariants holds. Likewise, with respect to the contract, a component guarantees a service will perform the work specified in the post-condition, and in a state where the invariants holds.

With use of the iContract [6] extension tool for Java we can add contracts to the syntactical specification of the audioplayer component. The syntax for the specifications are a combination of a own specification language which is a subset of OCL and ordinary Java syntax. The semantic information is added as Javadoc comments. Here is an example of

pre- and post-conditions for the addTrack method.

```
/**
 *@pre  t != null
 *@post playlist.size()
        = playlist.size()@pre + 1;
 *@post playlist.includes(t);
*/
void addTrack(Track t)
  throws RemoteException;
```

The pre-condition states that addTrack has to be invoked with an ingoing parameter that is not null. The post-condition ensures that, if the pre-condition holds, the track will be added to the playlist and the size of the playlist will be increased with one track after the invocation. We can refer to the state of the pre-condition in the post-condition by using the @pre postfix.

In some specification techniques it is also possible to specify that components are dependent on interfaces implemented by other components. These interface dependencies are often specified, as the required and offered interfaces of the component, respectively.

2.3 Synchronizational Specification

At the third identified specification level we are able to describe dependencies between services provided by a component in terms of synchronization between method calls. When specifying operations for distributed component systems transactional behavior becames very important. Most component technologies (such as COM/DCOM, CORBA, EJB) provides rich schemes for the definition of transactional behavior. Therefore a specification needs to describe a mapping onto these technologies.

The specification issues to cover here is whether a service starts needs its own transaction or if it runs in an excisting transaction. This is a binary choice to take care of, so in the specification we only need a mechanism for call out the forced new transaction for those operations, and absence of this mechanism implies that the operation runs in an exising transaction. Java provides this functionality through the synchronization keyword. E.g. if we always needs to know that, whatever service other clients request, the addTrack method will execute behave correctly we can define this method to be mutual exclusive.

```
synchronized void addTrack(Track t)
  throws RemoteException;
```

2.4 Quality-of-Service Specification

At the fourth and final level we have possibilities to specify quality–of-service, such as maximum response time, precision of results, etc. How to specify these kind of extra-functional properties and environmental constraints is largely an open issue, and has recently started to be discussed in the software engineering comunity and it will likely impact the future of software component specification. It is of course of great concern in system with high requirements on reliability, such as embedded systems and realtime systems.

When we specify the quality of service we can statically specify the features the component will respect. Alternatively, we can have a more dynamic solution where a server and a client can negotiate about the features. Examples of quality-of-service attributes are

- maxium response time
- average response time
- precision of the result (the bounds)
- memory usage

3 Usage of Contracts

A contract is a formal agreement between two or more parties. It specifies the detail of the agreement in a non-ambiguous form. This involves stating the responsibilities of each party. Contracts do not have to be complete. Specifications are of nature nondeterministic. If we over-specify and make them deterministic we have put to strict bounds of how to do things.

We can distinguish between two different needs of contracts for software components [2]:

Usage the contract between a component's interface and the clients.

Realization the contract between a component specification and its implementation.

The people which use components are often not the same as who use them. This is the fact, even if we are far from independent third component markets and the most components which are used today are developed internally in an organization. Not all aspects of a specification is relevant to a client.

A usage contract describes the relationship between a component's interfaces and a client. This is specified by a specification of the operations and eventual semantic information, depending on which level of specification technique we using. A realization contract is a contract between a component specification and an implementation of the component. While an interface specification defines a set of behavior, a component specification defines an implementation and deployment boundary.

3.1 Monitoring Contracts

What happens when a system violates the contract is a question not directly related to the specification problem. This depends of course of the implementation. However, many techniques like iContract [?] and Eiffel [5] provides a whole environment for both possibilities of specifying contracts and monitoring them. But, on the other hand is these techniques more focused on providing ways of implementing contracts and not on specification of them. In iContract we can add an exception to the assertions which will be raised if the system violates the contract. Here is an example for the `addTrack` method of the audioplayer component.

```
/**
 *@post playlist.includes(t) #TrackException;
*/
void addTrack(Track t)
  throws RemoteException;
```

Then we need to monitor du execution and catch eventual raised exceptions.

4 Specifications in UML

In this section we look closer into one particular specification technique. This technique is based on UML and OCL, with some extensions. Here we can specify a contract on level two for the audioplayer component. The operation signatures will look more or less the same as in Java specification on level 1, see Figure 1. We have associated a partial model of the components state to the component. The `<<interface type>>` stereotype is used to state that this is an interface type, interface type will be realized by an interface. Interface types can have a model of there state associated to them. We use the convention to prefix interface types with `IT`, and interfaces with an `I`. The audioplayer has two interfaces, which both is a realization of a corrsponding interface type. Both these interface types is associated to a number of tracks. We associates pre- and post-conditions, defined in OCL to each of the provided services. Here is an example of the pre- and post-conditions for the `addTrack` method of the `ITPlayer` interfacetype. The syntax for this example looks very similar to the iContract specification in Section 2, and has the same semantics.

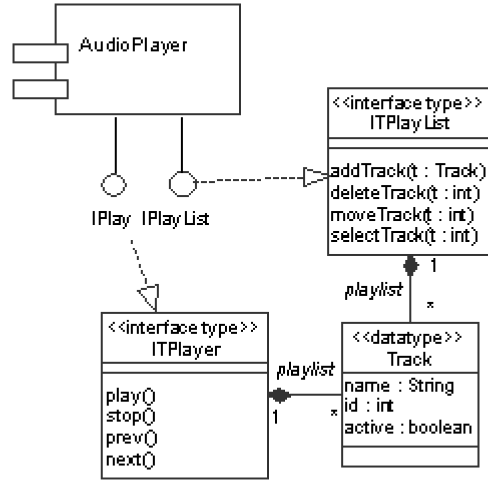

Figure 1: An UML specification of the Audioplayer component

```
context ITPlay::addTrack(in t:Track)
pre:
  t <> null
post:
  (playlist-playlist@pre)->size = 1
  and playlist.includes(t)
```

We can also add an inter-interface constraint, specifying that the set of tracks in the state model of the two interface must be the same.

```
context AudioPlayer
IPlay::playlist=IPlayList::playlist
```

In this technique we can also adds stereotypes which states that operations has to be executed in a own transaction and also add information about which interface a component uses and offers [2].

5 Future Work

As discussed above current component specification techniques are mostly focused on syntactic specifications. We will therefore investigate how we can use syntactic specifications which we have extended with some semantic information. Eventually, we will also specify extra-functional requirements in these specifications (this is a open issue, which has not been much research on so far). We will use well-known, techniques as UML to specify semantics (and maybe extra-functional properties) for components. By the use of standard techniques, and by providing tool support for the use of them, we hope to contribute to bridge the gap between formal specifications and practical implementations of these.

The modelling method presented in UML Components [2], and briefly discussed in Section 4, uses UML and OCL to write syntactic specifications, extended with semantic information in form of contracts. Currently we investigate how to use a variant of this method to specify components in an existing UML tool, and then generate stubs for the implementation of components. We will then experiment with specifications of components, and test how to specify both in the design of new components and to specify contracts for existing components. Contract can for example be added to existing components by use of wrappers, i.e. an “external contract”. We will also perform analysis of examples of components with and without contracts (with internal or external contracts) to see how much this extra code will affect different properties (e.g. performance, time and space complexity, etc.).

6 Summary

We have in this paper discussed the for Component-Based Development very important concept of specification of software components, in particular to be able to use a CBD strategy in systems with high requirements, e.g. on reliability. We discussed four identified levels of specification and showed some examples of different specifications. We discussed how we can have different use of the specifications in different stages. We shortly introduced a particular specification technique based on UML, and finally discussed some further work with this method.

References

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

[2] J. Chessman and J. Daniels. *UML Components - A simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[3] Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Systems*. Artech House, June 2002.

[4] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, Vol. 12:576–580, 1969.

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

[6] R. Kramer R. icontract - the java design by contract tool. In *Proceedings Technology of Object-Oriented Languages*, number TOOLS26. IEEE Computer Society, 1998.