Supporting Behavioral Contracts for COM Components

Sonal Bhagat, Rushikesh K. Joshi
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai - 400 076, India.
Contact Author Email: rkj@cse.iitb.ernet.in

Abstract

Specifying behavioral specifications for components apart from the conventional syntactic interface specifications can avoid many problems during component composition. Preconditions and postconditions describe one form of behavioral aspect of components. We discuss a tool and an implementation mechanism to incorporate behavioral contracts expressed in terms of preconditions and postconditions for COM components. A method invocation on a component is executed only if the precondition is satisfied. Similarly, the results are successfully returned upon successful execution of the postcondition. A design criterion was to facilitate contract specifications for existing components with least amount of changes at client and server side code. The tool requires that the component should provide an extra interface called *IAccess* if the behavioral contract needs component state. No modification is required to existing clients of the component.

1 Introduction

Though contracts play an important role in component composition, the conventional syntactic contracts specifying name, constituents and type signatures of component's interfaces are not sufficient for safe composition of components. The conventional contracts do not specify the behavioral aspects such as constraints on interface parameters and the semantics of interfaces. The behavioral aspects are important for selecting the right component for reuse and achieving correct composition of components. Hence the notion of behavioral contracts [1], [3] is increasingly becoming important. Behavioral contracts in *Design by contract* method [2] include boolean invariants, preconditions and postconditions.

In this paper, we describe a method and a tool for integrating behavioral contracts in the form of preconditions and postconditions to existing COM components. One goal has been to allow contract specifications to existing components with as little modifications to existing code as possible. The behavioral contract itself is specified in a *contract component*. A method invocation on a component is invoked only if the precondition is satisfied in its contract component. Similarly the contract component ensures that the postcondition is satisfied after the execution of the method. Upon failure, an exception is returned to the client. The client remains unaware of the existence of the intermediate contract component. This approach is similar to filter objects [4], in which, every method invocation is trapped transparently by the intermediate filter object and computations can be carried out before and after the method invocation without the knowledge of the client.

By specifying behavioral contracts in the form of intermediate contract components, not only preconditions and postconditions can be verified, but other intermediate tasks also can be performed. A contract component may require access to implementation of the associated server component. In this case, the component has to implement an interface called *IAccess* through which the contract component can gain access to the component's implementation. Access to the component through *IAccess* is restricted and only the contract component is allowed to invoke this interface.

A tool has been built to support development of the contract components. The tool generates the necessary abstractions and partial implementations for contract specification for a given component. We will describe the design and implementation of this method in the subsequent sections.

2 The Design

The tool requires the IDL specification of the server component and the desired name of the contract component. The IDL specification of the server component is typically specified in a *.idl* file containing the description of the interfaces that are implemented by the server component. The tool parses the *.idl* file to generate required abstractions and partial implementations. Let us consider an input *.idl* containing the interface declaration as shown in Figure 1.

As required by the contract component, the server component has implemented the *IAccess* interface. The *IAccess* interface contains methods for reading states of the implementation members of the server component, which are required for testing the preconditions and the postconditions. Access to interface *IAccess* is provided only to the contract component. This is ensured through one of the input parameters to all methods of *IAccess* interface, which is the CLSID of the contract component. By comparing the CLSIDs, implementation of *IAccess* may reject an invocation. Figure 2 depicts the class diagram of the system of components after it is made contract aware.

As shown in Figure 2, *CServer* implements interfaces *IAccess* and *ISampleInterface*. The tool generates the classes *ContractSuperClass*, *CContract* and *CContractFactory*. Class *CContract* is the contract component class and class *CContractFactory* creates the contract component.

```
// Interface ISampleInterface
[
object,
        uuid(8D70DF12-6F45-4418-A083-F4ED3CDB9D48),
helpstring("ISampleInterface Interface"),
pointer_default(unique)
]
interface  ISampleInterface : IUnknown
{
    HRESULT function([in] int a,[in] int b, [out] int* c);

} ;

// Interface IAccess
[
object,
uuid(0416229D-147D-4065-B5B0-BDB2B0BDEE59),
helpstring("IAccess Interface"),
pointer_default(unique)
]
interface IAccess : IUnknown
{
HRESULT getPrivateMember([in] CLSID clsid,[out] int* av);
} ;
```

Figure 1: The Component IDL

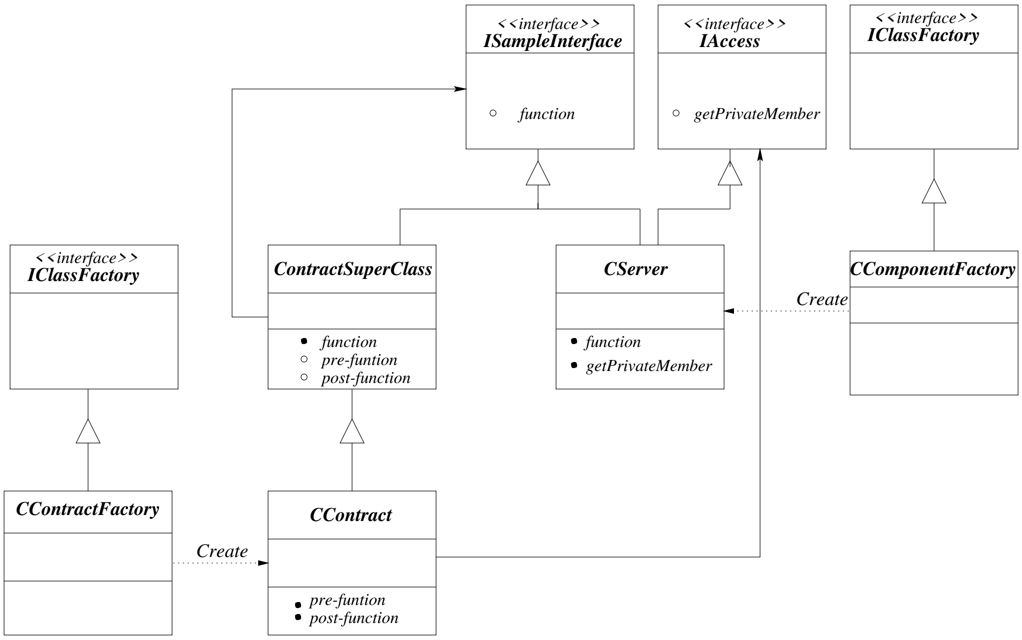

Figure 2: The Class Diagram

Class *ContractSuperClass* implements the interfaces implemented by *CServer* making the classes conceptually compatible. This allows the component class to be transparently filtered through the same abstractions provided by the contract component. The contract programmer only needs to implement the abstract methods in this class through inheritance. This class implements the server component's interface in such a way that preconditions and postconditions are invoked before and after the intended function invocation on the component. Preconditions and postconditions are modeled as abstract methods (hooks) in this class. This is achieved through COM's support to change implementations keeping the same abstraction through a *treat as* directive.

An example contract class implementation of a method in the component interface is given in Figure 3. It can be seen that if the precondition is not satisfied, an error code *CContract_E_PRECONDITION* is returned to the client without passing on the invocation to the server component. An existing client will be able to handle this error code in a generic fashion.

The precondition method receives the input parameters as arguments. It also has an access to the *IAccess* interface on the server component. A pointer to this interface is made available through a tool generated implementation. The *IAcess* interface can be used by the precondition and postcondition implementation if required by their semantics. Similarly, the postcondition method receives the output parameters. If the postcondition is not satisfied, an error code *CContract_E_POSTCONDITION* is returned to the caller. If an error code is returned by the server component itself, it is passed on to client. This special case has been omitted from

```
HRESULT __stdcall ContractSuperClass::function(int a,int b,int* c) {
   if (pre_function(a,b)) {
        int contract_c;
        pISampleInterface->function(a,b,&contract_c);
        if (post_function(&contract_c)) {
                *c = contract_c;
                return S_OK;
        }else
                return CContract_E_POSTCONDITION;
   }else
        return CContract_E_PRECONDITION;
}
```

Figure 3: Contract Implementation of Component's Interface

Figure 3 for the sake of readability.

The concrete contract component specifies the implementations of postcondition and precondition. As shown in the class diagram, class *CContract* implements the contract. The name of the contract component is chosen by the user through an interface provided by the tool. The skeleton of the contract class is generated by the tool and the contract developer only needs to complete the implementations of the hook methods of the superclass. The tool also generates a factory class for the contract component, in this example, class *CContractFactory* as shown in the class diagram.

We can view the relationship between the contract component and the server component from the point of view of the user, through *COM containment* as shown in Figure 4. It can however be noted that the class diagram depicts the relationship as *using* rather than aggregation. This is because although the server component and the contract component may be located on different machines and may be aggregated in their respective servers, for the outside world, the server component is not directly accessible. All requests to the server components are routed to the contract component.

A CoCreateInstance for the server component actually invokes a CoCreateInstance on the contract component which in turn directs the component's server process to create an instance of the server component. In this way, from the user's viewpoint, the relationship can be described through COM's containment model as shown in Figure 4. However, from the system's viewpoint, it is an example of *using* relationship since it is possible to remove or change a contract component at any point of time. The implementation mechanism is discussed in the next section.

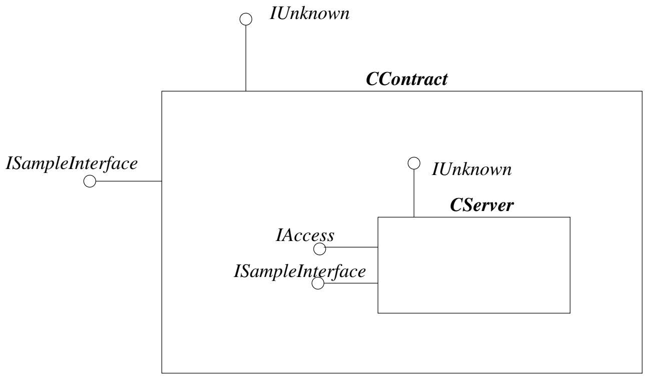

Figure 4: Containment of Server Component by Contract Component

3 Implementation Mechanism

The sequence of events that leads to the installation of a contract component is shown in the interaction diagram in Figure 5. The mechanism uses out-of-process servers. When the server process is brought up, it creates and registers an instance of the component's class factory.

Whereas, when the contract server process is started, it calls the *CoTreatAsClass(CLSID_S,CLSID_C)* API to switch the implementation of the server component. *CLSID_S* represents the CLSID of the server component and *CLSID_C* represents the CLSID of the contract component. This API call assigns the *TreatAs* key of the server component to *CLSID_C*. This causes the contract component class to emulate the server component class. Subsequently, calls to *CoGetClassObject* with *CLSID_S* as the parameter transparently use *CLSID_C*.

As show in the interaction diagram creating an object of *CLSID_S* results in *CreateInstance* being invoked on *CContractFactory*. The contract factory now needs to perform two tasks: the server component should be created and returned to the user as desired by the original call, and a contract component should be created and assigned in relationship with the server component. At this stage, the contract component cannot directly instantiate the server component since all creation calls on the component are redirected to itself. Hence momentarily, it disables the redirection by a call to *CoTreatAsClass(CLSID_S, CLSID_NULL)* which removes the redirection. *CContractFactory* instance then calls *CoCreateInstance* on the server component factory. After remembering the pointer to the server component, it switches back to the redirection and returns the server component's interface to the client. From this point onwards, the contract component filters the message invocations sent to the server component.

This design can be extended to include an elaborate contract plug-unplug protocol to remove or change contract components dynamically. The method does not require the client code to be modified. An existing server needs to provide an additional interface if the contract

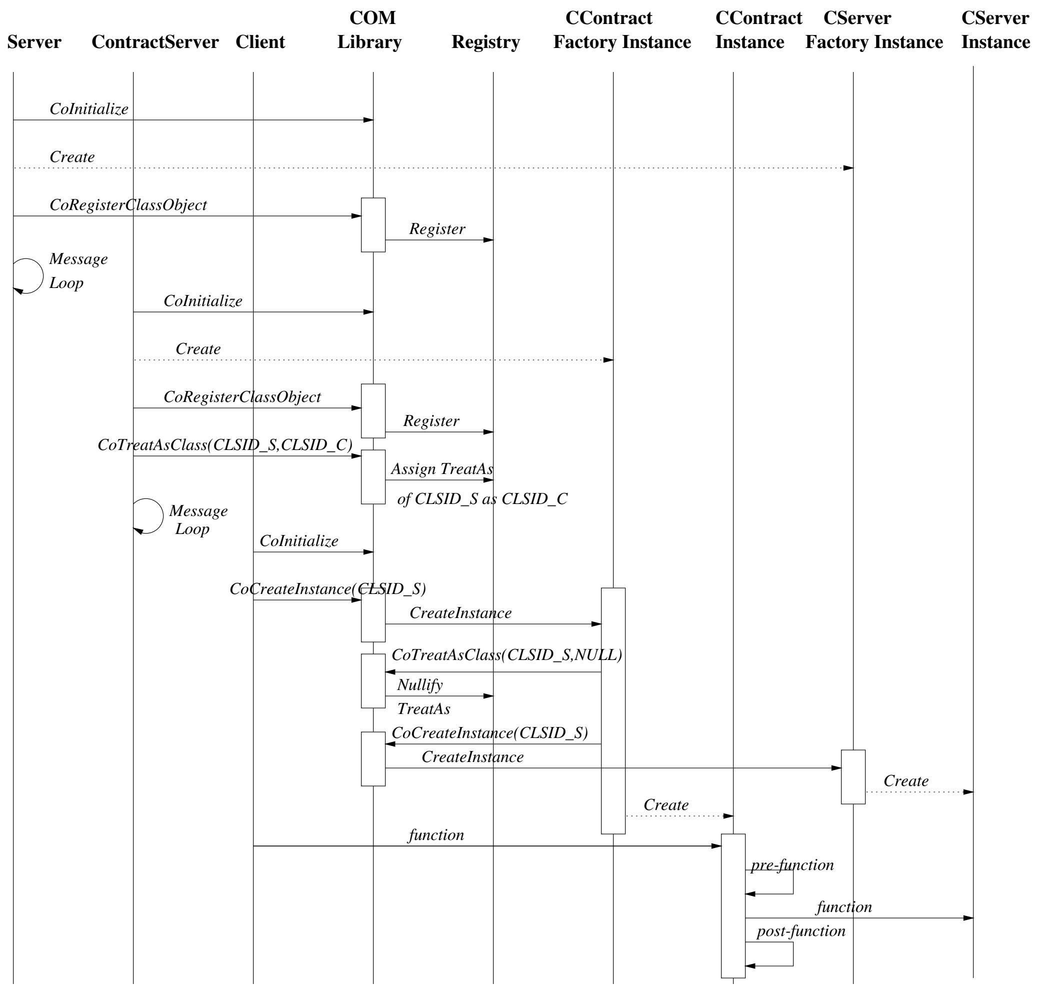

Figure 5: Interaction diagram

component requires the state of the component.

4 Conclusion

A method of specifying behavioral contracts for existing COM components was discussed. The method considers contract specification in terms of preconditions and postconditions for existing COM components. A tool support for the development of contract components was discussed. The implementation is built around COM's *treat as* directives. The contract component remains transparent to the client of the server component.

References

[1] Antoine Beugnard, Jean Mark, Noel Plouzeau, Damien Watkins, Making Components Contract Aware, IEEE Computer, 32(7), July 1999.

[2] Bertrand Meyer, Design by Contract: The Lessons of Ariane, IEEE Computer, 30(2), Jan 1997

[3] Cynthia Della, Shmuel Rotenstreich, Behavioral Specification of Distributed Software Component Interfaces, IEEE Computer, 32(7), 1999.

[4] R. K. Joshi, Modeling with Filter Objects in Distributed Systems, Proceedings of the 2nd Workshop on Engineering Distributed Objects (EDO 2000), LNCS Vol. 1999, pp. 176-181.