Color profile: Disabled
Composite Default screen

2

Specification of Software Components

Frank Lüders, Kung-Kiu Lau, and Shui-Ming Ho

Introduction

In its simplest form a software component contains some code (that can be executed on certain platforms) and an interface that provides (the only) access to the component. The code represents the operations that the component will perform when invoked. The interface tells the component user everything he needs to know in order to deploy the component. Components can of course be deployed in many different contexts. Ideally, components should be black boxes, to enable users to (re)use them without needing to know the details of their inner structure. In other words, the interface of a component should provide all of the information needed by its users. Moreover, this information should be the only information they need. Consequently, the interface of a component should be the only point of access to the component. Therefore, it should contain all of the information that users need to know about the component's operations (that is, what its code enables it to do) and its context dependencies (that is, how and where the component can be deployed). The code, on the other hand, should be completely inaccessible (and invisible) if a component is to be used as a black box.

The specification of a component is therefore the specification of its interface. This must consist of a precise definition of the component's

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:28 2002

Color profile: Disabled
Composite Default screen

operations and context dependencies and nothing else. Typically, the operations and context dependencies will contain the parameters of the component.

The specification of a component is useful to both component users and component developers. For users, the specification provides a definition of its interface, namely, its operations and context dependencies. Because it is only the interface that is visible to users, its specification must be precise and complete. For developers, the specification of a component also provides an abstract definition of its internal structure. Although this should be invisible to users, it is useful to developers (and maintainers), not the least as documentation of the component.

In this chapter, we discuss the specification of software components. We identify all features that should be present in an idealized component, indicate how they should be specified, and show how they are specified using current component specification techniques.

Current Component Specification Techniques

The specifications of components used in practical software development today are limited primarily to what we will call syntactic specifications. This form of specification includes the specifications used with technologies such as COM [1], the Object Management Group's Common Object Request Broker Architecture (CORBA) [2], and Sun's JavaBeans [3]. The first two of these use different dialects of the IDL, whereas the third uses the Java programming language to specify component interfaces. In this section, COM is mainly used to illustrate the concepts of syntactic specification of software components.

First, we take a closer look at the relationships between components and interfaces. A component provides the implementation of a set of named interfaces, or types, each interface being a set of named operations. Each operation has zero or more input and output parameters and a syntactic specification associates a type with each of these. Many notations also permit a return value to be associated with each operation, but for simplicity we do not distinguish between return values and output parameters. In some specification techniques it is also possible to specify that a component requires some interfaces, which must be implemented by other components. The interfaces provided and required by a component are often called the incoming and outgoing interfaces of the component, respectively.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:28 2002

Color profile: Disabled
Composite Default screen

Figure 2.1 is a UML class diagram [4] showing the concepts discussed above and the relationships between them. Note that instances of the classes shown on the diagram will be entities such as components and interfaces, which can themselves be instantiated. The model is therefore a UML metamodel, which can be instantiated to produce other models. It is worth noting that this model allows an interface to be implemented by several different components, and an operation to be part of several different interfaces. This independence of interfaces from the component implementations is an essential feature of most component specification techniques. The possibility of an operation being part of several interfaces is necessary to allow inheritance, or subtyping, between interfaces. The model also allows parameters to be simultaneously input and output parameters.

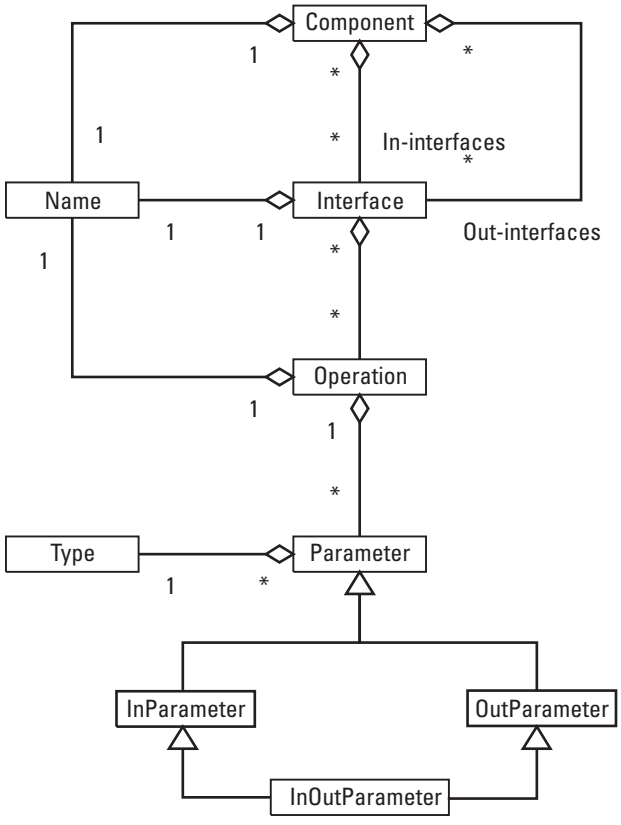

Figure 2.1 UML metamodel of the concepts used in the syntactic specification of software components.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:32 2002

Color profile: Disabled
Composite Default screen

The model presented in Figure 2.1 is intended to be a generic representation of the relationships between components, interfaces, and operations. In practice, these relationships vary between specification techniques. For example, one can distinguish between object-oriented specifications and what might be called procedural specifications. In this chapter we consider only object-oriented specifications that are used by current technologies. This leads to no loss of generality, because procedural specification can be seen as a special case of object-oriented specification. Subtle differences are seen in the precise nature of the relationship between a component and its interfaces in different object-oriented specification techniques. In COM, for example, a component implements a set of classes, each of which implements a set of interfaces. The statement that a component implements a set of interfaces thus holds by association. In more traditional object-oriented specification techniques, a component is itself a class that has exactly one interface. The statement that a component implements a set of interfaces still holds, because this interface can include, or be a subtype of, several other interfaces.

As an example of a syntactic specification, we now consider the specification of a COM component. Below we provide a slight simplification of what might be the contents of an IDL file. First, two interfaces are specified, including a total of three operations that provide the functionality of a simple spell checker. Both interfaces inherit from the standard COM interface `IUnknown`. (All COM interfaces except `IUnknown` must inherit directly or indirectly from `IUnknown`. See [1] for more information about the particulars of COM.) All operations return a value of type `HRESULT`, which is commonly used in COM to indicate success or failure. A component is then specified (called a *library* in COM specifications), thus implementing one COM class, which in turn implements the two interfaces previously specified. This component has no outgoing interfaces.

```
interface ISpellCheck : IUnknown
{
  HRESULT check([in] BSTR *word, [out] bool *correct);
};

interface ICustomSpellCheck : IUnknown
{
  HRESULT add([in] BSTR *word);
  HRESULT remove([in] BSTR *word);
};

library SpellCheckerLib
{
```

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:33 2002

Color profile: Disabled
Composite Default screen


```
    coclass SpellChecker
    {
        [default] interface ISpellCheck;
        interface ICustomSpellCheck;
    };
};
```

Relating this specification to the model above, there is one instance of Component, which is associated with two instances of Interface. Taking a closer look at the first interface, it is associated with a single instance of Operation, which is itself associated with one instance of InParameter and two instances of OutParameter, representing the two named parameters and the return value. The information that can be obtained from a component specification such as the above is limited to what operations the component provides and the number and types of their parameters. In particular, there is no information about the effect of invoking the operations, except for what might be guessed from the names of operations and parameters. Thus, the primary uses of such specifications are type checking of client code and as a base for interoperability between independently developed components and applications. Different component technologies have different ways of ensuring such interoperability. For example, COM specifies the binary format of interfaces, whereas CORBA defines a mapping from IDL to a number of programming languages.

An important aspect of interface specifications is how they relate to substitution and evolution of components. Evolution can be seen as a special case of substitution in which a newer version of a component is substituted for an older version. Substituting a component Y for a component X is said to be safe if all systems that work with X will also work with Y. From a syntactic viewpoint, a component can safely be replaced if the new component implements at least the same interfaces as the older components, or, in traditional object-oriented terminology, if the interface of the new component is a subtype of the interface of the old component. For substitution to be safe, however, there are also constraints on the way that the semantics of operations can be changed, as we will see in the next section.

Specifying the Semantics of Components

Although syntactic specifications of components are the only form of specifications in widespread use, it is widely acknowledged that semantic information about a component's operations is necessary to use the component

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:33 2002

Color profile: Disabled
Composite Default screen

effectively. Examples of such information are the combinations of parameter values an operation accepts, an operation's possible error codes, and constraints on the order in which operations are invoked. In fact, current component technologies assume that the user of a component is able to make use of such semantic information. For instance, COM dictates that the error codes produced by an operation are immutable; that is, changing these is equivalent to changing the interface. These technologies do not, however, support the specification of such information. In the example with COM, there is no way to include information about an operation's possible error codes in the specification.

Several techniques for designing component-based systems that include semantic specifications are provided in the literature. In this section, we examine the specification technique presented in [5], which uses UML and the Object Constraint Language (OCL [6]) to write component specifications. OCL is included in the UML specification. Another well-known method that uses the same notations is Catalysis [7]. The concepts used for specification of components in these techniques can be seen as an extension of the generic model of syntactic specification presented in the previous section. Thus, a component implements a set of interfaces, each of which consists of a set of operations. In addition, a set of preconditions and postconditions is associated with each operation. Preconditions are assertions that the component assumes to be fulfilled before an operation is invoked. Postconditions are assertions that the component guarantees will hold just after an operation has been invoked, provided the operation's preconditions were true when it was invoked. In this form of specification, nothing is said about what happens if an operation is invoked while any of its preconditions are not fulfilled. Note that the idea of pre- and postconditions is not a novel feature of component-based software development, and it is used in a variety of software development techniques, such as the Vienna Development Method [8] and Design by Contract [9].

Naturally, an operation's pre- and postconditions will often depend on the state maintained by the component. Therefore, the notion of an interface is extended to include a model of that part of a component's state that may affect or be affected by the operations in the interface. Now, a precondition is, in general, a predicate over the operation's input parameters and this state, while a postcondition is a predicate over both input and output parameters as well as the state just before the invocation and just after. Furthermore, a set of invariants may be associated with an interface. An invariant is a predicate over the interface's state model that will always hold. Finally, the component specification may include a set of inter-interface conditions, which are predicates over the state models of all of the component's interfaces.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:33 2002

Color profile: Disabled
Composite Default screen

The concepts introduced here and the relationships among them are shown on the UML class diagram of Figure 2.2. For the sake of readability, the classes Name, Type, and InOutParameter are not shown, because they have no direct relationships with the newly introduced classes. Note that this model allows the same state to be associated with several interfaces. Often, the state models of different interfaces of a component will overlap rather than be identical. This relationship cannot be expressed in the model because we cannot make any assumptions about the structure of state models. Note also how each postcondition is associated with both input and output parameters and two instances of the state model, representing the state before and after an invocation.

In the model presented in Figure 2.2, a partial model of the state of a component is associated with each interface, to allow the semantics of an

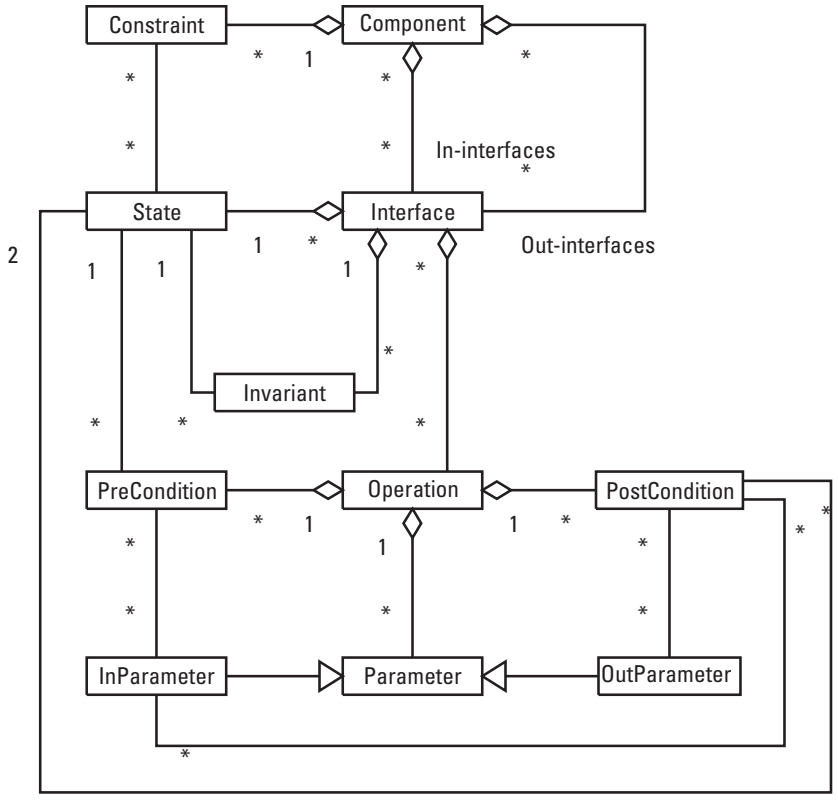

Figure 2.2 UML metamodel of the concepts used in semantic specification of software components.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:38 2002

Color profile: Disabled
Composite Default screen

interface's operations to be specified. The reader should note that this is not intended to specify how a state should be represented within the component. Although state models in component specifications should above all be kept simple, the actual representation used in the component's implementation will usually be subject to efficiency considerations, depending on the programming language and other factors. It is also worth mentioning that the model is valid for procedural as well as object-oriented specification techniques.

Before discussing the ramifications of this model any further, we now consider an example specification using the technique of [5]. Figure 2.3 is an example of an interface specification diagram. It shows the two interfaces introduced in the previous section as classes with the <<interface type>> stereotype. Thus, all information in the syntactic interface specifications is included here. The state models of the interfaces are also shown. A state model generally takes the form of one or more classes having at least one composition relationship with the interface to which the state belongs. The special stereotype <<interface type>> is used instead of the standard <<interface>> because the standard <<interface>> would not allow the state models to be associated with the interfaces in this way.

The interface specification diagram is only a part of the complete interface specifications. The pre- and postconditions that specify the semantics of the operations as well as any invariants on the state model are specified separately in OCL. Below is a specification of the three operations of the two interfaces above. There are no invariants on the state models in this example.

```
context ISpellCheck:: check(in word : String, out correct :
Boolean) @code = : HRESULT
pre:
```

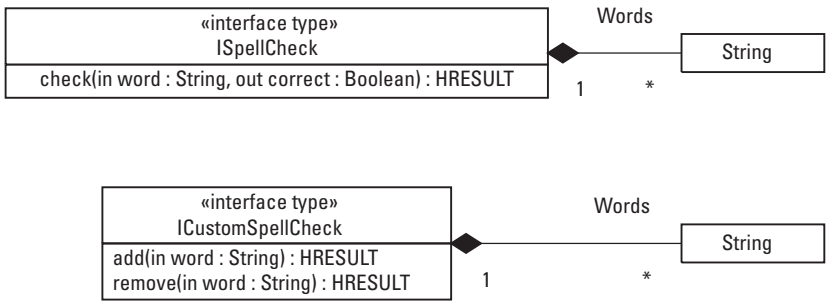

Figure 2.3 Example interface specification diagram.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:42 2002

Color profile: Disabled
Composite Default screen


```
   word <> ""
post:
   SUCCEEDED(result) implies correct = words->includes
(word)

context ICustomSpellCheck::add(in word : String) : HRESULT
pre:
   word <>
post:
   SUCCEEDED(result) implies  words = words@@pre->includ-
ing(word)

context ICustomSpellCheck::remove(in word : String) :
HRESULT
pre:
   word <>
post:
       SUCCEEDED(result) implies words = words@@pre->
exluding(word)
```

The precondition of the first operation states that if it is invoked with an input parameter that is not the empty string, the postcondition will hold when the operation returns. The postcondition states that if the return value indicates that the invocation was successful, then the value of the output parameter is true if word was a member of the set of words and false otherwise. The specifications of the two last operations illustrate how postconditions can refer to the state before the invocation using the @pre suffix. This specification technique uses the convention that if part of an interface's state is not mentioned in a postcondition, then that part of the state is unchanged by the operation. Thus, words = words@pre is an implicit postcondition of the first operation. All specifications refer to an output parameter called result, which represents the return value of the operations. The function `SUCCEEDED` is used in COM to determine whether a return value of type `HRESULT` indicates success or failure.

Like interface specification diagrams, component specification diagrams are used to specify which interfaces components provide and require. Figure 2.4 is an example of such a diagram, specifying a component that provides the two interfaces specified above. The component is represented by a

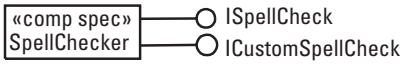

Figure 2.4 Example of a component specification diagram.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:46 2002

Color profile: Disabled
Composite Default screen

class with stereotype <<comp spec>> to emphasize that it represents a component specification. UML also has a standard component concept, which is commonly used to represent a file that contains the implementation of a set of concrete classes.

The component specification is completed by the specification of its inter-interface constraints. The component in this example has one such constraint, specifying that the sets of words in the state models of the two interfaces must be the same. This constraint relates the operations of the separate interfaces to each other, such that invocations of add or remove affect subsequent invocations of check. The constraint is formulated in OCL as follows:

```
context SpellChecker
ISpellCheck::words = ICustomSpellCheck::words
```

An important property of the model presented above is that state models and operation semantics are associated with interfaces rather than with a component. This means that the semantics is part of the interface specification. Consequently, a component cannot be said to implement an interface if it implements operations with the same signatures as the interface's operations but with different semantics. Note that the terminology varies in the literature on this point, because interfaces are sometimes seen as purely syntactic entities. In such cases, specifications that also include semantics are often called contracts. UML, for instance, defines an interface to be a class with only abstract operations and it can have no state associated with it.

Although the main uses of syntactic specifications are for type checking and ensuring interoperability, the utility of semantic specifications is potentially much larger. The most obvious use is perhaps for tool support for component developers as well as developers of component-based applications. For the benefit of component developers, one can imagine an automatic testing tool that verifies all operations produce the correct postconditions when their preconditions are satisfied. For this to work, the tool must be able to obtain information about a component's current state. A component could easily be equipped with special operations for this purpose that would not need to be included in the final release. Similarly, for application developers, one can imagine a tool that generates assertions for checking that an operation's preconditions are satisfied before the operation is invoked. These assertions could either query a component about its current state, if this is possible, or maintain a state model of their own. The last technique has a requirement, however, that other clients cannot affect the state maintained by a component,

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:47 2002

Color profile: Disabled
Composite Default screen

since the state model must be kept synchronized with the actual state. Such assertions would typically not be included in a final release either.

With the notion of an interface specification that include semantics, the concept of substitution introduced in the previous section can now be extended to cover semantics. Clearly, if a component Y implements all (semantically specified) interfaces implemented by another component X, then Y can be safely substituted for X. This condition is not necessary, however, for substitution to be safe. What is necessary is that a client that satisfies the preconditions specified for X must always satisfy the preconditions specified for Y, and that a client that can rely on the postconditions ensured by X can also be ensured it can rely on Y. This means that Y must implement operations with the same signatures as the operations of X, and with pre- and postconditions that ensure the condition above. More specifically, if X implements an operation O, where pre(O) is the conjunction of its preconditions and post(O) the conjunction of its postconditions, Y must implement an operation O′ with the same signature such that pre(O′) implies pre(O) and post(O) implies post(O′). In other words, the interfaces implemented by Y can have weaker preconditions and stronger postconditions than the interfaces implemented by X. It follows from this that the state models used for specifying the interfaces of X and Y need not be identical. This condition for semantically safe substitution of components is an application of Liskov's principle of substitution [10]. Note that the above discussion is only valid for sequential systems. For multithreaded components or components that are invoked by concurrently active clients, the concept of safe substitution must be extended as discussed in [11]. Finally, note that a client can still malfunction after a component substitution, even if the components fulfill semantic specifications that satisfy the above condition. This can happen, for instance, if the designers of the client and the new component have made conflicting assumptions about the overall architecture of the system. The term *architectural mismatch* has been coined to describe such situations [12].

The component specification diagram in Figure 2.4 shows how we can indicate which interfaces are offered by a component. In this example, we indicated that the spell checker offered the interfaces `ISpellCheck` and `ICustomSpellCheck` and used the constraint

```
ISpellCheck::words = ICustomSpellCheck::words
```

to specify that the interfaces act on the same information model. We could, however, extend such diagrams to indicate the interfaces on which a component depends. This is illustrated in Figure 2.5.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:47 2002

Color profile: Disabled
Composite Default screen

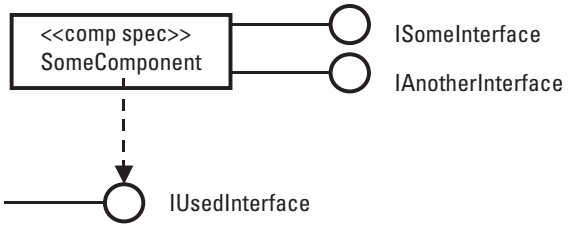

Figure 2.5 Component specification showing an interface dependency.

We can also specify realization contracts using collaboration interaction diagrams. For example, in Figure 2.6 we state that whenever the operation op1 is called, a component supporting this operation must invoke the operation op2 in some other component. Component specification diagrams and collaboration interaction diagrams may therefore be used to define behavioral dependencies.

Specifying Extrafunctional Properties of Components

The specification of extrafunctional properties of software components has recently become a subject of interest, mainly within the software architecture community. In [13], it is argued that the specification of architectural components is not properly addressed by conventional software doctrines. Architectural components are components of greater complexity than algorithms and data structures. Software components, as defined above, generally belong to this class. Conventional software doctrine is the view that software

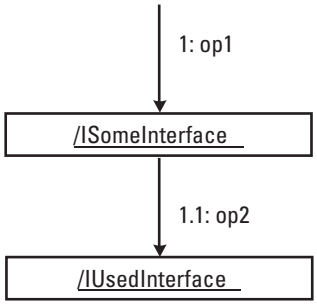

Figure 2.6 Collaboration interaction diagrams.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:55 2002

Color profile: Disabled
Composite Default screen

specifications must be sufficient and complete (that is, provide everything a user needs to know and may rely on in using the software), *static* (written once and frozen), and *homogeneous* (written in a single notation).

To use an architectural component successfully, information about more than just its functionality is required. This additional information includes structural properties, governing how a component can be composed with other components; extrafunctional properties, such as performance, capacity, and environmental assumptions; and family properties, specifying relationships among similar or related components. It is not realistic to expect specifications to be complete with respect to all such properties, due to the great effort this would require. [Nor is it realistic to expect that the developer of a component could anticipate all aspects of the component in which its user might be interested]. Because we cannot expect software components to be delivered with specifications that are sufficient and complete, and because developers are likely to discover new kinds of dependencies as they attempt to use independently developed components together, specifications should be extensible. Specifications should also be heterogeneous, since the diversity of properties that might be of interest is unlikely to be suitably captured by a single notation.

The concept of credentials is proposed in [13] as a basis for specifications that satisfy the requirements outlined above. A credential is a triple <Attribute, Value, Credibility>, where Attribute is a description of a property of a component, Value a measure of that property, and Credibility a description of how the measure has been obtained. A specification technique based on credentials must include a set of registered attributes, along with notations for specifying their value and credibility, and provisions for adding new attributes. A technique could specify some attributes as required and others as optional. The concept has been partially implemented in the architecture description language UniCon [14], which allows an extendable list of <Attribute, Value> pairs to be associated with a component. The self-describing components of Microsoft's new .NET platform [15] include a concept of attributes in which a component developer can associate attribute values with a component and define new attributes by subclassing an existing attribute class. Attributes are part of a component's metadata, which can be programmatically inspected, and is therefore suitable for use with automated development tools.

The concept of credentials has been incorporated in an approach to building systems from preexisting components; this approach is called *Ensemble* [16] and it focuses on the decisions that designers must make, in particular when faced with a choice between competing technologies,

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:55 2002

Color profile: Disabled
Composite Default screen

competing products within a technology, or competing components within a product. In Ensemble, a set of credentials may be associated with a single technology, product, or component, or with a group of such elements. In addition, a variation of credentials is introduced to handle measures of properties that are needed but have not yet been obtained. These are called postulates and can be described as credentials where the credibility is replaced by a plan for obtaining the measure. The credential triple is thus extended with a flag `isPostulate`.

Returning our focus to the specification of single components, we now extend the ideas of Ensemble to allow a set of credentials to be associated with a component, an interface, or an operation. A UML metamodel with the concepts of syntactic specification augmented with credentials is shown in Figure 2.7. The class Name and the subclasses of Parameter have been omitted for brevity. Note that the concept of credentials is complementary to the specification of a component's functionality and completely orthogonal to the concepts introduced for semantic specifications. Because the specification of extrafunctional properties of software components is still an open area

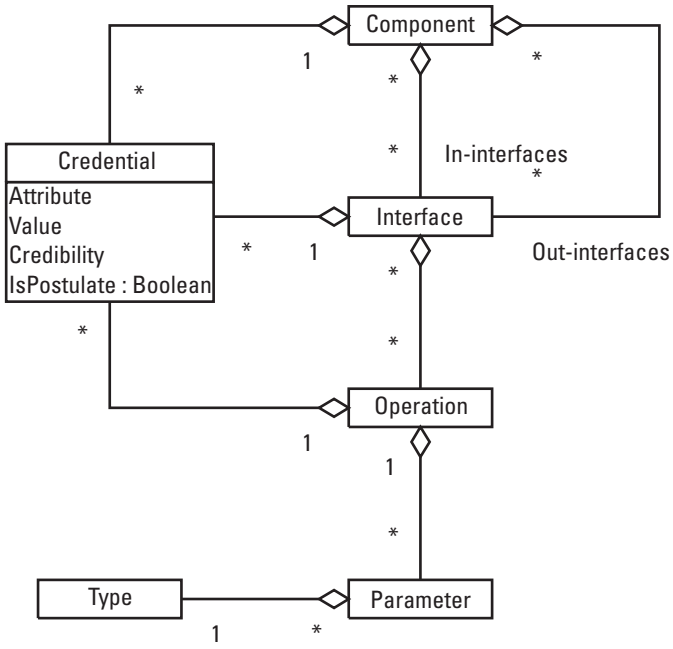

Figure 2.7 UML metamodel of concepts used to specify extrafunctional properties of software components.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:59 2002

Color profile: Disabled
Composite Default screen

of research, it would probably be premature to proclaim this to be a generic model.

Because the extrafunctional properties that may be included in a component specification can be of very different natures, we cannot formulate a general concept of safe substitution for components that includes changes of such properties. A set of extrafunctional properties, which can all be expressed as cost specifications, is studied in [17] where it is shown that, depending on the chosen property, weakening, strengthening, or equivalence is required for substitution to be safe

Summary

A component has two parts: an interface and some code. The interface is the only point of access to the component, and it should ideally contain all of the information that users need to know about the component's operations: what it does and how and where the component can be deployed, that is, its context dependencies. The code, on the other hand, should be completely inaccessible (and invisible). The specification of a component therefore must consist of a precise definition of the component's operations and context dependencies. In current practice, component specification techniques specify components only syntactically. The use of UML and OCL to specify components represents a step toward semantic specifications. Specification of extrafunctional properties of components is still an open area of research, and it is uncertain what impact it will have on the future of software component specification.

References

[1] Microsoft, "The Component Object Model Specification," Report v0.99, Microsoft Standards, Redmond, WA: Microsoft, 1996.

[2] OMG, "The Common Object Request Broker: Architecture and Specification," Report v2.4, OMG Standards Collection, OMG, 2000.

[3] Sun Microsystems, "JavaBeans 1.01 Specification," http://java.sun.com/beans.

[4] OMG, "OMG Unified Modeling Language Specification," Report v1.3, OMG, June 1999.

[5] Cheesman, J., and J. Daniels, *UML Components—A Simple Process for Specifying Component-Based Software*, Reading, MA: Addison-Wesley, 2000.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:59 2002

Color profile: Disabled
Composite Default screen

[6] Warmer, J., and A. Kleppe, *The Object Constraint Language*, Reading, MA: Addison-Wesley, 1999.

[7] D'Souza, D., and A. C. Wills, *Objects, Components and Frameworks: The Catalysis Approach*, Reading, MA: Addison-Wesley, 1998.

[8] Jones, C. B., *Systematic Software Development Using VDM*, Upper Saddle River, NJ: Prentice Hall, 1990.

[9] Meyer, B., *Object-Oriented Software Construction*, Upper Saddle River, NJ: Prentice Hall, 1997.

[10] Liskov, B., "Data Abstraction and Hierarchy," *Addendum to Proc. OOPSLA'87*, Orlando, FL, SIGPLAN Notices, 1987.

[11] Schmidt, H., and J. Chen, "Reasoning About Concurrent Objects," *Proc. Asia-Pacific Software Engineering Conf.* Brisbane, Australia, IEEE Computer Society, 1995.

[12] Garlan, D., R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software*, Vol. 12, No. 6, 1995, pp. 17–26.

[13] Shaw, M., "Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does," *Proc. 8th Int. Workshop Software Specification and Design*, Schoss Velen, Germany, IEEE Computer Society, 1996.

[14] Shaw, M., et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. on Software Engineering*, Vol. 21, No. 24, 1995, pp. 314–335.

[15] Conrad, J., et al., *Introducing .NET*, Wrox Press, 2000.

[16] Wallnau, K. C., and J. Stafford, "Ensembles: Abstractions for a New Class of Design Problem," *Proc. 27th Euromicro Conf.*, Warsaw, Poland, IEEE Computer Society, 2001.

[17] Schmidt, H., and W. Zimmerman, "A Complexity Calculus for Object-Oriented Programs," *Object-Oriented Systems*, Vol. 1, No. 2, 1994, pp. 117–148.

T:\Books\Crnkovic\Crnkovic31.vp
Wed Apr 24 11:20:59 2002