Software Components in Contexts and Service Negotiations

Guijun Wang, H. Alan MacLean

Applied Research and Technology, The Boeing Company, Seattle, WA 98124

guijun.wang@boeing.com, alan.maclean@boeing.com

1. Software Components

Before the term "component" became popular, terms such as subroutine, procedure, function, module, and object have been used from the earliest days of software development to denote software artifacts. Software developers have a subjective understanding of what they mean when speaking of subroutines, procedures, functions, modules, and objects, including a sense of how the meanings are distinctive one from another. In general, the notions embodied in these terms have been introduced and used with the objective of identifying and addressing issues pertaining to reusable software artifacts, the management of complexity, and the development of more flexible systems. The point of departure of this article is the premise that components should be used to address the same basic objectives, but that they constitute a distinctly new form of software artifact, one that offers better capability in meeting the objectives. As such, the same kind of direct programming language level support available for subroutines, procedures, functions, modules, and objects is required for components.

Software Engineering is an iterative process that comprises multiple stages, including modeling, design, implementation, and deployment. The "component" concept has been utilized in all these stages, albeit without consensus on a common definition. Indeed, a number of definitions for "components" appear in the literature [1]. For example, Kozaczynski defines a business component as the software implementation of an "autonomous" business concept or business process. It consists of the software artifacts necessary to express, implement, and deploy the concept as a reusable element of a larger business system. Szyperski defines a software component as a unit of composition with contractually specified interfaces and explicit context dependencies only. In this context, a component can be deployed independently and is subject to third-party composition. The Gartner Group defines a run-time software component as a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at run-time. Clearly, there are substantial differences between these definitions. In large part, these differences appear to arise as a consequence of different perspectives regarding the software engineering process. More precisely, the definitions reflect views of components at different software engineering stages. For example, Kozaczynski's definition reflects the component concept at the business process modeling stage of a software engineering process, while Szyperski's definition embodies the component concept at the architectural design stage. The Gartner Group's definition reflects the component concept at the deployment stage of a software engineering process.

At the architectural design stage of a software engineering process, Szyperski provides what we believe to be the appropriate general definition of a component. The definition stops short of describing the details needed to characterize what attributes a component should possess, and what functional and nonfunctional elements a component should have, as well as how they should be specified and exposed. In this regard, current research combined with developments in the commercial sector on component models help provide important details. For example, [5] [6] [7] present an architectural component model and describe a "Port and Link" framework to support component assembly for distributed systems. In this architecture level component model, a component's functional boundary is exposed by means of four types of functional elements, namely services a component provides, services a component requires, events a component observes, and events a component generates. Ports are used to represent these functional elements, and act as agents for communication between a component's internal parts and other components outside the component's boundary. Links are used to achieve communications, local or distributed, between components in general via specific local or distributed infrastructures. In a similar fashion, commercial models including

JavaBeans, Enterprise JavaBeans, ActiveX, and CORBA Components explicitly or implicitly allow components to expose their capabilities via the four functional elements. For example, the CORBA Components specification currently under consideration by OMG [4] explicitly allows a component to declare services it provides, services it uses, events it emits, and events it consumes. Rather than using Ports and Links along the lines described in [5] [6] [7], the commercial component models impose various architectural constraints on components, such as the interfaces every component must implement. A comparison between the architectural component model discussed in [5] [6] [7] and the commercial component models is summarized in [5]. Others such as [3] used a sliding interpretation approach to characterize components where a component's interface is characterized by signature, constraints, configurations, and non-functional properties.

We believe that the Szyperski component definition combined with the details supplied by the component models discussed above provides a good working definition of what a component should be in terms of the attributes it should possess, and how its functional elements should be specified and exposed. This characterization distinguishes the concept of a component from that of a subroutine, procedure, function, module, or object, and potentially establishes an appropriate level of abstraction to more effectively address the objectives of these earlier software artifacts. More importantly, this approach provides a foundation for addressing the nonfunctional properties of components and systems of components, which is discussed in the next section.

2. Functional Elements and Nonfunctional Properties of Architectural Components

As noted in Section 1, components exist in different contexts within the software development lifecycle, including business process models, software architecture, and real software systems. Here we focus on components at the software architecture level, which we refer to, naturally enough, as architectural components [3] [5] [6] [7]. As a working definition, we take the phrase "software architecture" to mean the collection of components constituting a system, together with a description of their interactions, as well as the constraints imposed on the components and interactions. Components in an architecture have contractual obligations: they provide services to others and/or generate events that others may be interested in, and they may require services from others and/or observe events generated by others. These contractual obligations are the functional elements of a component's boundary in an architecture context.

Nonfunctional properties of architectural components include performance, reliability, security, and the other so called 'ilities' that contribute to the overall Quality of Service (QoS) provided by the system. The characterization of components adopted in the previous section is central here because it enables nonfunctional properties of a component to be addressed via the functional elements of the component. In fact, as a consequence of the separation of the services provided by a component from the services required by a component, we can specify nonfunctional properties that a component offers (for provided services) and nonfunctional properties that a component expects (for required services). Of course, for components in a specific architecture, these nonfunctional properties must be specified in concrete terms to be useful. We illustrate these notions in the examples that follow.

An Architecture Style is a family of software architectures that share common architecture properties [2]. For example, the Supplier-Mediator-Consumer is an architecture style known for its flexibility and maintainability due to the separation of suppliers and consumers. A simple instance of this architecture style is the Supplier-Buffer-Consumer architecture where the Buffer is a simple Mediator. For simplicity, let's assume that the data which the Supplier supplies, the Buffer stores, and the Consumer consumes is a data stream of a simple type, say Integer. In addition, let's assume that the Supplier pushes data into the Buffer, the Consumer pulls data from the Buffer, and the Buffer has a fixed size. The functional elements of the Supplier component, the Buffer component, and the Consumer component can be specified as follows: **Supplier**: provides *InputDataStream* service, **Buffer**: requires *InputDataStream* service, provides *OutputDataStream* service, and **Consumer**: requires *OutputDataStream* service.

The *InputDataStream* service can be defined by an interface InputDataStream with operations like "void

nextInputItem(int)", and the *OutputDataStream* service can be defined by an interface OutputDataStream with operations like "int nextOutputItem()".

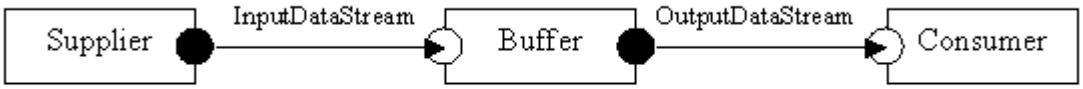

Figure 1. Components in the Supplier-Buffer-Consumer architecture

To describe the nonfunctional performance property of the Supplier, Buffer, and Consumer, we may specify that the Supplier provides data at a rate *alpha*, the Buffer expects input data at a rate *x* and offers output data at a rate *y*, and the Consumer expects data at a consuming rate *beta*. Suppose we have a QoS requirement stating that each data item in the Buffer must be consumed once and only once. Since the Buffer size is fixed, the following constraints must hold to avoid buffer overflow: alpha <= x, beta = y. Assuming the Buffer does not do any further data processing, we know that x = y. As a consequence, we derive the constraint beta = alpha.

Up to this point, our specification of the nonfunctional performance property assumes that the inter-component communication time (the time to transfer data from the Supplier to the Buffer, and to transfer data from the Buffer to the Consumer) is negligible. This assumption is more than likely not true, especially for distributed systems, in which it may take a substantial amount of time to transfer data from one component to another. Thus it may also be necessary to specify nonfunctional properties on communication links (for Links see [5] [6] [7]). In a distributed Supplier-Buffer-Consumer architecture, we can accomplish this by specifying that the time it takes to transfer one unit of data from the Supplier to the Buffer is t1, and the time it takes to transfer one unit of data from the Buffer to the Consumer is t2. Then the time it takes for one unit of data to reach the Buffer is t1+1/alpha, and the time it takes for one unit of data to be taken out of the Buffer is t2+1/beta. To avoid buffer overflow, the constraint t2+1/beta <= t1+1/alpha must hold. Hence, we obtain the same constraint beta = alpha as in the case when the inter-component communication time is negligible (i.e., t1=t2=0).

A set of constraints is "under constrained" if there are more unknown variables than the number of constraints. Generally speaking, constraints for component nonfunctional properties in an architecture are under constrained. Moreover, some constraints may have an infinite number of potential solutions. As a consequence, negotiations among components to establish desired nonfunctional properties may be necessary, and system QoS may vary as a result of such negotiations. At the architecture level, components are primarily conceptual, and while constraints can be imposed, many negotiations cannot occur until the components are assembled into a real system or until run-time. We explore this point further in Section 3.

For simple architectures like the Supplier-Buffer-Consumer described above, a service request from one component may be satisfied in a straightforward manner by a service provided by another component. For complex architectures, however, a component that requires a service may have to go through a number of steps to obtain the right service from a provider. For example, the CORBA Event Model architecture, which also employs the Supplier-Mediator-Consumer architecture style, has three types of components: Supplier, Event Channel, and Consumer, each of which can be typed. The Event Channel plays the role of the Mediator. For any two components, communication between them can use either a Pull interaction style or a Push interaction style. Figure 2 illustrates the CORBA Event Service architecture and associated component roles. The functional specifications for the components in the CORBA Event Service can be found in [4]. A Typed Push Supplier that intends to connect to and communicate with a Typed Push Consumer in an Event Channel must first use the "TypedSupplierAdmin for_suppliers()" operation to obtain a TypedSupplierAdmin component and then use the "TypedProxyPushConsumer obtain_typed_ push_consumer(in Key supported_interface)" operation to obtain a TypedProxyPushConsumer component. Next, the "Object get_typed_consumer()" operation is used to get the typed push consumer from the TypedProxyPushConsumer component and narrow it to the expected type, the *supported_interface* parameter

in the obtain_typed_push_consumer(in Key supported_interface) operation. Finally, use the "void connect_push_supplier(push_supplier)" operation to actually connect to the typed push consumer. Note that only the typed push consumer offers services required by the typed push supplier, and that a series of negotiations are required to obtain this service. The example shows also that functional service negotiations may be necessary to match a service request with a service provision. This kind of negotiation results primarily from using the aggregation method to form a larger component by composing other components, a point that will be discussed further in the Section 3.

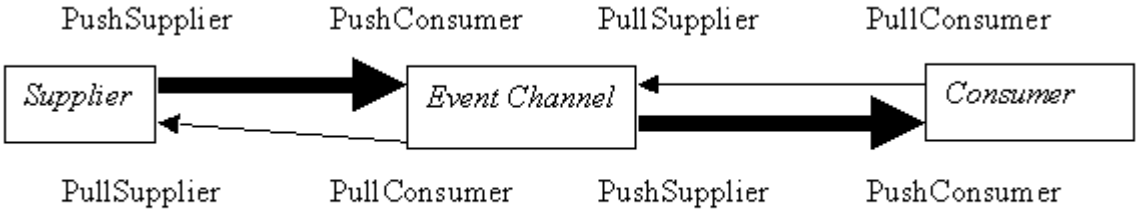

Figure 2 CORBA Event Service Architecture and Component Roles

3. Service Negotiations: functional and nonfunctional negotiations for components in real systems

The functional elements (services and events) and nonfunctional properties (performance, reliability, etc.) that are specified for components during the architecture phase of the software development process reflect only those characteristics that are critical to the system from the perspective of the overall architecture. In order to construct real systems based on a selected architecture, however, components need not only possess the functional and nonfunctional characteristics specified in the architecture, but also those functional elements and nonfunctional properties that arise as specialized requirements in a specific application area. We saw in the previous section that the acquisition of desired functional and nonfunctional characteristics often requires negotiation between components in the system. In the case of nonfunctional properties, these negotiations cannot happen until the implementation phase, when components are to be assembled into real systems.

3.1 Negotiations of Nonfunctional Properties

While desired nonfunctional properties can be specified at the architecture level via components, and constraints on nonfunctional properties may be imposed by an architecture, it is only at the implementation level that a component can concretely state how nonfunctional properties are supported. The manner in which a component implements its support for a nonfunctional property in terms of values offered or expected varies from a single fixed value, to a discrete set of choices, to a continuous range of values. Further, the real system under construction very likely has a variety of concrete QoS requirements that first appear during the implementation stage. To illustrate these points, we revisit the Supplier-Buffer-Consumer example described in the last section, but this time in the context of a real application.

Suppose we are to build a signal processing application based on the Supplier-Buffer-Consumer architecture. Here the Supplier is an analog/digital Sampler component and the Consumer is a Compressor component. The Sampler component offers data at three different sampling rates: 8kb/s, 14kb/s, or 16kb/s. The Compressor component is capable of compressing data at 2.5 ms per frame (32b/frame) if it runs on a 12 MHz microprocessor, 2.125 ms per frame if it runs on a 16 MHz microprocessor, and 1.75 ms per frame if it runs on a 33 MHz microprocessor. Suppose the communication times between the Sampler and the Buffer and between the Buffer and the Compressor are negligible. It is not difficult to calculate that the Compressor data consumption rate is approximately 12800b/s, 15058b/s, and 18285b/s, respectively for the three different microprocessor speeds. The architecture constraint beta = alpha, where alpha is the data supply rate and beta is the data consumption rate, limits the number of choices for a sampling rate as a consequence of the available compression rates, but may not uniquely restrict the sampling rate to one choice with respect to

the compression rate. For example, the required resources may be offered to the Compressor component enabling it to compress data at a rate of at least 1.75 ms per frame, and thus the Sampler can attain the maximum sampling rate of 16kb/s. The ability to negotiate a higher sampling rate to provide higher signal fidelity may increase the overall QoS offered by the system. If the Compressor component is to negotiate the sampling rate dynamically, the Compressor may find it necessary to ask the Buffer to issue a request informing the Sampler to adjust its sampling rate subject to the architecture constraint beta = alpha. In this case, the Buffer acquires increased responsibility, and we must implement additional interfaces to support the required negotiations between the Sampler and the Buffer, and between the Buffer and the Compressor.

It is worth noting that it is also possible to have mismatches, or even conflicts, for nonfunctional properties. For example, suppose that a video Supplier produces video at 30 frames per second, while the associated video Player can only consume 29 frames per second. In this case, the constraint beta = alpha in the Supplier-Buffer-Consumer architecture cannot be satisfied. Such a system may still work, but with considerably degraded QoS (e.g., the Buffer discards a frame per second and, in effect y=x-1). Another example occurs when the range of values offered by one component in support of a nonfunctional property overlaps, but does not coincide, with values expected by another component. In such situations, negotiation to obtain a mutually acceptable value may be necessary.

3.2 Negotiations of Functional Services

The discussion in this section centers around the negotiation of functional services, and how the need for such negotiation arises.

First, the negotiation of functional services may result from the fact that a number of steps are required to match a service expected by one party to a service offered by another party. This may depend simply on how the components involved in the negotiation were developed, e.g., as a consequence of whether they were constructed using either the Containment or Aggregation method of component composition. A component composed from "parts" using the Containment method offers services as a whole and does not expose its internal parts, even though certain services are performed by its parts. Interactions with the component will always go through the component as a whole. On the other hand, a component composed from parts using the Aggregation method offers initial services that will lead to the exposure of services offered by its parts. Interactions with the component will actually be with the part directly once it is exposed. Since multiple parts of an aggregated component may offer the same service, a series of negotiations may be necessary in order to find a right part to interact with. One example of this is the CORBA Event Service architecture discussed earlier. There might be multiple push consumers of the same type in the Event Channel. As shown in the last section, only the typed push consumers offer the services required by the typed push supplier, and a typed push supplier must go though a number of steps to find a typed push consumer. Once it finds a typed push consumer, it interacts with the consumer directly.

A second reason that the need for negotiation of functional services may arise is that a component may offer multiple services that eventually achieve the same purpose, but with different QoS characteristics. For example, a Store might offer two shopping services: one allows negotiation for discounts, but takes longer to deliver, while the other does not offer discount negotiation, but delivers more quickly. Whether a Shopper uses the shopping service with discount negotiation or the one without discount negotiation depends on the Shopper’s desired QoS in terms of cost, quantity of purchase, urgency, and so on. A component that offers multiple levels of similar services, but with different security requirements provides another example of this situation.

A third reason that negotiation of functional services may occur is simply that there may be mismatches between services offered and services expected. Simple cases are syntactic mismatches involving different names, different operation signatures, different orderings, and so on. Syntactic differences are typically easy to bridge. On the other hand, the mismatches may be semantic. A simple example of this situation is when the expected data type is Real, but the service offered returns an Integer type. Another example occurs when the communicating components involved in the negotiation are implemented in different languages on different

platforms. Semantic mismatches may be resolved by employing adapters to bridge the difficulty, or they may not be resolvable at all.

Finally, negotiation of functional services may be necessary because the availability of certain services is unknown until a later time, e.g., until run-time. In this situation, a component must be prepared to negotiate in order to find a matching service provider at run-time.

4. Conclusions

This paper examines definitions of software components and argues that different component concepts arise in relation to the different stages of the software engineering process. We suggest combining Szyperski's general component definition with the details supplied by component models to obtain the right mix of ingredients for defining what a component should be, including what attributes a component should possess, and how component functional elements and nonfunctional properties should be specified and exposed. The paper then focuses on issues related to components at the software architecture level and the implementation level.

For Architectural Components, this paper suggests a concrete approach to describe component functional elements and nonfunctional properties. Component functional services can be separated into services a component provides and services a component requires. Nonfunctional properties correspond to services provided and required, and can be characterized in terms of parameters a component offers or expects. The paper also demonstrates how architecture constraints on component nonfunctional properties can be specified and analyzed. While it is possible that a service provision matches exactly with a service request, service negotiations both in terms of functional elements and nonfunctional properties are necessary in many cases. Service negotiations may be quite complex and may impact the Quality of Service provided by the system. This paper characterizes and demonstrates service negotiations using several examples. It also argues that many service negotiations cannot be carried out until component assembly time or run-time. It shows that the issue of service negotiation is a fundamental consideration in the development of component-based systems.

5. References

[1] Alan Brown, Kurt Wallnau, "The Current State of CBSE", IEEE Software, October 1998, page 37-46.

[2] D. Garlan, R. Allen, J. Ockerbloom, "Exploiting Style in Architectural Design Environments", Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering, December 1994.

[3] Jun Han, "Characterization of Components", 1998 international workshop on component-based software engineering, Kyoto, Japan, April 25-26, 1998.

[4] OMG, OMG Event Service specification, http://www.omg.org

[5] Guijun Wang, Liz Ungar, Dan Klawitter, "A Framework Supporting Component Assembly for Distributed Systems", Proceedings of the Second Enterprise Distributed Object Computing, page 136-146, San Diego, CA, November 1998.

[6] Guijun Wang, "SoftBean Composer: a Visual Environment for Component Assembly", in Proceedings of the IEEE 14th Symposium on Visual Languages, Halifax, Canada, page 92-93, Sept. 1998.

[7] Guijun Wang, H. Alan MacLean, "Architectural Components and Object-Oriented Implementations", 1998 international workshop on component-based software engineering, Kyoto, Japan, April 1998.