Exposing Models of Behaviour of E-Service Components

Giacomo Piccinelli[♣]
Hewlett-Packard Laboratories (UK)

Abstract: As EAI (Enterprise Application Integration) emerges, the concept of service components as complex standard objects becomes inadequate. A major strength of the object model is implementation hiding. The execution logic behind a method invocation is transparent to the user, and the assumption is that the signature of the method contains all the information the user needs to know. We suggest that in the problem domain in which service components are used, this assumption is too strong. For the services (methods) offered by a service component, a more comprehensive conversational schema replaces the basic invocation-result model of interaction. The conversation triggered by a service request may involve a number of different components in a number of different roles. We refer to these conversations as the observable behaviour of a component, and we propose they should be part of the meta-data exposed by a component.

In this work, we propose a process-oriented approach to behaviour modelling for software components. An example of possible implementation for our proposal is given with reference to EJB (Enterprise Java Bean) components.

1.Background

The concept of interface is fundamental in the object model. An interface abstracts the internal characteristics of an object, and it defines the contract that the object offers to the outside world. Extending the concept of interface to components, an interface should indicate what the component requires from a system, and what it provides to the system. The classic structure for the interface of a component consists of a set of functions, of which only the signature is known. The type and meaning of the result expected from the invocation of a function are inferred from the result type in the function signature and the name of the function. From the type of the parameters required by the function, it is possible to infer some of the resources involved in the internal processes of the component. Nevertheless, very little information is given about the observable behaviour of the function.

The concept of *observational behaviour* of a system appears in many theoretical frameworks (e.g. CCS [7], CSP [6], modal and temporal logic [2]), and it models the information available from the "observation" of the system during its evolution. The interest in this kind of model comes form two main directions. One the one side, the observable behaviour of a system can be used to infer properties of the system. On the other side, information on the system can be used to infer properties of its observable behaviour. The notion of "observational equivalence" of two systems is central to all these conceptual frameworks. The classic approach to component interfaces is based on very strong assumptions about the way in which they interact. The ability (interest) to deliver all the parameters in one go, and to deliver the result in one go, are examples of such assumptions. The idea that all the behaviour of a component should be hidden is another example. While these assumptions are valid in many significant cases (e.g. middleware), a number of equally significant cases are emerging in which they become inadequate (E-Services).

We propose that the standard notion of interface for a component should be replaced by the more comprehensive notion of *behavioural interface*. Leveraging on the observable behaviour of components [3], we propose an interface model, in which the interaction process involved in a service execution is explicitly modelled. This is fundamental in order to address problems like composition and management for complex components [1, 9]. The enforcement of the proposed model can be useful for static design/implementation of a system, and for run-time deployment and management of the system itself. Component-level policies and mechanisms can be enforced for problems like access control, usage management, and billing.

2.Limitations for standard component interfaces

The main reason why standard component interfaces are so widely adopted (e.g. CORBA, COM, Java, C++, and CGI), is that they usually offer a good compromise between management complexity for the interface and the

♣ Also member of the Software Engineering Group at UCL - University College London

complexity of the actual component. For example, a function invocation has a very coarse granularity for the parties involved. Using the client-server metaphor, after the client has sent a request to the server, it loses any visibility on what happens until it gets back some form of result. The situation is symmetric on the server side. Neither the client nor the server know what their counterpart is doing, even if they may be engaged in other form of communication in order to determine the result of the original request. The interaction logic is hidden into the internal behaviour specification for each component, and formal/automatic reasoning on this logic is difficult.

The common model for components derives from the metaphor of a black box: if the client makes the right question, it gets the right answer. The emphasis already is on the correct interaction between client and server, but it is obfuscated by the assumption that the server can get all what it needs at the beginning, and it delivers the entire result in one go. The assumption is acceptable only for very simple components, offering specialised services [4].

As the component model finds applications in new domains and at higher levels in the system abstraction stack, the complexity of components grows and the room for mutual adaptation becomes smaller. In order to cope with the complexity of new systems the idea of the black box is more important than ever before, but it needs to be extended. Fundamental addition to the metaphor is the fact that the box no longer has only one way in and one way out, but a number of connectors to the external world that have to be properly plugged in. Moreover, it is no longer enough to specify only *what* can flow in the pipes, but also *when* there is something flowing.

An explicit definition of the interaction that a component has with other components of a system has important implications in terms of performance and security management [5]. Depending on the connections a component needs to build for the execution of a service, different policies could be enforced for the deployment of the component itself and/or the components that have to interact with it. Connections to a component may also involve billing mechanisms, which require specific handling. With explicit knowledge of all the inter-component connections, as well as the conditions under which each of them can be used, fine-grain component-level access control mechanisms and policies can be enforced.

3.Behavioural Interfaces

Exposing the behaviour of a component goes against the very nature of the component model itself, and it requires careful considerations. In our proposal, we concentrate on the observable aspects of component behaviour.

The formalisation of some of the concepts proposed relies on multi-sets. The concept of multi-set is an extension of the notion of set, whereby more instances of the same element can be present at the same time. As a notational help, the mark $^\blacklozenge$ is used to remind the reader about situations in which multi-sets are involved. The mark $^\blacklozenge$ applied to a data structure indicates the multi-set composed by the basic element of the data structure (e.g. $(a, b, c, a, a)^\blacklozenge \approx \{a, a, a, b, c\}$).

We need the following support definitions:

***Definition* (History)**: *Given a set V, we define a history H_V any element of $(V^*)^*$.*

***Definition* (Sub-History)**: *Given a history $H_V = (v_1, ..., v_n)$ where $v_i = (v_{i1}, ..., v_{iM})$. We define a sub-history of H_V any history $S_W = (w_1, ..., w_n)$, where $W \subseteq V$ and $w_i \subseteq v_i$. The notation is $S_W \sqsubseteq H_V$.*

We propose the following core definitions:

***Definition* (Observable Action)**: *Given a component C, we define an observable action for C any action that can be performed by C and that involves getting information from entities external to C, or giving information to entities external to C.*

***Definition* (Observable Property)**: *Given a component C, we define an observable property of C any information that can be exposed by an observable action.*

***Definition* (Observable History)**: *Given a set P of observable properties, we define an observable history OH_P on P any history on P.*

***Definition* (Observable Behaviour)**: *Given a set A of observable actions, we define an observable behaviour OB_A on A any history on A.*

The concept of a *behavioural interface* deals with explicit knowledge about the way a component interacts with the external world (possibly, other components) during the execution of a function. This goes beyond the static understanding of how to connect it to the other components in order to guaranty the compatibility of the data exchanged. The idea is to capture the full *interaction process* associated with the execution of a function.

Expanding on the interpretation of the previous definitions, observable properties deal with information that has been exposed to external observation by observable actions. For example, an observable action can be sending an array V of integers. A related observable property could state something like "average(V) = 5". Something like "average(V) > max" is not valid unless the value "max" was previously exposed by a different observable action. Different types of information can be associated with an observable action, like the roles of the parties involved or QOS requirements. A set of observable properties can be used to capture the state of an interaction process, and an observable history captures the evolution of the state.

Considering A the set of all observable actions that a component can perform, an element of A* can be interpreted as a multi-set of observable actions that the component can perform concurrently at a certain point in time. The observable behaviour of a component represents the evolution of its external interaction activity while the interaction process progresses. We propose that a coherent management of the information on both the state and the ongoing actions is fundamental for the comprehensive management of an interaction process.

Definition **(Behavioural Interface)**: *Given a set A of observable actions, P_A the set of all the observable properties on A, H_A the set of all the observable histories on P_A, and B_A the set of all the observable behaviours on A. A behavioural interface on A is a function $\jmath$ of the form:*

$$\jmath_A : H_A \times B_A \rightarrow A^*$$

Given an observational history and an observational behaviour, the function $\jmath$ returns a multi-set of observable actions. These actions can be concurrently executed at the present step in the interaction process. The multi-set returned is assumed to be Æ where the function is not defined.

A behavioural interface not only gives information on the content of the information exchanges involved in the interaction process (looking at the single observable actions). It also gives information on the sequence, concurrency and conditional aspects of the process. A more comprehensive description for the concept and usage of behavioural interface can be found in [10].

4.E-Service Bean

As an implementation example of the component model proposed, we instrumented an EJB (Enterprise Java Beans [8]) platform with process-oriented componentisation capabilities. The work revolved around the implementation of a new type of EJB container, within which an XML-based process description file can be used to model the observable behaviour of the bean. Clients and other beans will only be able to execute methods on a bean in this container if they are consistent with the process description. The outbound communication initiated by the bean is also monitored for compliance with the behavioural interface captured in the process. In line with the naming conventions for EJB, we refer to the new container as ESB (E-Service Bean).

A bean models a service unit, and the process description captures the service delivery process deriving from the external interaction of the bean. Different roles can be involved in the delivery process behind the service implementation. The ESB container manages at run-time the behaviour of the entities playing these roles. When a bean is created, the roles involved can be partitioned into groups and assigned transparently to either client programs or other beans. The only interaction allowed is the one deriving from the process description (both inbound and outbound). The aim of our prototype was to implement a basic container that demonstrates this kind of protection for the beans. The container in which service beans are to be deployed has the following features not found in normal EJB containers:

- The bean provider can specify the service behaviour in a process description file (using XML) that is then enforced by the container. This means that the container will generate exceptions whenever a method is called in an incorrect way (at the wrong point in the process or with invalid parameters). Exceptions will also be generated in cases where a service bean invokes a method on another service bean that does not comply with the specified behaviour.

- A process can be specified to have a number of roles that can be played. Clients can create a bean, specifying the role(s) they want to play, or contact an existing bean to have a role/roles assigned to them. The container makes sure that a service cannot be started until all roles are assigned.
- The client can request role specific descriptions from a service bean to see what is required to do as the entity responsible for a specific role/roles.
- The system makes the state of each service instance persistent so that everything can be reconstructed in the event of a system crash.

The tasks performed by the system can be divided into two parts. The first part is the creation of the home and remote object classes. The second part is the actual runtime handling of the beans, where the home objects are made available via JNDI allowing them to be created and used. A paper containing the full description of the ESB platform has been submitted for review.

5.Conclusions

Components are moving up in the abstraction layers for system and solution architectures. Particularly in the e-business space, there is a clear interest in identifying well-defined aggregates of functions as service units. The e-service model focuses on the electronic virtualisation of service units in a way that they can be traded, managed, and delivered electronically.

An e-service derives from the business processes of a service provider, and has to be connected to business processes of the service consumer. In this paper, we propose that the observable behaviour of complex components like e-services should be formalised and externalised. We perceive this as a fundamental step in making component-based e-business systems more manageable in terms monitoring capabilities and predictability. We briefly present a process-oriented model for component behaviour, and a prototype implementation based on the EJB architecture (Enterprise Java Beans).

References

[1] Abadi M, and Lamport L. *"Composing specifications"* Digital Equipment Corporation TR66/90, 1990.

[2] Arapis C. *"Object Behavior Composition: A Temporal Logic Based Approach"* In Object Frameworks, D. Tsichritzis (Ed.), Centre Universitaire d'Informatique, University of Geneva, 1992.

[3] Cicalese C.D.T. and Rotenstericht S. *"Behavioural specification of distributed software component interfaces"* In IEEE Computer Journal, vol. 32 n.7, 1999.

[4] Garlan D., Monroe R. and Wile D. *"ACME: an architecture description interchange language"* In Proc. of CASCON'97, 1997.

[5] Hewlett-Packard *"E-Speak. Architecture specification"* Available from http://www.e-speak.hp.com, 1999.

[6] Hoare C.A.R. "*Communicating Sequential Processes*" International Series in Computer Science, Prentice Hall, 1985.

[7] Milner R. *"A calculus of communicating systems"* In Lecture Notes in Computer Science Vol. 32. Springer-Verlag, 1980.

[8] Monson-Haefel R. *"Enterprise Java Beans"* O'Reilly, 1999.

[9] Nierstrazs O. and Meijler T.D. *"Research directions in software composition"* In ACM Computing Surveys, vol. 27, n.2, June 1995.

[10] Piccinelli G. and Lynden S. "Concepts and Tools for E-Service Development" Proc. 7th International Workshop HP OVUA, 2000.