Validating Message Sequences in Web Services

Web Services are in the vanguard of a new breed of ad hoc cooperative processes potentially involving the interactions of several agents/processes/organizations. This puts demands on the expressed interfaces of these participants beyond what is presently supported by traditional OO class definitions as currently reflected in WSDL. The primary difficulty with "traditional" interfaces, such as class interfaces or WSDL definitions, is their concentration on the allowed operations while ignoring the importance of sequences of operations. In the business world, at least, this order is very significant - whether goods are paid for before they are shipped, or vice versa, can be very important to participants. Therefore, Web Services will require ways for agents to express which sequences of methods are acceptable to them, ways to negotiate over allowed sequences, ways to compose them, and ways to type-check both the resulting agreements and the eventual sequences.

In the past we have done research in mobile objects and concentrated heavily on exploiting markup (XML and its parent, SGML) for describing domain-specific languages for agent communication. The latter effort was prescient, given the current role of XML in Web Services. More recently, we've focused on helping develop the XML Schema language from the W3C. Our interest in Web Services represents a return to interest in agent interactions. We've been exploring the parameters of cooperative processes among multiple parties in the context of electronic commerce at Commerce One.

In essence, distributed systems of this sort require not just lists of methods, but also protocols of interaction. These protocols need to be made explicit just as lists of methods currently are. We've seen a number of proposals in the Web Services arena attempting to provide such a protocol. These include Microsoft’s XLANG, IBM’s WSFL, the Business Process Management Initiative’s BPML, and ebXML’s BPSS, not to mention various proprietary languages and systems not yet vying for the status of a standard. More traditionally, process algebras have been used to describe complex protocols.

One interesting way to generalize this as an evolution from "traditional" OO is to consider a class definition as defining a language: the methods form an alphabet, and the set of legal sequences of method invocations describes the grammar. An object becomes a language processor. The language described by a traditional class is any finite sequence, or any finite sequence...
terminated by a destructor. Adding a description of an object’s protocol in terms of permitted sequences becomes just a straightforward evolution. Of course, the immediate questions are: What are the allowed grammars? How can they be type-checked? Obvious choices are regular expressions and context-free grammars, both of which are similar to process algebras.

Adding to the difficulty is the requirement to model the interactions among the parties. While regular expressions and CFGs might reflect the isolated behavior of a single object, a collaboration must include the mutual interactions of the parties – if A sends a message to B, then we must also model the requests that B can make of A (or of other parties) before answering.

Beyond process calculi, the pi calculus is being viewed as a likely entrant for describing these protocols; XLANG, for example, has its roots in the pi-calculus. While the “raw” pi-calculus as such doesn’t make much attempt to distinguish among the different agents in a protocol (it is only concerned with ports and their interactions, where each port itself behaves like a free-floating method), it is not too difficult to layer a language on top of the pi-calculus to organize ports as if they were methods in objects. However, the added complexity of the pi-calculus once again increases the difficulty of finding an adequate definition of type-safety. In this arena, type safety means not only that message parameters are correct, but that they are in the correct sequence as well.

We may be able to exploit early efforts to encode linear logic in the pi-calculus to accomplish this, by turning around the encoding and using linear logic as a typing system, just as the classical intuitionistic logic has proven useful for type safety of sequential programs in the lambda calculus. Various other type systems have been proposed for the pi-calculus of varying degrees of ambition and tractability will be an important consideration – it is not desireable for type checking to become a denial of service attack.