By L.G. Meredith and
Steve Bjorg

CONTRACTS AND TYPES

Addressing the need for a more formal account of service descriptions and service contracts with a mobile process algebra.

WEB SERVICES ARE GAINING MOMENTUM AS A platform, both conceptual and technological, upon which to develop applications that take advantage of the Internet infrastructure. The popularity of standards and proposed standards like XML, WSDL [4], WS-ReliableMessaging [2], WS-Security [2], BEPL4WS [5], WSCI [1], and others lend credence to this claim. Much to their credit, these and other standards have led to the development of a basic substrate of tools for wiring together the nodes of a distributed application. For example, Microsoft's .Net offering provides support for XML and WSDL, and its WSE 1.0 supports many WS-* standards and proposals.

On the other hand, by and large, these standards say little about the correct functioning of

such an application. WSDL, for example, only addresses the message format and transport bindings at (aggregates of) endpoints. This situation is unfortunate because much of the substantive complexity of distributed heterogenous applications lies in the interoperation of the services, not the connectivity. All the issues of safety and liveness, for example, arise only after connectivity among the nodes has been addressed. Put another way, even in a world of perfect connectivity with perfect, synchronous communication among perfectly trusted peers, safety and liveness are still thorny issues. And, they are made thornier still in a dynamic environment, such as the Internet, where services discover and interact with each other over the course of a computation.

The situation is exacerbated by proposals like BEPL4WS and WSCI because these proposals do address behavioral aspects of services. From the point of view of correctness, however, these proposals say too much. Both offer Turing-complete languages for orchestrating behavior across ensembles of services. The expressive power of these languages causes them to be more about implementation than specification of useful and interesting properties. At best, these languages, had they formal semantics, might be used for model-checking-style verification. Of course, the worry of a model-checking approach is that one may simply run out of time or space and have no idea whether a given service operates according to the model. Further, throwing more "iron" at the problem, may or may not improve the situation. Finally, exposing implementation even at the levels of abstraction, such as they are, provided by BEPL4WS or WSCI is arguably not the role of a description language, nor of much interest to a service provider. Business value lies in the separation of the "what" from the "how."

The question arises, are there any interesting or useful specification mechanisms in between pure connectivity (WSDL) and full implementation (BEPL, WSCI)? The answer is very positive, since the last 30 years of research on the algebraic specification of behavior has yielded simple, but powerful, methods for specifying the behavior of mobile and distributed systems. In turn, this research has led to a program for the development of type systems for the specification and automatic verification of crucial properties of such systems. These properties include both lock-freedom and leak-freedom. Of necessity, the expressive power of these types systems is below Turing-complete.

Services, Ports, and Port Types

The Web service notion provides an abstraction for autonomous computational entities. Much as it may surprise some, the fundamental abstraction is one based on message-passing. Even WSDL makes no assumptions about synchrony, or RPC-like transports. Rather, it proposes an ontology of services built on ports and messages. Messages arrive at ports. Depending on their point of origin, they are either received by the service or by the service's environment. That is, programs and processes requiring access to a Web service access it by sending messages to ports. Likewise, a Web service accesses the programs and processes that make up its environment by sending messages to ports.

Autonomy. This abstraction is appealing from two different vantage points. First, the expanding commercialization of the Internet has made it apparent that autonomy is a reality. Applications must be able to interact across trust boundaries, for example, from shipper to supplier. No single program is in charge. The message-passing nature of the Web services abstraction fits well with the reality of autonomy. Message-passing does not require more centralized resources than are currently provided by the Internet infrastructure. Message-passing, unlike shared-memory, or generalizations thereof, does not require a single, centralized store to govern synchronization and consistency between concurrently communicating applications.

Programming Model Neutrality. Secondly, the Web services abstraction normalizes different programming models. Object-based models can be made to look like Web services. Traditional EAI messaging applications can be made to look like Web services. Databases can be made to look like Web services. The different binding models of WSDL, as it was originally conceived, support this point of view.

This kind of programming model neutrality is essential. The increase of connectivity amongst programs and processes brought on by the expansion of the Internet has also entailed an increase in the heterogeneity of any single process' environment. That is, a more connected program is a more "worldly" program, more likely to be in touch with programs and processes written in different styles, using different operating assumptions.

Scale Invariance. Additionally, this abstraction is neutral in regard to the scale of application. In particular, we expect a business process with a temporal extension of several months probably should be message-driven. Similarly, OS-level components are often message-driven. And, in a world where radio communication may ultimately be cheaper than cable, it is reasonable to think of such components as potentially separated by a network; and, an OS as

a collaborating set of services. So, the notion of Web service provides a very simple ontology out of which a very rich and complicated set of computational phenomena may be marshaled.

Port Types. Of course, this ontology is a little too simple. In practice we know that not every kind of service is willing to handle any kind of message sent to any port. Rather, a service expects that at particular ports certain kinds of messages appear. That is, we expect to be able to discipline the use of ports and need to be able to describe that discipline so that programs wishing to access a service via such a port may do so according to the discipline.

These considerations give rise to the notion of a port type. In the original WSDL specification the notion of port type ultimately described a kind of *sort* or *interface*. That is, when all was said and done, it described a collection of message types (XML schema, really) that were expected to show up at the port, from one direction or the other.

PROCESS TERM	INTUITIVE EXPLANATION
P ::=	A process, P , is recursively defined as
0	the process which does nothing;
| $\Sigma\alpha.P$	or, a choice among several I/O operations, and their continuations, $\alpha.P$;
| (new x)P	or, a process in which the name x is fresh;
| P|P	or, a parallel composition of processes;
| rec(k).P	or, a recursive definition; *
| k	or, a recursive invocation.
α ::=	An I/O operation, α , is either
$x(\tilde{y})$	an input of the ordered tuple of names $\tilde{y}$ at the channel x ;
| $\overline{x}[\tilde{y}]$	an output of the ordered tuple of names $\tilde{y}$ at the channel x .

*In standard presentations of π calculus the recursive definition is replaced with !P, the replication operator. These two combinators are in fact interderivable as the example illustrates.

EBNF Grammar Process Classes.

This sort of typing is useful and can help to catch many kinds of usage errors at service binding time, as opposed to runtime. This alleviates the programmer from writing tedious and repetitive code and eliminates the introduction of many errors committed by getting this code wrong. It also speeds up the runtime of service provision because these checks may be done at service binding time.

Types as Contracts. In providing a way to describe these kinds of usage disciplines, the WSDL notion of port type describes a kind of contract between service requester and service provider. Of course, this is a very impoverished notion of contract. It says really very little about quality of service. It says very little about time. And, it says very little about order. But, all of these things and many others would have to appear in a business-level contract governing an actual service provision.

Of course, the reason such information appears in such a contract is that in a world of autonomous agents acting from either side of a trust boundary, it is necessary to make implicit assumptions about service provisioning explicit. A contract that merely says the service provider will provide certain data upon receipt of a digitally signed request is certainly met if the provider offers the data a quarter of a century later.

Similarly, we see that order, as well as time, is a critical aspect of service level contracts between components, and one that applies at varying temporal scales. For example, device drivers critically depend on ordering and timing information. An OS can be brought to sudden and abrupt halt if it and one of its device drivers differ in their expectations on the order and timing of messages.

Moreover, even though the WSDL port type does allow the contract to specify the provider will provide such and such data upon receipt of a digitally signed request it does not allow any other ordering constraint to be specified. For example, if the protocol was initiated by the service ("you have been selected from a pool of applicants to make the next level request") soliciting the signed request and the granting the data, WSDL simply remains silent. This kind of protocol-level information must be specified elsewhere.

Scope: What Should A Contract Cover? Since this contractual information is practical and necessary to successfully implement and execute service provisioning in the real world, ultimately, we claim, in a real contract language, there will be a place for all of it: quality of service, duration, and order. Here, however, we focus on logical constraints like order and channel format because they have such profound impact on overall service function and of the three kinds of information identified is the most scale-invariant.

There are several reasons for wishing to add these kinds of logical constraints to port types. One arises from the practical need to make more and more of the protocol semantics part of the mechanistically enforced discipline on the use of the port. The extent to which it is not is the extent to which runtime code is burdened with checks for protocol violation.

Another reason is that correct service functioning is really a matter of disciplined communication. If the service requires an initialization message before it is called upon to do work then usually all bets are off

if it processes work requests before being initialized. Similarly, if a service is expecting to be notified that an applicant wishes processing while the applicant is expecting to be notified it may apply for processing, these two agents will make no progress. These are instances of safety and liveness properties that may be addressed if protocol-level information is added to the service description.

Finally, the more such contractual information can be made explicit and mechanized the more service level agreements can be automated. This has impact on the discovery process. If it can be determined before the service is actually provisioned that there will be a problem because the protocol expectations do not match between service requester and service provider, then the service requester can reject this service provider at discovery time.

Saying Too Much. There is a central question at the heart of the Web services contracts discussion: how much protocol information should be exposed at the level of the public contract? Too much protocol information may lead to feasibility and tractability issues. Too little and we will have complicated Web service description for no recognizable gain.

The critique levied against proposals such as BEPL4WS and WSCI may be illustrated in terms of a game. Suppose the contract definition language is a computationally complete programming language. Player (service requester) and opponent (service provider) want to engage in business and want to do so in a timely fashion. Suppose that opponent challenges player, indicating that if player wants to engage in business it should download the following protocol definition, implement, and execute it.

A good sport, player does download the definition. But, not entirely gullible, player looks over the protocol and because the protocol definition may be a program of arbitrary complexity, eventually times out not completely satisfied that there are not clauses in the definition that do not leave player in a disadvantageous position. Still wishing to engage in business, though, player counters with a protocol of its own devising, suggesting that because it cannot ratify opponents proposal, opponent should adopt player's.

Opponent recapitulates player's steps, and, after a certain amount of time, is also unable to ascertain whether or not player's counter is going to leave opponent in an awkward position. What is the mechanistic procedure by which this game converges in a finite time to a protocol that both player and opponent can ratify? There is none. This approach to the problem is not feasible. Contract languages that are computationally complete—and BEPL4WS and WSCI are just two of several currently being considered for standardization—will not work.

Instead, a different approach is required. Our contention is that Web services are naturally implemented as processes as defined in the process algebra literature, but that the properties one would like to observe at the contractual level include, at least, those that may be guaranteed by typing such processes. We observe that there are at least four major classes of problems amenable to remedy by static checks over descriptions of much simpler complexity than a complete programming language. In particular, many aspects of safety, liveness, security and resource management may be addressed by so-called behavioral types [6, 7, 9].

$$(\text{new init work fin})\overline{\text{web}}[\text{init},\overline{\text{work}},\text{fin}].(\text{new x})(\overline{\text{work}}[\text{x}].\overline{\text{fin}}[\text{x}])$$

Figure 1a. Client makes two work requests.

$$(\text{new web})$$
$$\text{web}(\text{init},\text{work},\text{fin}).(\text{rec}(\text{k}).(\text{work}(\text{x}).\text{k} + \text{fin}(\text{x})))$$
$$(\text{new init work fin})\overline{\text{web}}[\text{init},\text{work},\text{fin}].(\text{new x})(\overline{\text{work}}[\text{x}].\overline{\text{work}}[\text{x}].\overline{\text{fin}}[\text{x}])$$

Figure 1b. System in which client talks to server.

Services as Processes

A comprehensive introduction to the theory of mobile processes is beyond the scope of this article. However, one of the great advantages of basing an approach on this theory is that curious readers can find a rich collection of literature on the subject; we encourage investigating it. In particular, Milner's "The Polyadic π-Calculus: A Tutorial" [8] is the best introduction to the subject the authors have read. Suffice it to say it is not an accident the simple ontology adopted by Web services is the same base ontology adopted for the algebraic treatment of mobile processes. This correspondence between the Web service and process algebraic views of the world is the result of an active and vigorous design process. People involved in the development of one are involved in the development of the other. For example, the first author of this article is also co-author of the WSDL specification.

To make things a little more concrete, the π-calculus (and its relatives) are built around the abstraction of the port, often called a name in that setting. Processes are built in terms of synchronization constraints over I/O requests (messages), at a collection of ports (also called channels in the sequel). For example, a mild variant of the π-calculus presented in Milner [8], offers the classes of processes described

by the EBNF grammar shown in the table here.

Example: Initialization, Work, Finalization

The following example provides a simplified implementation of a service that expects to be initialized, then is willing to accept repeated work requests or a finalize request. The initialization request, which is sent on the "well-known" port web, is expected to provide, in order, the initialization data, init, as well as two ports, work and fin, where work and finalization requests are sent. The port, x, merely represents data that might be sent in the work and finalization requests. Note that on receiving a request on the fin port the agent has no continuation, and therefore ceases to interact:

web(init,work,fin).(rec(k).(work(x).k + fin(x))

Rendering this in XML the reader would note a striking similarity to BEPL4WS. Again, the similarity is not accidental. BEPL4WS drew much of its inspiration from the version of XLANG that was used as the internal processing language of the BizTalk engine. That language, the principal designer of which is one of the authors, was explicitly based on an asynchronous π-calculus. Of course, this example is not chosen at random, either. Rather, it abstracts a common pattern found in everyday programming: opening a file or socket, reading and writing to it, repeatedly, and closing it; opening a database connection, querying and updating over it, repeatedly, and then closing it.

A client of this service that makes two work requests before finalizing would appear as in Figure 1a. Completing the example, the system in which client talks to server would be written as shown in Figure 1b. Note the use of the parallel composition to bring together client and server. This technique is one of the hallmarks of process algebras. It allows components to be defined independently and executed autonomously and yet come together interactively; readers are encouraged to consider how this example might be rendered in XML.

Behavioral Types as Contracts

Port Types Versus Process Types. One question to ask when considering behavioral types is simply "What is typed?"—there are at least two possible answers. In one case, only the (collection of) port(s) is typed. Here, the notion of type may be considered an extension of Milner's original notion of sort [8]. This approach is conceptually simple and appealing. The notion of sort, for example, proscribes the shape of data that may flow over a port, much like a data

DESCRIPTIONS BASED ON BEHAVIORAL TYPES CAN FORM THE BASIS OF A NEW KIND OF DISCOVERY MECHANISM, SPECIFICALLY ONE BASED ON PARTIAL BEHAVIORAL DESCRIPTIONS.

type might proscribe the shape of a parameter. The usage types of Kobayashi [7] may be seen as augmentations of the notion of sort to describe ordering constraints to be respected by operations at the port.

But, it is not clear, for example, how a system guaranteeing lock-freedom will address terms of the form $xy.P_0 + zw.P_1$. Because interaction at the port x precludes interaction at z, and vice versa, the choice forms an invisible information link between the two ports; there is no place in a typing system based solely on port types to record this link.

In the other case, the entire process is typed. Here, the notion of type is a homomorphic image of the process. In many systems, such as Kobayashi and Ishiguro's generic type system [6], the type *is* a process. In such systems there will invariably be a type compositor that corresponds to the term compositor for parallel composition. This compositor plays a key role in the semantics of the type systems.

Subject Reduction. Behavioral types systems may also be classified by how they treat the type compositor corresponding to the term compositor for parallel composition ($P_0 \mid P_1$). Effectively, there are two possible interpretations—lazy and eager. In the lazy treatment, as in Kobayashi's systems [6, 7], the type compositor mirrors precisely the semantics of the term compositor. No calculation of the "reductions" of the types is effected in the resulting type. This design decision has two consequences.

First, the system must introduce a set of reduction rules on types that parallels the reduction rules on processes. Secondly, the subject reduction theorem—the theorem indicating how the evolution of processes is related to their types—is a kind of simulation. That is, the subject reduction theorem is of the form

$$P : T \wedge P \to P' \Rightarrow \exists\, T',\ T \to T' \wedge P' : T'$$

In other words, when the process P is typed T and the process can make a move to P', then there is a type T' such that the type T can move to it and P' types T'.

In the eager treatment, the type compositor eagerly calculates what is left of the types after "communication" (between the types) occurs. Thus, in Yoshida, et al [9] the compositor $T_0 \odot T_1$ causes the types of the ports involved in communication to reflect the communication that would occur between the processes they type. This has two corresponding consequences.

First, the system must provide the definition of this compositor that is, in some sense, outside the process algebra compositors. Secondly, the subject reduction theorem is of the form

$$P : T \wedge P \to P' \Rightarrow P' : T$$

In other words, once a process P has a type T, then all evolutions thereof have the same type.

Example: Generic Types. Kobayashi and Igarashi develop a very general notion of behavioral types they call process types [6]. The type system is for yet another variant of the calculus than the one presented previously; the analogous server implementation to the example can be expressed in their calculus in the expression shown in Figure 2. His calculus is minimal and offers a replication operator (!P) instead of a recursive definition. Thus, the encoding is obliged to activate the recursive invocations along a private channel s; their type judgments are of the following form: $\Gamma \vdash P$.

$$\text{web(init).(new s)(!s().(work(r).}\bar{s}[\,]\text{+fin(r))} \mid \bar{s}[\,])^*$$

* Kobayashi actually uses a slight variant on the notation. For comparison, the example would be web?[init].(v s)(*s?[].(work?[r].s![]+fin?[r])| s![]) written in his notation.

Figure 2. Example server implementation.

$$\text{web } ((\text{init})(\Gamma\downarrow_{\{\text{init}\}})).!(\text{work}((\text{r})(\Gamma\downarrow_{\{\text{r}\}}))+\text{fin}((\text{r})(\Gamma\downarrow_{\{\text{r}\}}))) \vdash$$
$$\text{web(init).(new s)(!s().(work(r).}\bar{s}[\,]\text{+fin(r))} \mid \bar{s}[\,])$$

Figure 3. Example types.

The type environment, Γ, types the entire process. The judgment may be read as "Γ abstracts the process, P." The example types are shown in Figure 3. Ignoring the funny shape of the inputs, the expression to the left of the turnstile in Figure 3 (that is, the type), can indeed be seen as an abstraction of the process—note that the private channel activations disappear in the type.

In this setting, well-typedness is parameterized by a user-supplied predicate ok. Depending on the shape of the predicate, well-typedness guarantees a range of properties from lock-freedom to resource management in the form of static garbage-channel collection. It is enlightening to investigate the typing of a client of this service using different predicates.

The example system illustrates there are two different kinds of relationships that are governed by typing. One is evident in the shape of the judgment. It answers the question: does the service provider service actually provide the service advertised by the type? We call this relationship conformance.

The other relationship is more subtle. It answers the question of whether a pair of types match up so that these patterns of behavior allow successful inter-

operation. Again, this is realized differently depending on way the type system treats parallel composition. In an eager treatment, undesirable interoperation will occur early in the composition of types. In the lazy treatment it will show up as a property of the (evolution of the) parallel composition of the types. We call this relationship compatibility.

In this regard, the system also illustrates the flaw in a common misconception regarding compatibility. A cursory investigation might lead one to think these type systems require "matching" is one-for-one dual behavior on either side of a session. Kobayashi's system, however, has no such duality. Parallel composition of services may "leave something over" so that the aggregate service may have behavior it exposes to a third party, as we might see in a practical system involving a client, a supplier, and a shipper.

The usual notion of program correctness—a program meets a specification—corresponds to the "local" notion of conformance. End-to-end correctness is accommodated by successful answers to questions posed by both relationships. Specifically, client must conform to its type. Server must conform to its type. The type of client must be compatible with the type of server. However, it is important to realize that in systems built out of the composition of other systems checking conformance entails checking compatibility.

Interestingly, service discovery may be favorably impacted by the notion of compatibility. As many have observed, a WSDL-based discovery is ultimately based on service names. This means knowledge of how the service name relates to what it does must be magically applied to the search—this usually means a human is involved. When compatibility can be mechanized some of the chores of searching for the right service can be off-loaded to machines. Given that schema for broad classes of service requests (purchase orders, advanced shipping notices) are moving toward standardization, what can be mechanized is that services using these standard schemas exchange documents typed by them without locking up or leaking secrets or resources.

Conclusion

Magazine space constraints preclude an exhaustive survey of existing behavioral type systems here. As mentioned earlier, systems of more or less comparable complexity exist for guaranteeing properties ranging from lock-freedom to leak-freedom. The simplicity of these systems makes rendering them in XML schema routine. Their simplicity, however, belies their value. It is of considerable business value to be certain that when a client binds to a service it has discovered on the Web it can be assured it will not lock up, nor will it leak resources or secrets. These sorts of guarantees are more important to the bottom line than a description of the implementation of the client or the service.

In fact, descriptions based on behavioral types can form the basis of a new kind of discovery mechanism, specifically one based on partial behavioral descriptions. This mechanism crucially depends on the computation of compatibility of types being a terminating computation. Turing-complete service description languages will face difficulties filling this niche in the emerging Web services ecosystem.

Of course, a caveat remains. Whatever is chosen as the basis of the contractual mechanism, it is still a matter of trust, from client's point of view, that a given service actually conforms to the contract it advertises. This suggests even higher level services, for example, so-called reputation services analogous to business rating services like Dun and Bradstreet, will be needed to guarantee end-to-end quality of service. C

References

1. Arkin, A. et al. *Web Service Choreography Interface 1.0.* W3C Note (2002); www.w3.org/TR/wsci/.
2. Atkinson, B. et al. *Web Services Security Protocol.* MSDN Library (2003); msdn.microsoft.com/library/en-us/dnglobspec/html/wsrmspecindex.asp.
3. Bilorusets, R. et al. *Web Services Reliable Message Protocol.* MSDN Library (2003); msdn.microsoft.com/library/en-us/dnglobspec/html/wsrm-specindex.asp.
4. Christensen, E. et al. *Web Services Description Language (WSDL) 1.1.* W3C Note (2001); www.w3.org/TR/2001/NOTE-wsdl-20010315.
5. Curbera, F. et al. *Business Process Execution Language For Web Services.* Internal Document (2003); www.w3.org/TR/wsci/.
6. Igarashi, A. and Kobayashi, N. A generic type system for the Pi-calculus. In *Proceedings of POPL* (2001); www.kb.cs.titech.ac.jp/~kobayasi/papers/generic-pi-full.ps.gz.
7. Kobayashi, N. A type system for lock-free processes. *Information and Computation 177* (2002); www.kb.cs.titech.ac.jp/~kobayasi/papers/IC-lockfreedom.ps.gz.
8. Milner, R. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag (1993); www.lfcs.informatics.ed.ac.uk/reports/91/ECS-LFCS-91-180/ECS-LFCS-91-180.ps.
9. Yoshida, N., Honda, K., and Berger, M. Linearity and bisimulation. In *Proceedings of the Fifth International Conference, Foundations of Software Science and Computer Structure*s (FoSSaCs 2002), LNCS, Springer (2002); www.mcs.le.ac.uk/~nyoshida/paper/fossacs_ca_final.ps.gz.

L.G. Meredith (gregmer@microsoft.com) is a senior software architect at Microsoft in Redmond, WA.
Steve Bjorg (sbjorg@microsoft.com) is a development lead at Microsoft in Redmond, WA.