A Programming Language for Web Service Development

Dominic Cooney **Marlon Dumas** **Paul Roe**

Centre for Information Technology Innovation,
Queensland University of Technology, Australia
E-mail: {d.cooney,m.dumas,p.roe}@qut.edu.au

Abstract

There is now widespread acceptance of Web services and service-oriented architectures. But despite the agreement on key Web services standards there remain many challenges. Programming environments based on WSDL support go some way to facilitating Web service development. However Web services fundamentally rely on XML and Schema, not on contemporary programming language type systems such as those of Java or .NET. Moreover, Web services are based on a messaging paradigm and hence bring forward the traditional problems of messaging systems including concurrency control and message correlation. It is easy to write simple synchronous Web services using traditional programming languages; however more realistic scenarios are surprisingly difficult to implement. To alleviate these issues we propose a programming language which directly supports Web service development. The language leverages XQuery for native XML processing, supports implicit message correlation and has high level join calculus-style concurrency control. We illustrate the features of the language through a motivating example.

1 Introduction

With the increasing use of software applications for the daily conduct of business, the need to link these applications with minimal effort and in short timeframes is becoming ever more evident. Concomitant with the development of this need and greatly motivated by it, Service-oriented Computing (SoC) is emerging as a promising paradigm for enabling the flexible interconnection of autonomously developed and operated applications within and across organisational boundaries (Alonso, Casati, Kuno & Machiraju 2003).

SoC is a distributed application integration paradigm in which the functionality of existing applications (the services that they provide) is described in a way that facilitates its use in the development of applications which integrate this functionality. The resulting integrated applications can themselves be exposed as services, leading to networks of interacting services known as service compositions or composite services (Casati & Shan 2001, Benatallah, Sheng & Dumas 2003). The technology behind the SoC trend is mainly based on standards such as SOAP, WSDL, WS-Security, and BPEL4WS (BPEL for short). This technology enables businesses to describe the services that they offer (generally in an XML-based form), to publish these descriptions online, to find other services based on their descriptions, and to build applications using these services.

Composing Web services is a form of megaprogramming (or "programming in the large") (Wiederhold, Wegner & Ceri. 1992). Two schools of thought have emerged with respect to megaprogramming: one advocating the use of existing programming languages with conservative (or no) extensions and another advocating the use of new languages specifically designed for this purpose (i.e. "megaprogramming languages"). *Process Definition for Java* (JSR207) (Java Community Process 2003) is an example of the former approach whereas the Business Process Execution Language (BPEL) (Andrews, Curbera, Dholakia, Goland, Klein, Leymann, Liu, Roller, Smith, Thatte, Trickovic & Weerawarana 2003) represents the latter approach. JSR207 uses metadata tags to annotate Java programs with service composition elements. This trivially leads to a language with which programmers are familiar, but which does not necessarily lift the level of abstraction enough to deal with the complexities of Web service composition.

By comparison, BPEL can be seen as a programming language specifically designed for Web service composition. BPEL code is automatically generated by design-level tools and developers never need to delve into this code to perform fine-grained changes. However, in many situations fine-grained changes are required and such changes are not easy to incorporate given a language such as BPEL which provides minimal features for fine-grained programming (e.g. data manipulation operations) and which, in addition, does not follow a conventional programming language syntax. Conscious of this issue, some authors of the BPEL specification have defined BPELJ (Blow, Goland, Kloppmann, Leymann, Pfau, Roller & Rowley 2004): a hybrid between BPEL and Java. Like BPEL, BPELJ can be seen as a programming language designed for service composition. However, BPELJ extends BPEL with constructs borrowed from an existing programming language. Note that the emergence of initiatives such as BPELJ does not imply that standards such as BPEL are unnecessary. However, such standards should be used at a higher-level in the software development cycle, to describe behavioural aspects of service interfaces (i.e. "abstract processes") or coarse-grained processes which are executed in process brokers.

Our approach can be placed alongside BPELJ in terms of scope, but it adopts a different perspective. We advocate a completely new language (and thus avoid the problems inherent with conservative language extensions), but we pragmatically combine novel features for Web service development (and service composition in particular) with conventional pro-

gramming language features and syntax.

The contribution of this paper is a number of programming language features aimed at supporting Web services development. These include:

- A stratified integration of XQuery's flexible evaluation semantics with imperative language constructs.
- An application of join calculus-style concurrency to Web service messaging, and an embodiment of this concurrency style as a programming language construct.
- An approach to message correlation that provides direct support for both point-to-point and one-to-many Web service conversations.

The paper is structured as follows. In the next section we motivate the proposal and provide a working example. In Section 3 we describe the specific features, show examples of their use, and discuss some design choices. Section 4 discusses the innovative aspects of our approach with respect to previous work, and Section 5 outlines ongoing and future work.

2 Motivation

SoC brings along a number of specific requirements over previous paradigms (e.g. object and component-oriented) which unavoidably need to be taken into account by any Service-oriented Architecture (SOA):

Explicit boundaries: As services are expected to be developed by autonomous teams, designing SOAs is an inherently collaborative process involving multiple stakeholders from different organisational units. This raises the issue that certain organisational units may opt not to reveal the internal business logic of their services to others, making it difficult (yet indispensable) to ensure global consistency.

Coarse granularity: Services are highly coarse-grained, at least more so than objects and components (Szyperski 2003). Often, a service maps directly to a business object or activity (e.g. a purchase order or a flight booking service). It follows that the design of services (and in particular composite ones) is a complex activity. It involves reconciling disparate aspects such as the involved providers and consumers, their interfaces, interactions, and collaboration agreements, their internal business processes, data, and (legacy) applications.

Process awareness: As services often correspond to business functionality exported by an organisational unit, they are likely to be part of long-running interactions driven by explicit process models (Aalst 2003). Hence, SOA should take into account the business processes as part of which services operate and interact. In particular, collections of services may engage in complex conversations with a dynamically changing set of partners, and a large number of events that may occur in a variety of orders.

The nature of these requirements and their intricate interdependencies introduce a certain degree of complexity in the design and implementation of Web services. While efforts are underway to tackle some of this complexity at the design and middleware layer (e.g. through standards such as BPEL), we argue that there is also a need for programming support. In particular, the high degree of potential concurrency introduced by the "process-aware" nature of Web services calls for the introduction of appropriate abstractions into the languages used to develop and compose Web services.

The brokered procurement scenario depicted in Figure 1 provides a simplified but nonetheless realistic example motivating the need for programming language support for Web service development. This scenario involves a collaboration between three types of entities: buyer, broker, and seller. Each entity is represented by a Web service which sends and receives messages to/from the other entities. Following Alonso et al. (Alonso et al. 2003), we describe the involved interactions and their control-flow dependencies using an UML activity diagram in which each activity corresponds to a "send" or a "receive" task. Note that internal tasks (e.g. invoking a function provided by an internal application) are not shown in this diagram.

An instance of this procurement scenario starts when a buyer requests an offer from a broker. The broker then contacts a number of sellers, which can "bid" for obtaining an order. After receiving three bids or after a timeout[1], the broker either makes an offer to the buyer or terminates the process (if no bids are available). If the buyer accepts the offer, it sends an order to the broker who (only then) reveals the identity of the (winning) seller to the buyer and forwards the order to the seller. The buyer is then responsible for sending the shipment and payment information to the seller who waits for both of these items before sending a shipment notice.

In mainstream programming languages such as Java and C#, implementing this scenario requires the developer to hand-code some subtle and error-prone concurrency and message correlation aspects. In particular, the broker service involves a complex synchronisation point (wait for first three bids or for a timeout) which is difficult to implement using conventional thread synchronisation primitives.

3 Language features

The language's feature set has been designed in response to the three requirements outlined in Section 2: explicit boundaries, coarse granularity and process awareness. Explicit boundaries are supported by encouraging the programmer to work at the *service* level of abstraction. The service is the largest unit of implementation, and only behaviour-less interfaces are shared between services. We intend this lack of a small-scale reuse feature, such as objects, to promote service composition and granular services. Our concurrency and message correlation features make it easy to program service interactions by supporting the process-aware nature of services. Furthermore, we view XML data and messaging as fundamental to Web services, and the language includes features for XML data manipulation.

In this section we sketch the main features of the language and relate them to the buyer-broker-shipper scenario.

3.1 XML data manipulation

To exploit programmer's familiarity, the language integrates usual imperative constructs (sequence, conditional statements, and loops) and adopts XQuery (Boad, Chamberlin, Fernández, Florescu, Robie & Siméon 2003) as the language for writing expressions. The imperative subset of the language

[1] This type of "n-out-of-m" synchronisation cannot be easily expressed in UML Activity Diagrams so we denote it with annotations in the relevant tasks.

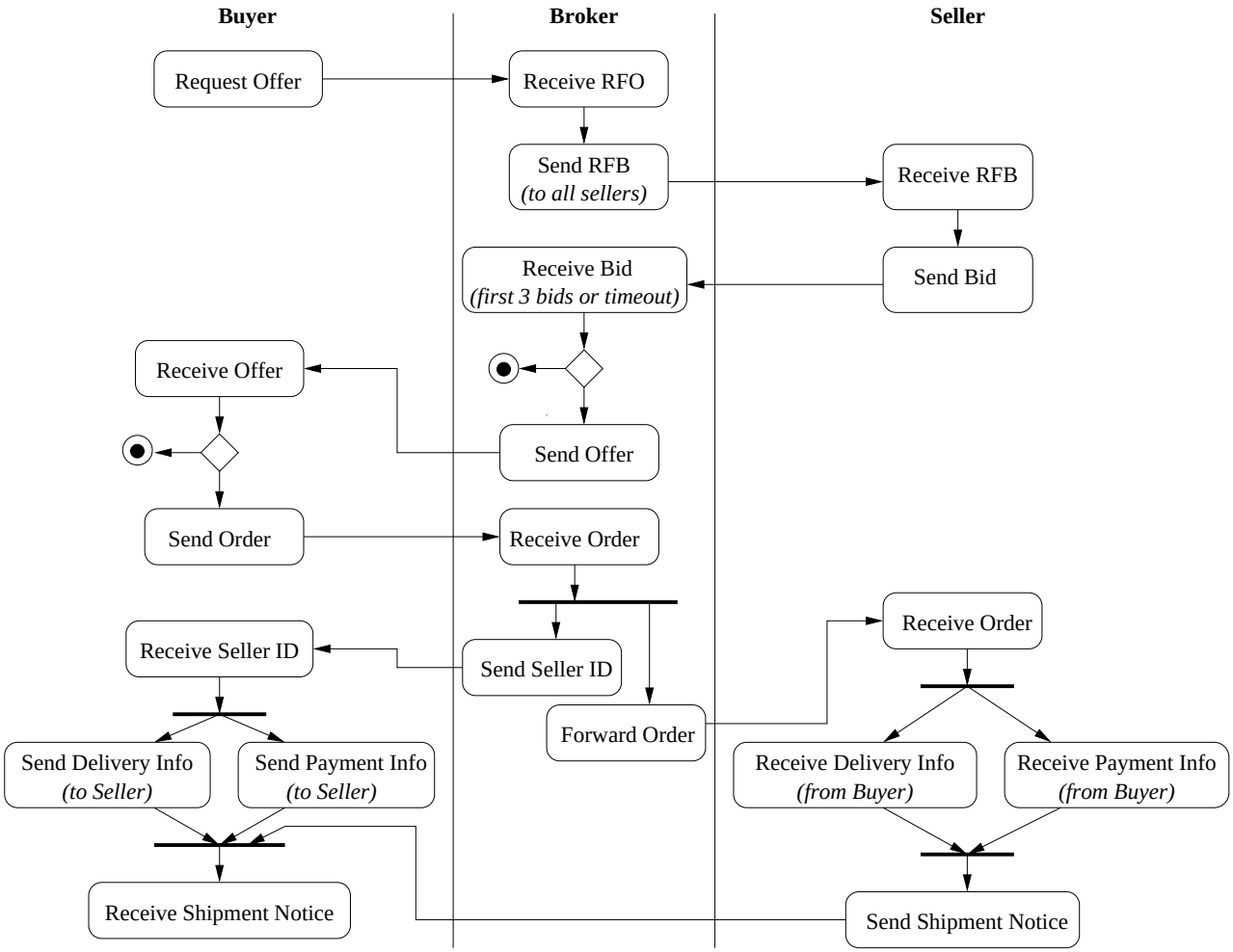

Figure 1: Global choreography for the brokered procurement scenario

(for commands) and the functional subset (for expressions) are strictly stratified. In other words, XQuery expressions can only appear at the bottom of the abstract syntax tree. This stratification simplifies the execution semantics of the language and prevents certain forms of counter-intuitive program behaviour. Indeed, XQuery gives the implementer a lot of freedom in the choice of evaluation order to permit optimisation (i.e. an XQuery implementation can adopt lazy, eager, or any other type of evaluation strategy). If it was possible to access the imperative features of the language from an XQuery expression, in particular the messaging features introduced later, the sequences of messages that may be sent would either be unpredictable, or the language would be obliged to define a detailed execution semantics for XQuery and forsake possible optimisations.

Underlying the choice of XQuery as an expression sub-language, is the assumption that XML Infoset is the fundamental data model of Web services, and hence, a language for Web services development should explicitly support consuming and synthesising XML data. Note that the XQuery data model is a superset of the XPath data model and the Post-Schema Validation Infoset (PSVI). The proposed language therefore adopts this model. In addition, when elements of the XQuery data model must be realised as XML (for example, when some data synthesised with XQuery is sent in a SOAP message body) the XQuery's serialization semantics is used.

As a side benefit of adopting XML, our language implementation (currently under development) is able to reuse third party XQuery implementations. On the other hand, a possibly undesirable side effect of this design choice is that some constructs are duplicated in the XQuery and in the imperative subsets of the language. For example, there is an imperative *if-then-else* construct, and there is a functional *if-then-else* construct from XQuery. This situation is analogous to C, C# and Java's *if-then-else* conditional for statements and conditional *? :* operator for expressions, except that in our language, the same keywords are used for both the imperative and the functional conditional constructs. A similar remark applies to "for" loops.

3.2 Messaging

Our language adopts operations, interfaces, and services from WSDL with some modifications. Our declarations for these elements take much of their syntactic flavour from XQuery.

Operations consist of incoming and outgoing messages, however only *in* and *in-out* message exchange patterns (MEPs) (Gudgin, Lewis & Schlimmer 2004) are supported. Cardinality information is not encoded, so *in-out* could map to a WSDL *in-out* or *in-multi-out* MEP– how many messages are sent depend on the imperative implementation of an operation. *Out* MEPs are not encoded because we do not have any high-level feature for service composition that would use this.

Like WSDL, interfaces are collections of operations, and services implement interfaces. To illustrate a service, interface and operation declaration, here is a fragment of the broker described in the example in Section 2:

```
declare interface Broker {
  declare operation Request
    in ($item, action = 'urn:broker:buy')
    out ($seller, $amount, action =
                          'urn:broker:buyResponse')
  ...
}

declare interface Seller {
  declare operation RequestBid
    in ($item, action = 'urn:seller:solicitBid')
    out ($amount)
  ...
}

declare service MyBroker implements Broker {
  (: Broker operations implementations :)
  ...
}
```

Here, *Broker* is an interface that would be shared between clients and providers. *Request* is an in-out operation. *item*, *seller*, and *amount* refer to the body of SOAP messages. SOAP actions are specified using special keyword arguments. *MyBroker* is a service that undertakes to support the *Broker* interface by providing an implementation for *Request*.

Unlike WSDL, our model of interfaces does not include interface inheritance. We take the view that any relationship between interfaces, including whether two interfaces are ultimately provided by the same service, is immaterial. Our language instead works by the programmer asserting that an endpoint will support a particular interface, with runtime faults if that is not the case. There is no concept of service or interface polymorphism. Although services may implement multiple interfaces, services are always referred to via interfaces. This is in keeping with the view that one implementation of an interface should be substitutable with another.

For example, the implementation of *Request* sends a message to sellers by asserting that some endpoint supports the *Seller* interface, and then sending a message to the *RequestBid* operation of the service at that endpoint:

```
(Seller at $seller).RequestBid(...)
```

3.3 Concurrency

The language supports join-calculus style concurrency, whereby operations can be 'joined' in join patterns. Join patterns are written much like a multi-headed method, where the body of the method does not execute until all heads have been invoked. Of course, the heads in our join patterns correspond to operations and not method headers, and join patterns guard statements which execute when incoming messages are available on all operations. For example, in the scenario described in Section 2, the seller must receive delivery information and payment information before it sends a shipment notice. This can be written by joining a pair of messages:

```
Delivery($d) & Payment($p) {
  (: send shipment notice :)
}
```

This service matches up messages sent to the *Delivery* and *Payment* operations of the seller. When a message to *Delivery* and a message to *Payment* are both available, the bodies of the messages are bound to d and p respectively and the statement is activated. If messages to *Delivery* are received without corresponding messages to *Payment* (or vice-versa) the excess messages are queued until a matching message is available.

Services can also send messages to operations that do not appear in an interface. Using these internal operations is a common idiom for encoding state in languages with join calculus-style concurrency. In the example outlined in Section 2, the broker acts on bids from sellers until a certain number are received. To stop a rogue seller submitting spurious bids in an attempt to exclude legitimate bids, we can use a private message to only entertain one reply per-seller:

```
for $seller in $known-sellers
...
  do {
    (Seller at $seller).RequestBid($item,
                        reply-to = BidResponse, ...);
    OnceOnly()
  }

  BidResponse($bid-amount) & OnceOnly() {
    BidReceived($seller, $bid-amount)
  }
```

Here, after the broker sends a message to a particular seller's *RequestBid* operation, the broker produces a message to its private *OnceOnly* operation. When the seller replies with a message to *BidResponse*, the pending *OnceOnly* message is consumed. If the seller subsequently sends messages to *BidResponse* the broker will not act on them because there will be no matching *OnceOnly*.

Collecting bids also uses private messages to maintain state. To reiterate the business rules for collecting bids, the broker reports the best bid of bids received during a certain time, up to some number of bids received (whichever comes first). If no bids are received in the time period, a fault is generated. The broker models this state with four private operations. Two of the operations, *NoBids* and *BiddingFinished*, are simple flags like *OnceOnly* above. The third operation, *NumBids*, counts the number of bids received in the body of the message; the fourth, *BestBid*, maintains the best bid received so far:

```
do {
  NoBids();           (: initial message to NoBids :)
  ...                 (: sleep :)
  BiddingFinished()   (: send a timeout message :)
}

BidReceived($seller, $bid-amount) & NoBids() {
  BestBid($seller, $bid-amount);
  NumBids(1)
}

BidReceived($new-seller, $new-amount) &
BestBid($best-seller, $best-amount) &
NumBids($n) {
  (: compare bids and produce a new BestBid message :)
  ...

  if $n = 3 then
    BiddingFinished()
  else
    NumBids($n+1)
}

BestBid($best-seller, $best-amount)
      & BiddingFinished() {
  (: report offer to client :)
}

NoBids() & BiddingFinished() {
  (: timed out before getting bids from
     sellers - generate fault :)
}
```

This idiom may seem unusual to programmers used to using mutable variables, however the advantage of using internal messages instead of mutable variables is that it is easy to write services that

are correct under concurrent messages. In the example above, because *BestBid* is consumed atomically in conjunction with either a *BidReceived* or a *BiddingFinished*, no bids will be lost. In Java or C# explicit locks would be required to prevent a race condition between comparing a received bid to the best bid and updating the best bid.

Join calculus-style concurrency primitives are a good choice for implementing Web services, compared to the concurrency primitives in Java/C# such as locks and mutexes, because join patterns are declarative, message oriented, and resolve contention locally:

Declarative: Join patterns are high-level and declarative, making it easy to model state machines in services. This makes concurrency aspects of a program simpler to understand. The meaning of code using locks can be occluded by complicated (concurrent!) imperative code.

Message-oriented: Locks are built around atomic access to a shared memory, whereas join patterns take concurrent messages as primitive. This message-orientation parallels message-oriented Web services.

Local: Join patterns resolve contention locally, where messages are bound to patterns. Because the code generated by the compiler to implement join patterns must use the primitive shared memory mechanisms, and these do not scale across slow and unreliable networks such as the internet, this locality property makes join patterns easy to implement.

3.4 Message correlation

Relating different messages into sets is fundamental to realising service conversations. Message correlation is what separates the messages related to one instance of an activity (say, the purchase of a particular item) from messages related to another instance of an activity. Sets of messages can be correlated via identifiers in the message headers. Whether messages contain correlation identifiers is not specified in WSDL, and we carry this into our language: Interface declarations do not mention correlation identifiers, and the programmer can associate correlation identifiers with messages in a fairly ad-hoc way.

Particular correlation identifiers can be bound to variables. By binding many correlation identifiers within a single lexical scope, the programmer can implement interactions that depend on different, concurrent conversations. This supports situations where a Web service conducts multiple, related conversations, but it is not possible or appropriate to share a correlation identifier.

For example, in the scenario outlined in Section 2, the broker will initiate conversations with many sellers on the client's behalf. The correlation identifier the broker shares with the client should not be reused in the (concurrent) conversations with the sellers because it would not uniquely identify them. Instead, we bind two kinds of correlation identifiers in nested lexical scopes using the *with* statement. The nested lexical scopes imply the correlation identifiers exist in a $1 : n$ relationship (in this example, one broker interacts with multiple sellers):

```
with cb as ClientBroker {
  Request($item, in cb) {
    for $seller in $known-sellers
    with bs as BrokerSeller {
      do {
        (Seller at $seller).RequestBid($item,
                           reply-to = ..., in bs)
        ...
      }
      ...
```

Correlation identifiers are bound lazily as messages are received or sent. In the above example when a message is received in *Request* (part of the interface of the *Broker* service) the correlation identifier in the message is extracted and bound to *cb*. The broker then sends messages to sellers. Because *bs* is unbound when the broker sends the message to *RequestBid*, a correlation identifier is created and bound as the message is sent.

ClientBroker and *BrokerSeller* (declaration not shown) specify the location of correlation identifiers in the SOAP header; these use a subset of XPath that contains enough information for the service to synthesize new identifiers when the service needs to initiate a set of correlated messages.

4 Related work

We now survey related work in our language's key feature areas of XML data integration, Web service standards support, concurrency, and message correlation.

4.1 XML Data

Our approach of embracing the XQuery data model is in contrast to practice with Java and C#, which define mappings from XML Schema to native Java or Common Type System types. The benefit of translating XML data to native objects is to provide the programmer with a homogeneous environment. However these mappings can be quite subtle, making it difficult to predict what XML will be synthesised from a particular object. These subtleties can mean Web services are simple to use in a homogeneous implementation environment where native types can effectively be shared (even if this is via an XML Schema representation), but create barriers to interoperation in a heterogeneous environment where different choices are made in doing the mapping. This dependency on implementation environments is contrary to the basic premise of Web services.

Xen (Meijer, Schulte & Bierman 2003*a*, Meijer, Schulte & Bierman 2003*b*, Bierman, Meijer & Schulte 2004) and its successor Cω (Microsoft Research 2004) take a 'superset' approach by supporting XML data and objects with simplified XML types and modest extensions to object types. Making XML data first-class avoids the problem of the opaque mapping mechanism, and the programmer has a literal view of the XML data instead. The disadvantage of this approach is added complexity of having two disjoint data models (XML data and objects) however language features smooth the transition. For example, Xen supports XPath-like expressions over objects and accessing elements of XML data the same way as fields of an object.

Our approach is to reject objects and concentrate on XML data. In this respect, our approach is similar to that of XL (Florescu, Grünhagen & Kossmann 2002, Florescu, Grünhagen & Kossmann 2003), which also leverages XQuery. However, our integration is more austere, with no extensions to XQuery and an emphasis on querying and synthesising XML data, instead of a model of updating XML data 'in place' as in XL's `UPDATE` clause.

4.2 Web service standards

Our language features are designed to be compatible with emerging Web service standards, particularly

SOAP 1.2 (Gudgin, Hadley, Mendelsohn, Moreau & Nielsen 2003) and WSDL 2 (Chinnici, Gudgin, Moreau, Schlimmer & Weerawarana 2004, Gudgin et al. 2004). In some areas our language supports a subset of a standard. For example, interface definitions written in our language are not as descriptive as interface definitions written in WSDL. Several options may be considered to integrate the proposed programming language within infrastructures supporting WSDL. For example, from an interface definition in our language, it is conceivable to output a partial WSDL interface definition that the programmer can then specialize. Another option would be to support metadata annotations in our language that programmers can use to add details necessary to generate full WSDL interface definitions. On the other hand, it is conceivable to generate stubs in our programming language from WSDL interface definitions. The same remarks apply to BPEL abstract processes which can be seen as WSDL interfaces augmented with behavioural aspects. The integration of our language features with Web services standards is discussed further in Section 5.

4.3 Concurrency

The approach to concurrency that we adopt is based on join calculus (Fournet & Gonthier 1996). It differs substantially from the approach followed by proposed standards for Web service composition such as BPEL. Indeed, BPEL (as well as other related proposals) integrates constructs inspired from process algebra (sequence, parallel blocks, conditional blocks, and block-based loops) and other control-flow constructs such as guarded transitions and block-based exception handling. This mix of features can be explained by the fact that BPEL inherits its design from business process modelling and workflow languages, where the basic building block is the task and emphasis is placed on capturing interdependencies between tasks. In our proposed programming language on the other hand, emphasis is on message production and consumption which we take as the fundamental building blocks for service interactions.

Join calculus concurrency features can be found in several proposed programming languages. Our compiler targets a modern virtual machine, and Join Java (Itzstein & Kearney 2002) and Polyphonic C# (Benton, Cardelli & Fournet 2002) demonstrate that join patterns are implementable in this setting. However neither Join Java and Polyphonic C# implement pattern matching, required for our correlation feature (see Section 5). In general, Web service message passing, as opposed to local method calls, make different implementation techniques appropriate in our situation.

4.4 Message correlation

The XL language (Florescu et al. 2002, Florescu et al. 2003) has explicit support for message correlation in conversations. The most important difference between the support we have designed for message correlation, and the support in XL, is that our design flexibly supports *multiple, different* correlation identifiers (and hence conversations) concurrently. The auction example in XL (Florescu et al. 2002, §5) enlists all participants with a single correlation identifier per-auction. As we have highlighted in Section 3.4, this is sometimes undesirable or impossible.

Specifically, the tradeoffs between XL and our programming language with respect to message correlation can be summarised as follows.

- In XL the programmer uses annotations, similar to those used to enlist components in transactions, to specify whether a message should contain a correlation identifier or whether a correlation identifier should be created for an outgoing message. Our scheme of lazily binding correlation identifiers, including on outgoing messages, makes implementing services simpler because there is no need for annotations. However XL supports some annotations that assert particular correlation identifiers *will not* be present and can generate faults automatically. A programmer working in our language would have to explicitly code this behaviour if it is desired.

- XL explicitly calls out certain variables as having conversation-scope. This is analogous to instance and static variables in a type definition in Java and C#. In contrast, we appeal to nested lexical scopes and up-level addressing. Nested lexical scopes work well with correlation identifiers, related by a service, that exist in $1 : n$ relationships. The superiority of our approach to handling $1 : n$ conversations over that of XL is illustrated by the example of Section 3.4. The code snippet in this section cannot be translated to XL in a simple and direct way. The reason is that XL's conversation variables are declared in flat, one-level statements. It would be possible to build the kind of nested conversations shown in Section 3.4 by mutating "service level" variables in XL. However, this encoding would require the use of transactional XQuery features, which does not directly capture the simple nature of the example. Note that $1 : n$ conversations are highly relevant in enterprise application development, which is one of the main application areas of Web services. For example, (Hohpe & Woolf 2003) identifies $1 : n$ message correlation as an important "pattern" of application integration (called the "Scatter-Gatherer") and shows the difficulty of capturing this pattern using contemporary programming languages and messaging systems.

- XL supports an explicit conversation timeout feature. A programmer working in our language would have to program a timeout explicitly. This involves modelling a simple state machine (not timed out/timed out) using internal operations.

5 Conclusion

We have highlighted challenging issues in implementing Web services and, acknowledging that service development can not be driven purely by high-level models, illustrated several novel programming language features that support the implementation of Web services. We have argued that these features improve over previous related proposals in the key areas of XML data, concurrency and message correlation.

Future directions We are implementing a compiler for our language design. The compiler currently only supports a subset of the features outlined here. The compiler produces binary components that run on the the .NET CLR. These components can be deployed as Web services with a minimal host program written in C#. For low-level Web services protocol support, the components bind to the Microsoft Web Services Extensions (WSE) library. The concurrency feature is implemented in terms of the shared-memory concurrency primitives of the CLR.

Two interesting implementation aspects yet to be explored are compiling correlated messages and

garbage collection. To compile correlated messages we plan to add pattern matching to our join mechanism, and desugar correlated messages to use join patterns and pattern matching. Garbage collection might be able to be lifted into our environment's garbage collector. However the compiler can detect 'dead' states in which closures can not activate any more messages and hence can be garbage collected, and emit information for a specialised garbage collector. Note that, by virtue of the locality property of the join calculus concurrency feature, this is a local and not a distributed garbage collector.

Some aspects of the language we will investigate later include the semantics of assignment under concurrent messages, and failures. A simple scheme is to capture copies of variables from containing scopes and encourage the programmer to use explicit messages where shared variables are required. Propagating failures to clients correctly is potentially complex and interesting. We plan to use WSDL's model of failure propagation as a starting point.

With respect to the integration of the proposed language with Web services standards such as XML Schema, WSDL, BPEL and WS-CDL, we intend to explore several alternatives for generating code or checking the consistency between services implemented in our language and types, interfaces, and processes defined in these standards. As discussed in Section 4, it is conceivable to generate (partial) WSDL interface definitions from service interfaces defined in our language, or vice-versa. Also, there are interesting relationships between the message correlation feature and XML Schema types: XML Schema types define cardinality constraints on data elements that may subsequently relate to cardinalities in message patterns (for example if a program iterates over several XML elements and sends a message for each element). Finally, consistency checks may be performed between collections of join patterns on the one hand, and BPEL abstract process definitions on the other. Given the complexity of the languages in question, and the potentially complex interplay between their features, we are not overly optimistic of being able to perform extensive static checks. However dynamic checks may be feasible.

References

Aalst, W. (2003), 'Don't go with the flow: Web services composition standards exposed', *IEEE Intelligent Systems* **18**.

Alonso, G., Casati, F., Kuno, H. & Machiraju, V. (2003), *Web Services: Concepts, Architectures and Applications*, Springer Verlag.

Andrews, T., Curbera, P., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I. & Weerawarana, S. (2003), 'Business process execution language for Web services version 1.1'. *`http://www-106.ibm.com/developerworks/webservices/library/ws-bpel`

Benatallah, B., Sheng, Q. & Dumas, M. (2003), 'The Self-Serv environment for Web services composition', *IEEE Internet Computing* **7**(1), 40–48.

Benton, N., Cardelli, L. & Fournet, C. (2002), Modern concurrency abstractions for C#, *in* B. Magnusson, ed., 'Proceedings of the 16th European Conference on Object Oriented Programming (ECOOP), Malaga, Spain, 10–14 June 2002', Vol. 2374 of *Lecture Notes in Computer Science*, Springer, pp. 415–440.

Bierman, G., Meijer, E. & Schulte, W. (2004), The Essence of Xen. Unpublished paper.

Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D. & Rowley, M. (2004), 'BPELJ: BPEL for Java', White paper. *`ftp://www6.software.ibm.com/software/developer/library/ws-bpelj.pdf`

Boad, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J. & Siméon, J. (2003), 'XQuery 1.0: An XML query language', W3C Working Draft. *http://www.w3.org/TR/2003/WD-xquery-20031112/

Casati, F. & Shan, M.-C. (2001), 'Dynamic and adaptive composition of e-services', *Information Systems* **26**(3), 143–162.

Chinnici, R., Gudgin, M., Moreau, J.-J., Schlimmer, J. & Weerawarana, S. (2004), 'Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language', W3C Working Draft. *`http://www.w3.org/TR/2004/WD-wsdl20-20040326`

Florescu, D., Grünhagen, A. & Kossmann, D. (2002), XL: an XML programming language for Web service specification and composition, *in* 'International World Wide Web Conference', Honolulu, HI, USA.

Florescu, D., Grünhagen, A. & Kossmann, D. (2003), XL: A platform for Web services, *in* 'Conference on Innovative Data Systems Research (CIDR)', Asilomar, CA, USA.

Fournet, C. & Gonthier, G. (1996), The reflexive chemical abstract machine and the join calculus, *in* 'The 23rd ACM Symposium on Principles of Programming Languages (POPL)', pp. 372–385.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J. & Nielsen, H. F. (2003), 'SOAP Version 1.2 Part 1: Messaging Framework', W3C Recommendation. *`http://www.w3.org/TR/2003/REC-soap12-part1-20030624/`

Gudgin, M., Lewis, A. & Schlimmer, J. (2004), 'Web Services Description Language (WSDL) Version 2.0 Part 2: Message Exchange Patterns', W3C Working Draft. *`http://www.w3.org/TR/2004/WD-wsdl20-patterns-20040326`

Hohpe, G. & Woolf, B. (2003), *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison Wesley Professional.

Itzstein, G. S. & Kearney, D. (2002), Applications of Join Java, *in* F. Lai & J. Morris, eds, 'Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002)', ACS, Melbourne, Australia. *`http://citeseer.ist.psu.edu/532429.html`

Java Community Process (2003), 'JSR 207: Process Definition for Java', `http://www.jcp.org/en/jsr/detail?id=207`.

Meijer, E., Schulte, W. & Bierman, G. (2003*a*), Programming with circles, triangles and rectangles, *in* 'Proceedings of the XML Conference, Philadelphia, United States, 7–12 December 2003'.

Meijer, E., Schulte, W. & Bierman, G. (2003*b*), Unifying tables, objects and documents, *in* 'Proceedings of Declarative Programming in the Context of OO-Langauges (DP-COOL), Uppsala, Sweden, 25 August 2003'.

Microsoft Research (2004), 'Comega'. *`http://research.microsoft.com/Comega/`

Szyperski, C. (2003), Component technology - what, where, and how?, *in* 'Proceedings of the 25th International Conference on Software Engineering (ICSE)', IEEE, Portland, OR, USA, pp. 684–693.

Wiederhold, G., Wegner, P. & Ceri., S. (1992), 'Toward megaprogramming', *Communications of the ACM* **35**(11), 89–99.