GPSL: A Programming Language for Service Implementation

Dominic Cooney, Marlon Dumas, and Paul Roe

Queensland University of Technology, Australia
{d.cooney, m.dumas, p.roe}@qut.edu.au

Abstract. At present, there is a dichotomy of approaches to supporting web service implementation: extending mainstream programming languages with libraries and metadata notations vs. designing new languages. While the former approach has proven suitable for interconnecting services on a simple point-to-point fashion, it turns to be unsuitable for coding concurrent, multi-party, and interrelated interactions requiring extensive XML manipulation. As a result, various web service programming languages have been proposed, most notably (WS-)BPEL. However, these languages still do not meet the needs of highly concurrent and dynamic interactions due to their bias towards statically-bounded concurrency. In this paper we introduce a new web service programming language with a set of features designed to address this gap. We describe the implementations in this language of non-trivial scenarios of service interaction and contrast them to the corresponding BPEL implementations. We also define a formal semantics for the language by translation to the join calculus. A compiler for the language has been implemented based on this semantics.

1 Introduction

There is an increasing acceptance of Service-Oriented Architectures as a paradigm for software application integration. In this paradigm, independently developed and operated applications are exposed as (web) services that are then interconnected using standard protocols and languages [1]. While the technology for developing basic services and interconnecting them on a point-to-point basis has attained some maturity, there remain open challenges when it comes to implementing service interactions that go beyond simple sequences of requests and responses or that involve many participants.

A number of recent and ongoing initiatives aim at tackling these challenges. These initiatives can be classified into conservative extensions to mainstream programming languages and novel service-oriented programming languages. The former provide metadata-based extensions for web service development on top of object-oriented programming languages. For example Microsoft Web Services Extensions, Windows Communication Foundation, Apache Axis and JSR-181, can be placed in this category. While these extensions are suitable for dealing with bilateral interactions and simple forms of concurrency and correlation, capturing complex interactions with these libraries remains daunting. On the other hand, a number of service-oriented languages have been proposed, ranging from research proposals (e.g. XL [2, 3]) down to standardisation initiatives, most notably the Business Process Execution Language for Web Services (BPEL) [4].

BPEL facilitates the development of services that engage in concurrent interactions and incorporates a declarative correlation mechanism, thus addressing some limitations of bespoke conservative language extensions. Nonetheless, it fails to provide direct support for typical service interaction scenarios. In [5], a number of patterns of service interaction are proposed. It is shown that while BPEL directly supports the most basic of these patterns, it fails to address the needs of more complex scenarios. In particular, BPEL has problems dealing with one-to-many interaction scenarios with partial synchronisation especially when the set of partner services is not known in advance.

The analysis of BPEL in [5] suggests that web service implementation requires novel programming abstractions for dealing with advanced forms of concurrency, synchronisation, and message correlation. Accordingly, this paper presents a programming language, Gardens Point Service Language (GPSL), that integrates concepts and constructs from join calculus [6], a declarative correlation mechanism with greater flexibility than BPEL's one, and direct support for complex XML data manipulation. Specifically, GPSL incorporates:

– Dedicated messaging constructs, both for interacting with the other services via SOAP, and for structuring the internal implementation of services
– A stratified integration of XQuery [7] expressions with imperative constructs.
– A join calculus-style approach to concurrent 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 [8].

A compiler implementation of GPSL can be found in [9]. The suitability of GPSL has been tested by implementing a number of scenarios, ranging from simple scenarios (e.g. an Amazon.com Queue Service client [10]) to scenarios corresponding to the more complicated service interaction patterns of [5]. In this paper, we sketch the implementations of three of these patterns.

The paper is structured as follows: Section 2 provides an overview of GPSL. Next, Section 3 describes the abstract syntax and formal semantics of GPSL. Section 4 illustrates how the language supports advanced service interaction patterns. Section 5 then briefly describes the compiler implementation of GPSL focusing on the code generation. Finally, Section 6 reviews related work while Section 7 concludes.

2 Overview of GPSL

To illustrate the basic features of GPSL, we consider the implementation of a simple 'echo' service and its client:

```
declare interface Echo {
  declare operation Shout in action = 'urn:echo:shout' out
}
declare service EchoService implements Echo {
  Shout($doc, Reply) { Reply($doc) }
}
declare service EchoClient {
  do {
    let $x := 'soap.tcp://localhost:4000/echo' in
    $x: Shout(element Say { 'Hello' }, Done)
  }
  Done($doc) { (: comment -- do nothing :) }
}
```

GPSL has explicit contract and service declaration elements. Metadata from contracts are used by the compiler to provide types to operations. For example, the *Echo* contract has one operation, *Shout. Shout* is declared as an *in-out* operation and by convention in GPSL has two parameters: one for data, and the other for a channel to send the reply on. When a *Shout* operation message is received, in the case of *EchoService,* or sent, in the case of the *EchoClient,* the first parameter is bound to the body of the SOAP envelope and the second parameter is bound to the WS-Addressing (WS-A) reply-to SOAP header.

EchoService declares *implements Echo* and includes a block guarded by a label *Shout* that takes two parameters. The *Shout* label refers to an operation in the *Echo* contract, so whenever the service receives a message with SOAP action *urn:echo:shout* the service executes the corresponding block of code. The language enforces a convention where variables bound to XML data are prefixed with a *$.* The *$doc* parameter is bound to the body of the SOAP message and the *Reply* parameter is bound to the WS-A reply-to header. *Reply,* although derived from XML in the SOAP envelope, describes the capability for sending a message and we do not prefix it with *$. Reply* is opaque and the capability can only be passed to another service or exercised to send a message. The syntax for sending a message is to write the channel variable and a parameter list in parentheses. In this example, *EchoService* sends in the reply the data it received in the request.

The data model of GPSL has two kinds of values: XML data, such as the element *Say,* and channels, such as *Reply.* All XML expressions in GPSL are XQuery expressions. For example, *element Say* is an example of the XQuery computed element constructor. This ability to construct new XML data distinguishes XQuery from the less powerful XPath. However XQuery alone is not sufficient for implementing services because it is a pure functional language with no messaging constructs. Moreover, there are some semantic tensions between XQuery's flexible evaluation semantics and messaging, because it is difficult to determine when a message will be sent or received. To avoid these tensions, GPSL is based on a stratified approach in which imperative constructs are used for messaging whereas XQuery is used for expressions.

Now let us consider the implementation of *EchoClient.* It contains a *do* block; *do* blocks are executed when a service starts up and can initialise state like a constructor in an object-oriented language. Here *EchoClient* sends a *Shout* message. For the first parameter it constructs an element *Say* that contains the text *Hello.* By convention, the second parameter becomes the WS-A reply-to header. Here *EchoClient* provides the label of a block, *Done,* as the second parameter. *Done* is a "private" label of *EchoClient,* and does not refer to any operation in a contract.

Messaging in GPSL is asynchronous; this encourages programmers to write services that make concurrent requests rather than sequences of request/responses, although this RPC programming style is also possible in GPSL. GPSL's means of spawning concurrent threads derives from asynchronous messaging. Using a private label to send an internal message starts the corresponding block of code which is executed concurrently with subsequent instructions.

```
    ...
    M(); (: sends local message, asynchronously :)
    ...  (: subsequent instructions go here :)
  }

  M() {
    (: this code executes concurrently when a message is sent on M :)
  }
```

Synchronisation is achieved through blocks of code guarded by multiple labels. Such multi-label guards are called *concurrency patterns* and are inspired by the join calculus. A block of code guarded by a concurrency pattern is executed when messages are available on all labels. For example, in the following code snippet, local messages *ResultA* and *ResultB* are sent in two different blocks of code $A()$ and $B()$ which we assume are executed concurrently (although their spawning is not shown). When both messages are available, then the rule at the bottom is reduced and the corresponding block of code is executed.

```
  A() {
    ...
    ResultA(...) (: produce message ResultA :)
  }
  B() {
    ...
    ResultB(...) (: produce message ResultB :)
  }
  ResultA($a) & ResultB($b) (* this is a join pattern *) {
    (: executed when ResultA and ResultB are available :)
  }
```

3 Syntax and Semantics

The syntax of GPSL statements and expressions is shown in Figure 1. For space reasons we focus on statements and expressions omitting the *service* and *contract*

$S,\ T ::=$		statement
	ϵ	empty
	S **;** T	sequence
	if E **then** S **else** T **end**	conditional
	let v **:=** E **in** S	let-binding
	for v **in** E **do** S **end**	iteration
	$E(G, [rc])$	send (2nd argument may be used for reply channel)
	E **:** $m(G, \cdots)$	endpoint send
	def D **in** S	receive rules
$D,\ F ::=$		definitions
	J **{** S **}**	receive rule
	$D\ F$	composition
$J,\ K ::=$		pattern
	$x(y, \cdots)$	internal message receive
	receive y **where** E	external message receive
	$x(y)$[**where** E]	contract receive: x is an "in" operation defined in a contract
	$x(y, rc)$[**where** E]	contract receive: x is an "in-out" operation defined in a contract (rc stands for "reply channel")
	J **&** K	synchronisation
$E,\ G ::=$		expression
	m	label
	$\cdots$	XQuery expression

Where $m,\ v,\ x,\ y$ and rc are identifiers.

Fig. 1. Abstract syntax of the imperative GPSL statements

elements; these elements provide metadata about the SOAP action and message exchange patterns of operations and do not have a direct operational semantics.

We sketch the semantics of GPSL in Figure 2 via an operational encoding in the join calculus [6]. Since GPSL's concurrency feature is based directly on the join calculus this encoding is often straightforward syntax translation.

For our encoding we assume a join calculus with XQuery expressions and values. Where XQuery has flexible evaluation semantics related to laziness/strictness and raising errors, GPSL needs predictable behaviour for message sending. We introduce an explicit channel, *eval,* to specify precisely when XQuery evaluation occurs. *eval* forces XQuery evaulation in its first argument and passes the result on its second argument. *cond,* for implementing conditionals, is like *eval* except it chooses a continuation based on the result.

For sending messages on internal channels (rule "Int. Snd" in Figure 2) we only give the encoding of the single-argument case. Other arities follow the same pattern, where the message receiver and arguments are evaluated left-to-right. Likewise, for sending messages to other services (Ext. Snd,) we only give the case when the operation is expected to reply, where by convention in GPSL the first argument becomes the body of the message, and the second argument is

$$
\begin{array}{lll}
\text{Empty:} & \llbracket \epsilon \rrbracket & \rightarrow 0 \\
\text{Seq:} & \llbracket S;T \rrbracket & \rightarrow \llbracket S \rrbracket . \llbracket T \rrbracket \\
\text{If:} & \llbracket \textbf{if } E \textbf{ then } S \textbf{ else } T \textbf{ end} \rrbracket & \rightarrow \textbf{def } t\langle\rangle \rhd \llbracket S \rrbracket \mid f\langle\rangle \rhd \llbracket T \rrbracket \textbf{ in } cond\langle E, t, f\rangle \\
\text{Let:} & \llbracket \textbf{let } v := E \textbf{ in } S \rrbracket & \rightarrow \textbf{def } s\langle v\rangle \rhd \llbracket S \rrbracket \textbf{ in } eval\langle E, s\rangle \\
\text{For:} & \llbracket \textbf{for } v \textbf{ in } E \textbf{ do } S \textbf{ end} \rrbracket & \rightarrow \begin{array}[t]{l}
\textbf{def } test\langle es, s\rangle \rhd \\
\quad \textbf{def } t\langle\rangle \rhd \\
\quad\quad \textbf{def } hd\langle e\rangle \rhd \\
\quad\quad\quad \textbf{def } tl\langle es\rangle \rhd s\langle e, es\rangle \textbf{ in} \\
\quad\quad\quad eval\langle es[position() > 1], tl\rangle \textbf{ in} \\
\quad\quad eval\langle es[position() = 1], hd\rangle \textbf{ in} \\
\quad \textbf{def } f\langle\rangle \rhd 0 \textbf{ in} \\
\quad cond\langle es = nil(), t, f\rangle \textbf{ in} \\
\textbf{def } s\langle v, es\rangle \rhd \llbracket S \rrbracket . test\langle es, s\rangle \textbf{ in} \\
\textbf{def } init\langle es\rangle \rhd test\langle es, s\rangle \textbf{ in} \\
eval\langle E, init\rangle
\end{array} \\
\text{Int. Snd:} & \llbracket E(G) \rrbracket & \rightarrow \begin{array}[t]{l}
\textbf{def } receiver\langle e\rangle \rhd \\
\quad \textbf{def } actual_1\langle f\rangle \rhd e\langle f\rangle \textbf{ in} \\
\quad eval\langle G, actual_1\rangle \textbf{ in} \\
eval\langle E, receiver\rangle
\end{array} \\
\text{Ext. Snd:} & \llbracket E{:}m(G, H) \rrbracket & \rightarrow \llbracket \begin{array}[t]{l}
\textbf{let } receiver := E \textbf{ in} \\
\textbf{let } actual := G \textbf{ in} \\
\textbf{let } reply := H \textbf{ in} \\
\textbf{let } id := gensym \textbf{ in} \\
\textbf{def receive } env \\
\quad \textbf{where } Header/RelatesTo = id \; \{ \\
\quad reply(env) \\
\} \textbf{ in} \\
send(receiver, m_{action}, id, actual) \rrbracket
\end{array} \\
\text{Recv:} & \llbracket \textbf{def } D \textbf{ in } S \rrbracket & \rightarrow \textbf{def } \llbracket D \rrbracket \textbf{ in } \llbracket D \rrbracket_{Init} . \llbracket S \rrbracket \\
\text{Reaction:} & \llbracket J\{S\} \rrbracket & \rightarrow \llbracket J \rrbracket \rhd \llbracket S \rrbracket \\
\text{Composition:} & \llbracket D \; F \rrbracket & \rightarrow D \wedge F \\
\text{Synch:} & \llbracket D \;\&\; F \rrbracket & \rightarrow D | F \\
\text{Int. Recv:} & \llbracket x(y) \rrbracket & \rightarrow x\langle y\rangle \\
\text{Ext. Recv:} & \llbracket \textbf{receive } y \textbf{ where } E \rrbracket & \rightarrow x\langle y\rangle, x \text{ is fresh} \\
\text{Contract Recv:} & \llbracket x(y) \textbf{ where } E \rrbracket & \rightarrow x\langle y\rangle \\
\text{Init Reaction:} & \llbracket J\{S\} \rrbracket_{Init} & \rightarrow \llbracket J \rrbracket_{Init} \\
\text{Init Comp.:} & \llbracket D \; F \rrbracket_{Init} & \rightarrow \llbracket D \rrbracket_{Init} . \llbracket F \rrbracket_{Init} \\
\text{Init Synch:} & \llbracket D \;\&\; F \rrbracket_{Init} & \rightarrow \llbracket D \rrbracket_{Init} . \llbracket F \rrbracket_{Init} \\
\text{Init Int. Recv:} & \llbracket x(y) \rrbracket_{Init} & \rightarrow \mathbf{0} \\
\text{Init Ext Recv:} & \llbracket \textbf{receive } y \textbf{ where } E \rrbracket_{Init} & \rightarrow \begin{array}[t]{l}
\textbf{def } x_{test}\langle y, t, f\rangle \rhd cond\langle E, t, f\rangle \textbf{ in} \\
subscribe\langle x, x_{test}\rangle
\end{array} \\
\text{Init Contract Recv:} & \llbracket x(y) \textbf{ where } E \rrbracket_{Init} & \rightarrow \begin{array}[t]{l}
\llbracket \textbf{def receive } env \\
\quad \textbf{where } Header/Action = x_{action} \textbf{ and} \\
\quad\quad E \; \{ \; x(env) \; \} \textbf{ in} \\
\cdots \rrbracket, \text{ for the first occurence of } x(y) \textbf{ where } E
\end{array}
\end{array}
$$

Fig. 2. Partial semantics by translation into join calculus

a channel to use for replies. It is the metadata from a contract element that dictates whether a reply is expected and the SOAP action m_{action}. Correlating replies involves generating a new message ID, establishing a closure to listen for incoming messages with a matching message ID, and then sending the message. We write *send* for this latter step; *send* formats a SOAP envelope and sends it over the network.

Definitions and patterns follow predictable syntactic translation, except for external message receive which has no parallel in the join calculus. External message receive is responsible for marshalling SOAP messages received from the outside world into a GPSL program. This raises the important semantical issue of precisely what point in a program messages are delivered *to*. GPSL is more flexible and powerful than most contemporary programming languages in that it supports a *where* clause for filtering incoming messages. This feature is akin to filtering capabilities in message-oriented middleware and enables, among other things, to correlate any sent or received message with follow-up messages.

To encode external message receives, we create fresh internal channels and bind them to the SOAP messaging machinery via a message to *subscribe* when the closure is created. *subscribe* is a global internal channel with a complete join-calculus definition in Figure 3. This gives a precise semantics to receiving messages in GPSL: there is no race condition between receiving and sending messages in a closure as a closure is created, because of the continuation k threaded through *subscribe*, which is important for the correctness of closures initiating conversations; messages are routed into matching closures; concurrently active *receive* statements cause a runtime error if they compete for a particular message; and messages that have no active *receive* to process them are silently dropped.

There is syntactic sugar for receiving messages from an operation of an implemented contract (Init Ctrct Recv) which includes a test against the SOAP action specified in the contract. Our translation omits one detail in that the *receive* clause constructed for an *in-out* operation also creates a channel carrying the reply. The translation in Figure 2 is for an *in* operation.

```
**def** $subscribe\langle msg, predicate, k\rangle | subscribers\langle f\rangle \rhd$
    **def** $g\langle x, found, done\rangle \rhd$
        **def** $true\langle\rangle \rhd found\langle msg, f\rangle$ **in**
        **def** $false\langle\rangle \rhd f\langle x, found, done\rangle$ **in**
        $predicate\langle x, true, false\rangle$ **in**
    $subscribers\langle g\rangle . k\langle\rangle$
$\wedge$  $external\langle env\rangle | subscribers\langle f\rangle \rhd$
        $subscribers\langle f\rangle |$
        **def** $done\langle\rangle | single\langle msg\rangle \rhd msg\langle env\rangle$ **in**
        **def** $fail\langle msg, k\rangle \rhd$ error **in**
        **def** $found\langle msg, k\rangle \rhd single\langle msg\rangle . k\langle env, fail, done\rangle$ **in**
        $f\langle env, found, done\rangle$ **in**
**def** $nil\langle x, found, done\rangle \rhd done\langle\rangle$ **in**
$subscribers\langle nil\rangle$
```

Fig. 3. Join-calculus definition of *subscribe*

4 Service Interaction Patterns in GPSL

In this section, we compare GPSL with BPEL by implementing scenarios corresponding to two of the service interaction patterns of [5]. Using the nomenclature and numbering of [5] we have chosen: one-to-many send/receive (pattern 7), and contingent requests (pattern 8). We choose not to illustrate patterns 1 to 4 since they correspond to simple point-to-point interactions and do not put forward significant differences between GPSL and other service-oriented programming languages such as BPEL; patterns 5 and 6 are partly subsumed by pattern 7; pattern 9 makes appeal to similar features as patterns 4 and 7; pattern 10 deals with transactional issues beyond the scope of GPSL and BPEL; and patterns 11 through 13 deal with interconnecting groups of services rather than implementing individual ones.

4.1 One-to-Many Send-Receive

We consider an interaction pattern where a service sends messages and collects responses before continuing. In this example we implement a broker service that solicits bids from a set of bidders, and collects responses, keeping track of the best (in this example, lowest) bid received. Bids are collected until a time-out occurs.

```
declare interface BrokerContract {
  declare operation InitiateAuction in action = 'urn:broker:init';
  ...
}

declare service Broker implements BrokerContract {
  InitiateAuction($env) {
    (: solicit bids :)
    for $bidder in $env/Bidders do
        $bidder: SolicitBid($env/Item, Reply)
    done;
    OutstandingBids(util:length($env/Bidders));

    (: start timer :)
    let $timeout := 'soap.inproc://timer' in
    $timeout: Time(10000, TimedOut);

    NoBids()
  }
  OutstandingBids($n) & Reply($bid) & NoBids() {
    Winning($bid);
    Decrement($n)
  }
  OutstandingBids($n) & Reply($bid) & Winning($best) {
    if xs:decimal($bid/Amount) < xs:decimal($best/Amount) then
      Winning($bid)
    else
```

```
      Winning($best)
    end;
    Decrement($n)
  }
  Decrement($n) {
    let $n := xs:int($n) - 1 in
    if xs:int($n) = 0 then
      BiddingFinished()
    else
      OutstandingBids($n)
    end
  }
  BiddingFinished() & Winning($bid) { (: process winning bid :) }
  TimedOut() & Winning($bid)        { (: fault or process winning bid :) }
  TimedOut() & NoBids()             { (: fault :) }
  ...
}
```

This program sends n *SolicitBid* messages. Although the *for* loop is sequential, the *SolicitBid* messages are sent in a non-blocking manner. The first bid received consumes the *NoBids* message and becomes the winning bid. Subsequent bids are compared to the winning bid. Messages *OutstandingBids* and *WinningBid* are used to capture state. They carry data for the number of outstanding bids and the best bid received. It is possible to check that this service correctly treats concurrent bids because contention for the *WinningBid* message acts as a mutual exclusion.

In previous work [8], we have sketched a more complicated variant of this scenario where the service stops after either receiving the first n-out-of-m responses or after the time-out, whichever occurs first.

Coding the above scenario in BPEL is complicated by several factors. First, given that the set of partners to which bid requests are sent is not known in advance, dynamic addressing is required. In GPSL, this is achieved by treating channels as first-class citizens. In BPEL, dynamic addressing is possible but requires manual assignment of endpoint references to *partner links*. Second, BPEL lacks high-level constructs for manipulating collections. Thus, capturing this scenario requires the use of *while* loops and additional book-keeping. Third, there is no direct support in BPEL for interrupting the execution of a block when a given event (e.g. a timeout) occurs. To achieve this, it is necessary to combine an event handler with a fault handler, such that the event handler raises a fault when the nominated event occurs and the fault handler catches this artificially created fault. This causes the immediately enclosing scope to be stopped. In GPSL, such interruption can be achieved simply by adding a join pattern that matches the event in question (in this case, the timeout). Finally, a further complication arises if explicit correlation using *correlation sets* is necessary. In this case, the first message needs to be treated differently from the following messages (at least in BPEL 1.1) since the first message initialises the correlation set. BPEL pseudo-code for this scenario is given below. The full version of this pseudo-code is considerably longer than the corresponding GPSL solution. The interested reader will find the full BPEL implementation of a similar scenario in the code repository of the service interaction patterns site.[1] This implementation comprises around 150 lines of BPEL code excluding comments, partly due to the verbosity of the XML syntax, but also because of the more fundamental drawbacks of BPEL mentioned above.

```
set partner link to the address of the first bidder;
send first bid request and initialise correlation set;
while (more partners)
   set partner link to the address of next bidder;
   send next bid request;
begin scope
   onAlarm timeLimit : throw timeoutFault
   catch timeoutFault : set flag to indicate time-out
   while (not stopCondition)
      receive a bid;
      update winning bid
end scope
(* process time-out or winning bid *)
```

4.2 Contingent Requests

In this pattern, a service sends a message and if a response is not received within a given timeframe, a message is sent to a second service, and so on. If while waiting for a response from the second service, the first service happens to respond, this response is accepted and the response from the second service is no longer needed. This implements a fail-over process. An example of this pattern is a conference that provides redundant services to accept a paper submission. The client submits the paper via the first service, and if a response is not received within ten seconds, it submits the paper via the second service, and so on. Here is the implementation of the “client” in GPSL.

```
declare variable $timeout := 'soap.inproc://timer';
declare interface PaperSubmission {
  declare operation Submit in action = 'urn:paper:submit' out
}
declare service PaperSubmitter {
  do {
    let $submission-points := element Point { ... }, ... in
    let $paper := ... in
    Submit($paper, $submission-points)
  }
  Submit($paper, $submission-points) {
    let $uri := $submission-points[1] in
    let $submission-points := $submission-points[position()>1] in
```

[1] See code sample “One-to-many send/receive with dynamically determined partners” at http://www.serviceinteraction.com

```
    $uri: Submit($paper, Response);
    $timeout: Time(10000, TimedOut);
    Waiting($paper, $submission-points)
  }
  Waiting($paper, $submission-points) & Response($doc) { (: success! :) }
  Waiting($paper, $submission-points) & TimedOut($doc) {
    (: submit to the next server :)
    Submit($paper, $submission-points)
  }
}
```

The *PaperSubmitter* service has knowledge of a list of services that accept submissions. *Submit* strips a URI from the head of the list and sets up a race between *Response* and *TimedOut* messages. If a *TimedOut* message is available first, *PaperSubmitter* submits to the next service. After a response is received from any of the contacted services, *PaperSubmitter* does not wait for other responses, unless the code in the "success" block issues a new *Waiting* message.

This example shows the *PaperSubmitter* service interacting with a timer service at a known URI through *Time* (an operation to arm the timer) and *TimedOut* (an internal channel that receives time-out messages from the timer). This timer service does not need to be located outside *PaperSubmitter*'s memory space. Our implementation supports an efficient in-process transport for SOAP messaging, namely *soap.inproc*. This way, we can extend the language by adding services implemented in (e.g.) C#, rather than adding new constructs for every required feature. This is similar to a library, except that GSPL's library calling convention is based on messaging rather than function or method calls.

Capturing this example in BPEL in all its details is complicated by two factors: (i) the lack of direct support for interruptions due to an event as discussed previously; and (ii) the lack of support for maintaining an a priori unknown number of conversations in parallel. Indeed, this pattern puts forward a case where a requester may start a new conversation with a partner, but keep another ongoing conversation alive. There is only one construct in BPEL that supports an unbounded number of threads to be entertained concurrently: event handlers. However, using event handlers to capture the scenario at hand leads to an unintuitive solution. In this solution, the code for submitting a paper to a given server is embedded in an event handler. To start this event handler for the first server, the process sends a message to itself. This starts a first instance of the event handler. This event handler is terminated if a response is received (to do so, a fault indicating this is thrown). If a time-out occurs within this first instance of the event handler, the process sends a second message to itself to activate a second instance of the event handler (without stopping the previous instance since a late response from the first server may still arrive). This process of starting new instances of the event handler continues until a response is received or all servers have been tried.

The BPEL pseudo-code for this scenario is given below. Again, the full BPEL code is considerably more verbose, partly due to the need to define and configure the partner link through which the process sends messages to itself. The full

BPEL code for a similar scenario available at the serviceinteraction.com site comprises around 120 lines of code.

```
responseReceived := false
begin scope
  onMessage X :
    begin scope
      onAlarm timeLimit :
         if (more servers) send a message of type X to myself
         else throw allServersTimedOut
      catch allServersTimedOut: do nothing (* terminates scope *)
      catch responseReceived: do nothing (* terminates scope *)
      send request to next server and wait for response;
      responseReceived := true;
      throw responseReceived
    end scope
  send a message of type X to myself
end scope
if (not responseReceived) (* deal with case where no response received *)
```

5 Code Generation

We have prototyped a compiler that produces Microsoft Intermediate Language (MSIL), from GPSL programs. MSIL is similar to Java byte code although differing from it in various respects. Despite these differences though, our prototype proves that the feasibility of compiling GPSL for modern virtual machines.

Despite the novelty in the programming language, the compiler operates in traditional parsing, analysis, and code generation phases. The parser must handle XQuery for expressions. For our prototype we found ignoring XQuery direct constructors—the angle-brackets syntax for synthesizing XML which require special handling of whitespace—greatly simplifies parser development. Because syntactically simpler computed constructors can do the job of direct constructors, the expressive power of XQuery is unimpeded.

Most of the complexity in the compiler is in the code generator, and specifically in the creation of closures and in the delivery of messages sent on internal labels (i.e. messages from a service to itself). For each *def* we create a class with a method for each concurrency rule, a field for each captured variable, and a method and field for each label. This field holds a queue of pending messages; the method takes a message to that label, tests whether any rules are satisfied, and if so, calls the method for the rule. We perform the rule testing on the caller thread and only spawn a thread when a rule is satisfied, which avoids spawning many threads. The rule testing follows the join calculus semantics and the definition for the *subscribe* reaction rule given in Figures 2 and 3 for internal and external messages respectively.

We do not compile XQuery expressions because implementing an XQuery compiler is a daunting task. Instead we generate code to call an external XQuery library at runtime. One critical criterion for the programming language implementer integrating an XQuery implementation is how that XQuery implementation accepts external variables and provides results. GPSL requires access to expression results as a sequence of XQuery data model values—which is distinctly different from an XML document—to behave consistently with XQuery when those values that are used later in subsequent expressions. We use an interoperability layer over the C API of Galax[2], which has exactly the kind of interface for providing external values and examining results that we want. Our biggest complaint about Galax is that evaluating expressions must be serialized because Galax is non-reentrant.

GPSL programs also rely on the Microsoft Web Services Extensions[3] (WSE) for SOAP messaging. WSE has a low-level messaging interface which is sufficient for GPSL's needs except for the fact that WSE does not support SOAP RPC/encoded. In the case of synchronous operations, the GPSL compiler generates some bookkeeping code to make SOAP over synchronous-HTTP work using the WSE messaging interface.

6 Related Work

Mainstream approaches to web service implementation are based on the use of Java, C#, and C++ in conjunction with libraries, such as Axis, and metadata annotations as in JSR181 and Windows Communication Foundation. Putting aside the mismatch between object-oriented and XML-based data manipulation, this approach has proven fairly suitable for programming point-to-point service interactions. However, it does not properly serve the requirements of multilateral interactions, especially those requiring partial synchronisation and message correlation beyond simple "request-response" scenarios. Message passing interfaces, like MPI [11], alleviate some of these issues, but even MPI's scatter and gather primitives assume barrier synchronisation and message correlation requires careful programming.

BPEL departs from mainstream web service implementation approaches by providing an XML data model, a set of message exchange primitives, concurrency constructs inspired from workflow languages, and a message correlation mechanism based on lexically scoped "static" variables. However, while BPEL supports high-level concurrency and barrier synchronisation constructs for fixed numbers of threads (for example through the "flow" construct), it does not support partial synchronisation nor unbounded numbers of threads, and thus, the expression of patterns such as one-to-many send-receive, multi-responses and contingent requests is cumbersome. Also, support for message correlation in BPEL is limited: BPEL's correlation sets can not be used to capture the type of correlation required by the one-to-many send-receive pattern.

Similar comments apply to XL, which provides a correlation mechanism suitable for 1:1 conversations, but not for $1 : n$ scenarios. Similarly, XL is suitable for

[2] http://www.galaxquery.org

[3] http://msdn.microsoft.com/webservices/building/wse

barrier synchronisation of conversations but not for partial synchronisation. Finally, XL relies on in-place updates of XML nodes through extensions to XQuery, while GPSL adopts a stratified approach where XQuery is only used as an expression language, orthogonal to the imperative part of the language. A more detailed comparison of XL and an earlier version of GPSL can be found in [8].

GPSL draws one of its main constructs, concurrency patterns, from the join calculus. The join calculus has inspired several extensions of object-oriented programming languages with concurrency features, namely Join Java [12] and Polyphonic C# [13]. Compared to these languages, GPSL adds XML data manipulation, messaging and message correlation. An extension to Polyphonic C#, $C\omega$[4], adds XML data manipulation, but retains the legacy object data model and does not have explicit support for messaging or message correlation.

7 Conclusion

We have presented the syntax and semantics of the GPSL language and illustrated its suitability for service implementation using scenarios corresponding to patterns identified elsewhere. This exercise showed how simple features based on SOAP messaging, join-calculus style declarative concurrency, and XQuery can be combined to implement non-trivial patterns of service interaction in a way that arguably leads to simpler solutions than in BPEL. Also, GPSL's formal semantics is much simpler than corresponding semantics of BPEL[5] thus providing a solid basis for program analysis. The compiler implementation of GPSL, especially with respect to compiling rules with *where* statements, mirrors the formal semantics.

GPSL integrates messaging, concurrency, and XML data manipulation cohesively. Examples of the cohesive fit are the interplay between sending messages and spawning concurrent threads on the one hand, and receiving messages and synchronising threads on the other; dynamic XML data describing message recipients; concurrency patterns describing thread-safe access to XML data; and the consistent treatment of inter- and intra-service messages. Sometimes the cohesion is imperfect. For example, channels and channel variables can not appear in arbitrary XQuery expressions. This is a deliberate restriction which provides a simple way to preserve strong typing for internal message sending, and to control when an internal channel has to be connected to the machinery for receiving SOAP messages from the outside world. However channels *are* reified to XML when they appear in a WS-A "reply-to" header.

GPSL could be extended to address other difficult aspects of service implementation such as transactions and faults. We expect to address these areas by leveraging the messaging and concurrency features, for example, by surfacing faults as messages. We also plan to introduce a garbage collection technique to reclaim resources when it is detected that a given message will not be consumed.

[4] http://research.microsoft.com/Comega

[5] For a semantics of BPEL see e.g. [14].

Acknowledgments. The first author completed this work while working at Microsoft. The second author is funded by a fellowship co-sponsored by Queensland Government and SAP.

References

1. Weerawarana, S., Curbera, F., Leymann, F., Story, T., Ferguson, D.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable-Messaging, and More. Prentice Hall (2005)
2. Florescu, D., Grünhagen, A., Kossmann, D.: XL: an XML programming language for Web service specification and composition. In: International World Wide Web Conference, Honolulu, HI, USA (2002)
3. Florescu, D., Grünhagen, A., Kossmann, D.: XL: A platform for Web services. In: Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA (2003)
4. Andrews, T., Curbera, P., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for Web services version 1.1 (2003)
5. Barros, A., Dumas, M., Hofstede, A.: Service interaction patterns. In: Proceedings of the 3rd International Conference on Business Process Management, Nancy, France, Springer Verlag (2005) Extended version available at: http://www.serviceinteraction.com.
6. Fournet, C., Gonthier, G.: The reflexive chemical abstract machine and the join calculus. In: The 23rd ACM Symposium on Principles of Programming Languages (POPL). (1996) 372–385
7. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML query language. W3C Working Draft (2005)
8. Cooney, D., Dumas, M., Roe, P.: A programming language for web service development. In Estivill-Castro, V., ed.: Proceedings of the 28th Australasian Computer Science Conference, Newcastle, Australia, Australian Computer Society (2005)
9. Cooney, D.: GPSL home page. WWW (2005) http://www.serviceorientation.com.
10. Cooney, D., Dumas, M., Roe, P.: Programming and compiling web services in GPSL. In: Proceedings of the 3rd International Conference on Service-Oriented Programming (ICSOC) – Short papers track, Amsterdam, The Netherlands, Springer (to appear) (2005)
11. Snir, M., Gropp, W.: MPI: The Complete Reference. 2nd edn. MIT Press (1998)
12. Itzstein, G.S., Kearney, D.: Applications of Join Java. In Lai, F., Morris, J., eds.: Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC2002), Melbourne, Australia, ACS (2002)
13. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. In Magnusson, B., ed.: Proceedings of the 16th European Conference on Object Oriented Programming (ECOOP), Malaga, Spain, 10–14 June 2002. Volume 2374 of Lecture Notes in Computer Science., Springer (2002) 415–440
14. Ouyang, C., Aalst, W., Breutel, S., Dumas, M., Hofstede, A., Verbeek, H.: Formal Semantics and Analysis of Control Flow in WS-BPEL (Revised Version). BPM Center Report BPM-05-13, www.bpmcenter.org (2005)