ASDL: A Wide Spectrum Language For Designing Web Services

Monika Solanki
Software Technology
Research Laboratory
De Montfort University
Leicester, LE1 9BH, UK
monika@dmu.ac.uk

Antonio Cau
Software Technology
Research Laboratory
De Montfort University
Leicester, LE1 9BH, UK
acau@dmu.ac.uk

Hussein Zedan
Software Technology
Research Laboratory
De Montfort University
Leicester, LE1 9BH, UK
zedan@dmu.ac.uk

ABSTRACT

A Service oriented system emerges from composition of services. Dynamically composed reactive Web services form a special class of service oriented system, where the delays associated with communication, unreliability and unavailability of services, and competition for resources from multiple service requesters are dominant concerns. As complexity of services increase, an abstract design language for the specification of services and interaction between them is desired. In this paper, we present ASDL (Abstract Service Design Language), a wide spectrum language for modelling Web services. We initially provide an informal description of our computational model for service oriented systems. We then present ASDL along with its specification oriented semantics defined in Interval Temporal Logic (ITL): a sound formalism for specifying and reasoning about temporal properties of systems. The objective of ASDL is to provide a notation for the design of service composition and interaction protocols at an abstract level.

Categories and Subject Descriptors

F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages; F.3.2 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; F.1.1 [**Computation By Abstract Devices**]: Models of Computation

General Terms

Design, Languages, Theory

Keywords

Abstract, Wide spectrum, Computational model, Web services, ASDL

1. INTRODUCTION

The Internet with its heterogeneous collection of infrastructures, has established itself as the largest universal source of information dissemination. Central to the different architectures and computing environments prevalent on the Internet, is the notion of a "Service". The umbrella term "Ser-

vice" encapsulates software, smart devices and sensors networked with one another, each responsible to achieve some portion of the entire goal. The proliferation of services available on the web coupled with robust networking protocols means that distributed systems are increasingly being designed as compositions of services.

Web services are prominently reactive systems, that repeatedly act and react in interaction with their environment without necessarily terminating. Dynamically composed reactive services form a special class of service oriented system, where the delays associated with communication, unreliability and unavailability of services, and competition for resources from multiple service requesters are dominant concerns. As complexity of services increase, an abstract design language for the specification of services and interaction between them is desired, before the behaviour can be mapped to constructs in XML-based languages such as WSDL [7], WSBPEL [39], WSCDL [27] and OWL-S [38], or implemented using frameworks such as J2EE or .Net as illustrated in Figure 1. The development of such a lan-
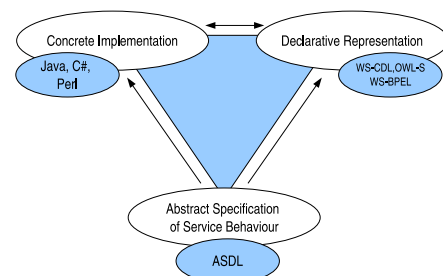

Figure 1: Various Levels of Representing Service Behaviour

guage should be based on a formal model of computation. The language should have sound formal semantics, an associated proof system and should be expressive enough to specify all possible behaviours. Current Standards (XML based/Ontologies) for specifying behavioural semantics of services consider only transformational aspects of service behaviour. They are based on an adhoc and informal model of computation. The semantics are buried in the execution engines that are bundled with distinctly different reference implementations of the language.

We consider the quintessential problem of how to char-

acterise formally, the global behaviours generated by a dynamic Web service composition, as well as how to reason about their correctness. Before addressing these issues, we need to precisely define a computational model which describes the underlying abstract architecture upon which service oriented systems will execute. The behaviour generated within a service oriented system, due to interactions among services, with regards to the exchange of messages and the sequence in which they are transmitted and received forms the basis of our model.

In this paper, we initially provide an informal description of our computational model for service oriented systems. We then present a design language "ASDL" along with its specification oriented semantics in Interval Temporal Logic (ITL), our underlying logical framework for reasoning about service behaviour over periods of time. Our objective is to develop a methodology for the design of service oriented systems, based on a sound model of computation and a compositional technique, that allows specification and validation within a unified semantic model. ASDL complements our computational model and provides a framework complete with algebraic rules for establishing useful properties about service composition.

The paper is organised as follows: Section 2 presents our computational model. Section 3 briefly discusses our underlying formalism ITL. Section 4 presents ASDL: Abstract Service Design Language for Web services. Section 5 discusses related work and Section 6 presents conclusions and directions for future work.

2. COMPUTATIONAL MODEL

A service oriented system is a collection of possibly concurrently executing services, which communicate asynchronously via message passing, using shared bounded buffers [6] called "channels". Service oriented systems can themselves be viewed as single services and composed into larger systems. The first class citizens within a service oriented system are a finite set of "services", "channels" and an infinite set of "messages". The restriction of having a finite number of services and channels is intuitive with respect to service oriented systems. The system state is defined by the values (messages) in the channels and the values in the local variables of the service. **Behaviour** in our model is defined as a sequence of system states. **Computation** is defined as any process that results in a change of system state.

2.1 Service

A service is a black box, in terms of representation of its internal behaviour. A service is described by a computation which may transform a local data space and may read and write messages during execution. Computation may be nondeterministic with timing restrictions. A service can be viewed as a computational entity that decides, based on the messages received, and the messages already sent, if a new message needs to be sent, to which services in the system the message needs to be sent or if the service session needs to terminate. An observable behaviour of a service is a sequence of message exchanges in which the service has engaged up to a particular moment in time. At the lowest level of granularity, we have services that undertake activities like "send" and "receive". These are termed as "Primitive" services. A service can be composed of several such activities, in which case they are termed as "Composite".

2.2 Communication
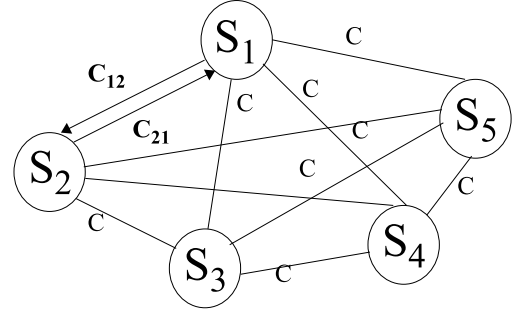
**Figure 2: Channel configuration in a Service-oriented system**

A service-oriented system, is built on the message-oriented-model (MOM) of communication. Message-oriented behaviour modeling is not only intuitive for services, but more importantly, it requires Web services to reveal the least amount of information that is necessary to make meaningful compositions. Thus complex internal states can be hidden. Services communicate using asynchronous message passing via first-in-first-out (FIFO), error-free channels as illustrated in Figure 2. Channels are uni-directional and can be used both to read and write messages. Every system has a finite number of channels, shared between exactly two **distinct** services. A channel is a bounded buffer where a message can be inserted at one end and retrieved sequentially at the other. Sending a message is equivalent to appending it to the tail of the sequence; receiving a message, to removing the head of the sequence. Sending is allowed only if there are empty places available in the buffer. A service cannot send a message back to itself, as it is counter-intuitive to do so.

Every send activity propagates the message to the last empty position in the sequence. Receiving is destructive, as it removes an element from the head of the buffer. The message to be sent through the channel is modelled as a triple. The elements of the triple are the message identifier, the data to be transmitted and the time-stamp, specifying the time of message transmission with reference to a global clock. Messages are tagged with unique identifiers for correlating between the sent and received messages. Identifiers are unique across complete transactions between services. At an abstract level, message identifiers are modelled using natural numbers. At an implementation level, identifiers can be modelled as sets of data having multiple types. We do not distinguish between input, output and fault messages in our model i.e. all messages are treated at a uniform level of abstraction.

2.3 Time

Service-oriented systems have timing constraints imposed at three different levels, system-wide communication deadlines, service deadlines (both on the service computation and communication) and sub-computation deadlines (within the computation of an individual service). A computation has a set of execution time constraints, maximum and minimum execution time for a computation may be chosen from this set at implementation level or at runtime. For a composed system of services, the configuration is fixed at any point of time. This does not imply that the topology cannot

change. It simply means that for a snapshot of the composition, the topology is fixed. Dynamic reconfiguration of the service topology can be realised if some services terminate, fail or new services are added to the system. Time is discrete, linear and modelled naturally by positive integers. To avoid the overhead of synchronisation between local clocks of the services in the system, we assume the existence of a utility service called "Clock" . It provides a single, global and logical [18] clock for all the services in the composed system. "Clock" provides a function that generates a linear sequence of integers corresponding to the sequence of ticks in a physical clock. As the integers are generated they are broadcasted to all the services in the system as shown in Figure 3. Each service runs a daemon "updateClock"
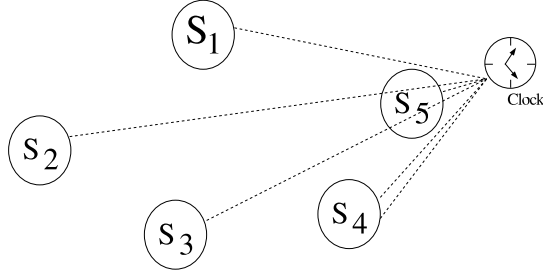
Figure 3: Broadcast of logical clock values

that receives the broadcasted value to the service. "updateClock" is referenced by each service while sending messages to peers or receiving messages from peers. There is a local variable "clock", associated with every service, which is updated by "updateClock" with every broadcast of the clock values. There is a unique 'first time' instant set by "Clock", as shown in Figure 4 from which we assume a service oriented systems will measure the message sending and receiving times for all services. A physical local clock may be internally referenced by a service for functional computation that cause local state changes.
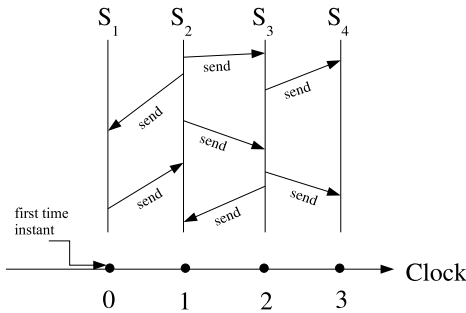
Figure 4: Clock values referenced by the services in the composition
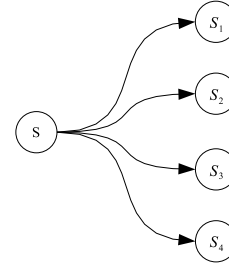
2.4 Failure

Failure of a service oriented system is defined as the unresolved failure of one or more services in the system. A service is said to have failed if

- There are communication problems i.e. breaking of channels.

- It does not return the results of an invocation within a predefined, stipulated period of time.

- If it returns an error message. Services may return error messages due to several reasons e.g. the host server being down, maintenance work being carried out on the service amongst others.

Failure of a crucial service may need to be compensated before deciding on the rollback of the overall service composition. As shown in Figures 5 and 6, failure support for a service oriented system can be provided by,

- invoking alternative services for the same computation in parallel. Results returned first are used to carry on further computations, while others are discarded.

- waiting for an invoked service to return for a predefined interval of time and then invoking one or more alternative service.

- rolling back the entire composition.

Figure 5: Failure Support in service oriented systems (01)
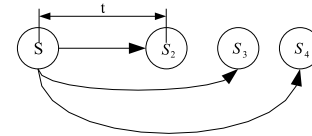
Figure 6: Failure Support in service oriented systems (02)

3. INTERVAL TEMPORAL LOGIC

We base our work on Interval Temporal Logic (ITL) [26, 25, 5]. Our selection of ITL is based on a number of points. It is a flexible notation for both propositional and first-order reasoning about periods of time. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulae in a slightly modified version of ITL. Tempura: an executable subset of ITL, provides a framework for developing, analysing and experimenting with suitable ITL specifications.

3.1 ITL: Syntax and Semantics

An interval is considered to be a (in)finite sequence of states, where a state is a mapping from variables to their values. An Interval σ in general has a length $|\sigma| \geq 0$ and a nonempty sequence of $|\sigma| + 1$ states $\sigma_0 \ldots \sigma_{|\sigma|}$. Thus the smallest intervals have length 0 and one state.

The syntax of ITL is defined in Figure 7. μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

Expressions
$e ::= \mu \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \imath a\colon f$

Formulae
$f ::= p(e_1, \ldots, e_n) \mid \neg f \mid f_1 \wedge f_2 \mid$
$\forall v \bullet f \mid \mathsf{skip} \mid f_1 \,;\, f_2 \mid f^*$

Figure 7: Syntax of ITL

ITL contains conventional propositional operators such as $\wedge$, $\neg$ and first order ones such as $\forall$ and $=$. There are temporal operators like "; *(chop)*", "* *(chopstar)*" and "skip". Additionally in ITL, there are temporal operators like $\bigcirc$(next) and $\square$(always). Expressions and Formulae are evaluated relative to the beginning of an interval.

The informal semantics of the most interesting constructs are as follows:

- $\imath a\colon f$: the value of a such that f holds. If there is no such an a then $\imath a\colon f$ takes an arbitrary value from a's range.

- skip : unit interval (length 1).

- $f_1; f_2$: holds if the interval can be chopped into a prefix and a suffix interval such that f_1 holds over the prefix and f_2 over the suffix.

- f^* : holds if the interval is decomposable into a number of intervals such that for each of them f holds.

The formula skip has no operands and is true on an interval iff the interval has length 1 (i.e. exactly two states). Both chop and chopstar permit evaluation over various subintervals. A formula $S; T$ is true on an interval σ with states $\sigma_0 \ldots \sigma_{|\sigma|}$ iff the interval can be chopped into two sequential parts sharing a single state σ_k for some $k \leq |\sigma|$ and in which the subformula S is true on the left part $\sigma_0 \ldots \sigma_k$ and the subformula T is true on the right part $\sigma_k \ldots \sigma_{|\sigma|}$. An example temporal formula is skip $; (J = I + 1)$. This formula is true over an interval σ iff σ has two states σ_0, σ_1 and $J = I + 1$ is true in the second one i.e σ_1. A formula S^* is true over an interval iff the interval can be chopped into zero or more sequential parts and the subformula S is true on each. Figure 8 pictorially represents the semantics of *skip*, *chop* and *chopstar*. Some ITL formula together with intervals which satisfy them are shown in Figure 9.

Derived constructs: Following is a list of some derived constructs which are useful for the specification of systems:

- $\bigcirc f \mathrel{\hat{=}} \mathsf{skip} \,;\, f$: next f, f holds in the next state. Example: $\bigcirc X = 1 \mathrel{\hat{=}} \mathsf{skip}; X = 1$: Any interval such that the value of X in the second state is 1 and the length of that interval is at least 1.

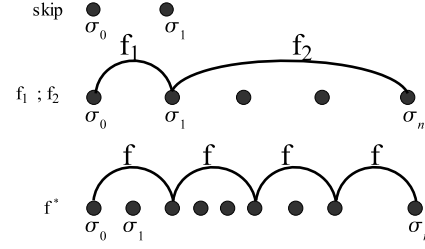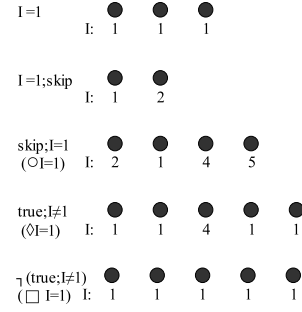**Figure 8: Informal illustration of ITL semantics**

Figure 9: Some sample ITL formulae and satisfying intervals

- $more \mathrel{\hat{=}} \bigcirc true$: non-empty interval, i.e., any interval of length at least one.

- $\mathsf{empty} \mathrel{\hat{=}} \neg more$: empty interval, i.e., any interval of length zero (just one state).

- $inf \mathrel{\hat{=}} true \,;\, false$: infinite interval, i.e., any interval of infinite length.

- $finite \mathrel{\hat{=}} \neg inf$: finite interval, i.e., any interval of finite length.

- $\Diamond f \mathrel{\hat{=}} finite \,;\, f$: sometimes f, i.e., any interval such that f holds over a suffix of that interval. Example: $\Diamond X \neq 1 \mathrel{\hat{=}} finite \,;\, X \neq 1$: Any interval such that there exists a state in which X is not equal to 1.

- $\square f \mathrel{\hat{=}} \neg \Diamond \neg f$: always f, i.e., any interval such that f for all suffixes of that interval. Example: $\square X = 1 \mathrel{\hat{=}} \neg(finite \,;\, X \neq 1)$: Any interval such that the value of X is equal to 1 in all states of that interval.

- $fin\ f \mathrel{\hat{=}} \square(\mathsf{empty} \supset f)$: final state, i.e., any interval such that f holds in the final state of that interval.

- $\bigcirc exp \mathrel{\hat{=}} \imath a\colon \bigcirc(exp = a)$: next value, i.e., the value of exp in the next state of the interval.

- $fin\ exp \mathrel{\hat{=}} \imath a\colon fin(exp = a)$: end value, i.e., the value of exp in the last state of the interval.

3.2 Types in ITL

There are two basic inbuilt types in ITL. These are integers N and Boolean (true and false). In addition the executable subset of ITL (tempura) has basic types: integer, character, boolean, list and arrays. Further types can be

built from these by means of X and the power set operator P (in a similar fashion as adopted in the specification language Z [16]). For example the following introduces a variable x of type T.

$$(\exists x : T).f \stackrel{\text{def}}{=} \exists x.type(x, T) \wedge f$$

Here $type(x, T)$ denotes a formula that describes x to be of type T. Although this might seem to be a rather inexpressive type system, richer types can be added following that of Spivey [36].

4. ASDL: ABSTRACT SERVICE DESIGN LANGUAGE

ASDL is a wide spectrum [2] language for,

- the behavioural specification of a service.

- the specification of service composition.

- design of interaction protocols for services.

ASDL abstracts from providing the commonly found XML-based dialects for service description and focuses instead on providing a minimalistic syntax with specification oriented semantics for describing services and their compositions. ASDL provides a convenient model for the verification of properties of services. Being wide spectrum, ASDL provides capabilities of modelling service oriented systems and specification of their desired properties, while remaining within a unified logical and semantic framework of ITL. As illustrated in Figure 10, compositions modelled using ASDL can be checked for conformance to desired properties specified in ITL, by reasoning about them at the same level of semantic abstraction and without the need for any external mappings, as commonly observed in other design notations for behavioural specification of services. ASDL assumes the
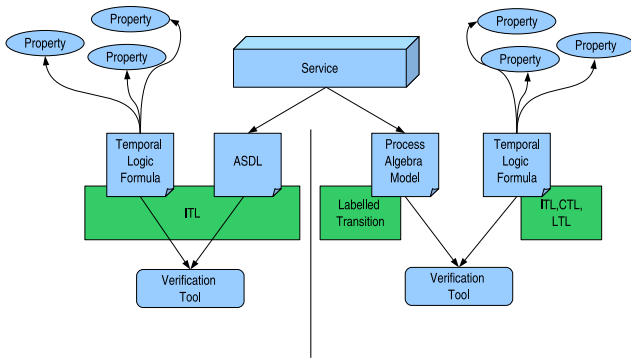

Figure 10: Semantic Unification in ASDL

existence of a registry, using which "named" services can be discovered for service composition. The computational model presented in Section 2 represents the abstract architecture for the execution of services designed using this language. During the design of the language we have striven to find a minimum set of operators and constructs from which more complex and specialised services can be composed. Although abstract and focusing primarily on the global sequence of message exchange between services, the language can be easily mapped to some of the most common

constructs for services provided by XML-based description languages such as WSDL [7, 27], industrial specifications e.g. WSBPEL [39], Microsoft's .Net, Java Message Service as well as academic research in service composition such as [38, 24]. It is hoped that the minimum set provided here will form a rich enough basis to undertake the development of service oriented systems at an abstract level and upon which users can define their own, specifically needed operators.

4.1 On the choice of constructs in ASDL

ASDL builds upon a choreography [28] based approach to specifying interaction protocols for services, although mapping to orchestration languages such as WSBPEL is equally intuitive as shown in [35]. The objective is to specify the mutually visible message exchange behaviour between services, without revealing their internal behaviour. Interactions between services are considered to be asynchronous, long-lived and stateful. Along with providing the most commonly found constructs for structured composition of processes, any notation for describing interaction protocols for such scenarios should enable:

- Representation of time, as an explicit identifier of the time at which messages are sent: Since communication is asynchronous, it is not known how long the message takes to arrive from the sender to the receiver, in the absence of such a parameter.

- Correlation between sent and received messages.

- Representation of wait, timeouts, interrupts and conditionals: service behaviour is invariably data-dependent. A service may choose to timeout if data expected is not received within a stipulated time frame. Services may interrupt other services, if data-dependent conditions are not satisfied.

- Specifying all possible behaviours: service composition does not always lead to desirable service behaviour. In some cases, the behaviour may be completely unknown.

- Guarded choices with priority: Most specification languages provide capabilities to specify non-deterministic choice. However for a composition, services need the capability to prioritise and conditionally choose from a set of services, rather than making a random choice at runtime.

Another motivating factor in making a choice of constructs for ASDL, is the relationship it defines with current XML-based industry specification standards like BPEL4WS and WS-CDL and with academic initiatives such as OWL-S and WSMO. A service specified in ASDL at the abstract level, should be capable of being represented in any of the XML-based representation formats.

In the following sections, we present the syntax of ASDL along with its informal and formal semantics. An e-shopping service is used as an example to explain certain constructs in the language. Due to space restrictions, we provide explanation for limited constructs however the complete example and algebraic rules for reasoning about useful properties of service composition designed using ASDL can be found in [35].
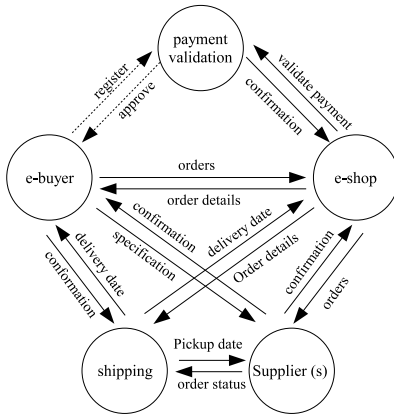
Figure 11: An e-shopping Service

4.2 An e-shopping service

Figure 11 shows a service composition that consists of five services: an e-buyer agent, an e-shop for made-to-specification goods, suppliers that stock/manufacture goods to be sold through the e-shop, shipping agency that ships the ordered goods to the e-buyer and a payment validation service. In the most typical (simplified) scenario, the e-buyer sends details of its requirements for a particular product to the e-shop. The e-shop provides a list of suppliers that can possibly meet those requirements and for which it works as a price - negotiating agent. The e-buyer forwards the product specification to the chosen supplier(s), who confirm if it is possible to supply the product. The suppliers forward the costing of the order to the e-shop. The e-buyer negotiates the price with the e-shop and places the order. The supplier contacts a shipping agency about delivering the order. The shipping agency replies with the pickup date. Finally the shipping agency informs the e-shop and the e-buyer about the delivery date. The e-buyer sends a confirmation on receiving the placed order. Since the goods are made-to-specification, cancellation of an order is allowed only within 24 hrs of placing the order.

4.3 Syntax

The modular abstraction mechanism in ASDL is the "Service" whose syntax can be defined recursively using constructs defined in Table 1, where $S, S_1, S_2...$ refer to names of services.

$$
\begin{aligned}
\mathcal{PS} ::= \ & send(id, y, \mathcal{S}_1, \mathcal{S}_2) \mid receive(id, x, t, \mathcal{S}_1, \mathcal{S}_2) \mid \Delta t \mid \\
& x := e \mid \mathsf{empty} \mid \mathsf{skip} \mid STOP \mid MAGIC \\
\mathcal{CS} ::= \ & \mathcal{PS} \mid var\ x:\ T\ in\ \mathcal{CS} \mid [t_1, t_2 \dots t_n]\mathcal{CS} \mid \\
& \mathcal{CS}_1\ ; \mathcal{CS}_2 \mid \mathcal{CS}_1 \parallel \mathcal{CS}_2 \mid \mathcal{CS}^n \mid \\
& \mathcal{CS}_1 \rhd_t^c \mathcal{CS}_2 \mid \bigsqcup_{m \in I} < p_m >: G_m \to \mathcal{CS}_m
\end{aligned}
$$

Table 1: Syntax: Design notation

Table 2 lists the names of some of the services in the e-shopping example.

- $\mathcal{PS}$ denotes a primitive service.

Service	Service Name
e-Buyer	*eBuyer*
e-Shop	*eShop*
Supplier	*supplier*

Table 2: Name declaration for the e-shopping service

- $\mathcal{CS}$ denotes a composite service.

- *var x : T in S* defines a variable within $\mathcal{S}$ of type T. Table 3 defines some of the variables used by the services in the e-Shopping Service.

var x :	T in S
var orderRequest :	*String in eBuyer*
var decisionAndPrice :	*Object in supplier*
var pricingInfo :	*Object in eShop*
var noCancellation :	*String in eShop*
var clock :	*integer in eShop*
var clock :	*integer in supplier*

Table 3: Variable declaration for the e-shopping service

- t is a time instant, modelled as $t \in N$. Note that *clock* is a special case of t, as *clock* denotes the value of t at exactly that instant.

- send(id, y, $\mathcal{S}_1$, $\mathcal{S}_2$) writes the message "y" with an identifier, "id" to be sent from service $\mathcal{S}_1$ to service $\mathcal{S}_2$, over a channel, time-stamping it with the time of the write i.e. the value of *clock* at that instant.

  The e-buyer sends his request of ordering a product to the e-shop

  $$send(i, orderRequest, eBuyer, eShop)$$

- receive(id, x, t, $\mathcal{S}_1$, $\mathcal{S}_2$) performs an input to receive the message triple, (id, x, t) sent by service $\mathcal{S}_1$ to service $\mathcal{S}_2$, where i is the message identifier, x is the message content and t is the time, the message was sent.

  The supplier receives product specification from the e-buyer

  $$receive(id, productSpec, t_{send}, eBuyer, supplier)$$

- Δt waits for t time units.

  The e-buyer waits for t_{SPEC} time units for any of the suppliers from the e-shop to confirm his order specification, before he investigates a competitor e-shop.

  $$\Delta t_{SPEC}$$

- $x := e$ evaluates the expression e, and assigns it to x. A special case can be defined as $x := y$ which is analogous to "copy" and assigns the value in y to x.

- skip denotes a service with a unit interval.

- empty defines a primitive service that does nothing.

- $STOP$ denotes failure of a service.

- $MAGIC$ denotes the most nondeterministic of all services. The behaviour of $MAGIC$ is defined by the set of all possible behaviours. It is the least predictable service. $MAGIC$ can be considered analogous to $CHAOS$ in CSP [13].

- $[t_1, t_2, \ldots t_i, \ldots t_n]\mathcal{S}$ gives $\mathcal{S}$ a duration t_i from the set, $\{t_i \mid i \in [0, 1, 2 \ldots n]\}$. if the service terminates before t_i seconds have elapsed then it idles away to fill the interval. If the service does not terminate within t_i seconds, then it is considered to have failed.

- $\mathcal{S}_1 ; \mathcal{S}_2$ denotes a sequential composition of $\mathcal{S}_1$ and $\mathcal{S}_2$. $\mathcal{S}_1 \dagger \mathcal{S}_2$ denotes a special case of sequential composition, where execution of the services is unordered.

  The e-buyer receives the name of a supplier from the e-shop and then sends detailed specification of the requirements to the supplier.

  $$receive(i, supplierName, t_{send}, eBuyer, eShop);$$
  $$send(j, productSpec, eBuyer, supplierName)$$

- $\mathcal{S}_1 \parallel \mathcal{S}_2$ denotes the parallel composition of $\mathcal{S}_1$ and $\mathcal{S}_2$, terminating when both the services terminate (distributed termination). The service terminating first idles away for the remaining time units. $\overset{n}{\parallel} \mathcal{S}$ is a spe-
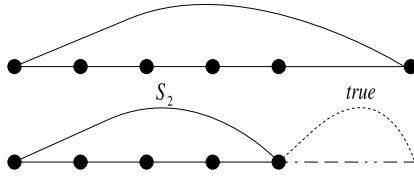

Figure 12: Distributed termination $\mathcal{S}_1 \parallel \mathcal{S}_2$

cial case that represents an n copies of $\mathcal{S}$ in parallel. $\overset{*}{\parallel} \mathcal{S}$ represents an (in)finite number of copies of $\mathcal{S}$ in parallel.

On receiving a product specification from the e-buyer, the supplier makes a decision on supplying the product. It informs the e-shop of the decision along with the price, and informs the e-buyer about the decision to supply.

$$receive(i, productSpec, t_{send}, supplier, eBuyer);$$
$$(send(j, decisionAndPrice, supplier, eBuyer) \parallel$$
$$send(j, decision, supplier, eShop))$$

- $\mathcal{S}^n$ executes $\mathcal{S}$ iteratively for "n" number of times, where n is an integer (n $\in$ N). while G do $\mathcal{S}$ and repeat $\mathcal{S}$ until G are special cases of $\mathcal{S}^n$ true.

  The e-buyer receives the list of suppliers from the e-shop and then sends detailed specification of the requirements to each of the suppliers.

  $$receive(i, supplierList, t_{send}, eBuyer, eShop);$$
  $$\text{while } (noOfSuppliers < 0) \text{ do}$$
  $$send(j, productSpec, eBuyer, supplierList)$$

- $\mathcal{S}_1 \rhd_t^c \mathcal{S}_2$ executes $\mathcal{S}_1$ till the condition "c" is true. If c holds after exactly t time units, it interrupts $\mathcal{S}_1$ and starts executing $\mathcal{S}_2$. If "c" is not true after exactly t time units, it continues to execute $\mathcal{S}_1$.
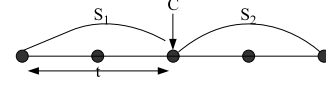

Figure 13: Interrupt: $\mathcal{S}_1 \rhd_t^c \mathcal{S}_2$

The e-buyer sends product specification to each of the suppliers from the list of suppliers received from the e-shop. While sending the messages, if the one of the suppliers confirms the order, the e-buyer interrupts the sending operation and starts negotiating the price with the e-shop.

$$\text{while } (noOfSuppliers < 0) \text{ do}$$
$$send(j, productSpec, eBuyer, next(supplierList)) \rhd_t^c$$
$$send(i, supplierDetails, eBuyer, eShop)$$

where, $c = confirmOrder(supplier)$

- $\displaystyle\bigsqcup_{m \in I} < p_m > : g_m \to \mathcal{S}_m$ evaluates the guards (g_m) in decreasing order of priority (p_m) and executes a service corresponding to a true guard. We define guards (g) to be state formula. Priority of a guard is defined as $p \in N$, increasing progressively as the priority increases. If all the guards evaluate to false then the service terminates.

  If a set of services has the same priority and all guards evaluate to true, the choice is made non-deterministically, i.e. $\mathcal{S}_1 \sqcap \mathcal{S}_2$ makes a non-deterministic choice between $\mathcal{S}_1$ and $\mathcal{S}_2$. "if G then $\mathcal{S}_1$ else $\mathcal{S}_2$" is a special case.

  On receiving a message to cancel an order, the e-shop checks the time when the order was placed (t_{order}) and compares it with the current time to send the appropriate message to the e-buyer

  $$\text{if } (clock - t_{order}) < 24 \text{ then}$$
  $$send(j, cancellation, eShop, eBuyer) \text{ else}$$
  $$send(j, noCancellation, eShop, eBuyer)$$

4.4 Specification Oriented Semantics of ASDL

In this section, we define an ITL based, specification oriented semantics for various constructs of ASDL. The semantics of *chop* (;), skip and empty are as defined in ITL.

The semantics of *receive* and *send* are defined with reference to Figure 14. A channel running from service S_i to S_j, $\mathcal{C}_{ij}$, is said to be directed from S_i to S_j. The channel is defined as an l place bounded buffer. The channel is divided into two parts, "head" and "tail", with head at the 0 th position in the buffer. The head is defined as a single position at the directed end of the channel. The tail is defined as the rest of the channel i.e. a buffer with $l - 1$ places. At any time, the number of empty places in the buffer is denoted by $|\mathcal{C}_{ij}|$. Sending of a message is allowed only if there is an empty position in the channel. In order to send a message,

the operator $\oplus$ appends the message to the last empty position in the channel. A pointer keeps track of the last empty position in the buffer where the message is to be appended. We define the set of all messages, M as,

$$M = [N \times V \times Time]$$

where V is an infinite set of message values and $Time$ is an infinite set of time stamps. A message, $m \in M$ that is sent from service S_i to S_j, is a triple (id, x, t), where $id \in N$ is a unique message identifier for correlating between the messages sent and received between S_i and S_j, and x is the message content i.e. data content that is sent or received. The time-stamp, t of message transmission is passed along with the message. We define a projection function, Π_k on the head of the channel C_{ij} which for k=1 gives id, for k=2 gives x and for k=3 gives t, the time-stamp.

- $\Delta t \;\widehat{=}\; fin(clock) = clock + t$

- $x := e \;\widehat{=}\; \bigcirc x = e$

- $STOP \;\widehat{=}\; false$

- $MAGIC \;\widehat{=}\; true$

- $var\ x :\ T\ in\ \mathcal{S} \;\widehat{=}\; \exists x.type(x, T) \wedge \mathcal{S}$

- $receive(id, x, t, \mathcal{S}_1, \mathcal{S}_2) \;\widehat{=}\;$

  $\mathsf{skip} \wedge if\ |\mathcal{C}_{12}| > 0$
  $then(id = \Pi_1(head(\mathcal{C}_{12})) \wedge x = \Pi_2(head(\mathcal{C}_{12})) \wedge$
  $\qquad\qquad t = \Pi_3(head(\mathcal{C}_{12})) \wedge \bigcirc(\mathcal{C}_{12}) = tail(\mathcal{C}_{12}))$
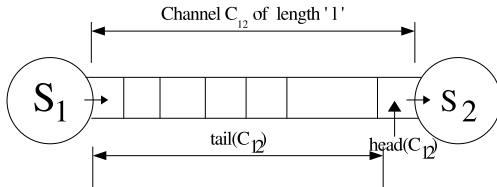
Figure 14: Channel C_{12} from service S_1 to S_2

- $send(id, y, \mathcal{S}_1, \mathcal{S}_2) \;\widehat{=}\; \mathsf{skip} \wedge if\ |\mathcal{C}_{12}| < l\ then(\bigcirc\mathcal{C}_{12} = \mathcal{C}_{12} \oplus < id, y, (clock) >)$

- $[t_1, t_2, \ldots t_n]\mathcal{S} \;\widehat{=}\; for\ some\ t_i \in \{t_1, t_2, \ldots t_n\}, (\Delta t_i \wedge (\mathcal{S}\ ; true)$

- $\mathcal{S}_1 \dagger \mathcal{S}_2 \;\widehat{=}\; \mathcal{S}_1\ ; \mathcal{S}_2 \vee \mathcal{S}_2\ ; \mathcal{S}_1$

- $\mathcal{S}_1 \parallel \mathcal{S}_2 \;\widehat{=}\; ((\mathcal{S}_1\ ; true) \wedge \mathcal{S}_2) \vee (\mathcal{S}_1 \wedge (\mathcal{S}_2\ ; true))$

- $\overset{*}{\parallel} \mathcal{S} \;\widehat{=}\; \exists n.\ \overset{n}{\parallel} \mathcal{S}, \overset{0}{\parallel} \mathcal{S} \;\widehat{=}\; \mathsf{empty}, \overset{1}{\parallel} \mathcal{S} \;\widehat{=}\; \mathcal{S}$

- $\mathcal{S}_1 \sqcap \mathcal{S}_2 \;\widehat{=}\; \mathcal{S}_1 \vee \mathcal{S}_2$

- $\mathcal{S}^n \;\widehat{=}\; \mathcal{S}^n$

- $\mathsf{while}\ G\ \mathsf{do}\ \mathcal{S} \;\widehat{=}\; (G \wedge \mathcal{S})^* \wedge fin\ \neg G$

- $\mathsf{repeat}\ \mathcal{S}\ \mathsf{until}\ G \;\widehat{=}\; \mathcal{S}\ ; (\mathsf{while}\ \neg G\ \mathsf{do}\ \mathcal{S})$

- $\mathcal{S}_1 \rhd_t^c \mathcal{S}_2 \;\widehat{=}\;$

  $\mathcal{S}_1 \wedge (fin(c) \wedge \Box(more \supset \neg c) \wedge (fin(clock) = clock + t); \mathcal{S}_2) \vee$
  $\qquad\qquad (\mathcal{S}_1 \wedge \Box(\neg c) \wedge \Delta t)\ ; \mathcal{S}_2$

- $\bigsqcup_{m \in I} <p_m>: G_m \to \mathcal{S}_m \;\widehat{=}\;$

  $\bigvee_{m \in I} ((G_m \wedge \mathcal{S}_m) \wedge p_m = max(P)) \vee$

  $(\bigwedge_{m \in I} \neg G_m \wedge \mathsf{empty})$ where $P = \{p_m | m \in I \wedge p_m \in N\}$

- $\mathsf{if}\ G\ \mathsf{then}\ \mathcal{S}_1\ \mathsf{else}\ \mathcal{S}_2 \;\widehat{=}\; (G \wedge \mathcal{S}_1) \vee (\neg G \wedge \mathcal{S}_2)$

5. RELATED WORK

In approaches such as Interface Automata [10], IO Automata [20], CCS [21] and CSP [12] the communication between the receive and send processes is synchronous, while in our model, the messages are stored in a bounded FIFO buffer. We believe that modelling using synchronous communication does not capture the realistic scenario of composition, because of the inherent distributed nature of Web services. Formalisms such as π-calculus and the approach used in Microsoft's Behave! project [30] use asynchronous message passing for modelling distributed and concurrently executing systems. Our model differs from Communicating Finite State machines [3], as it uses an unbounded buffer whereas the buffers in our model are bounded FIFOs. Our model is close to the physical implementation, where all buffers are bounded, however large the bound maybe. In Single Link Communicating Finite State machines [29], incoming messages from several processes to a single process come from a single FIFO, in our model, incoming messages to a single service from different services, come from separate FIFO channels. A crucial difference between CSFMs and our model is that in the former, time aspects are ignored. Messages sent out by the sending process are assumed to reach the receiving process instantly. In our model, sending and receiving of messages, takes at least one unit (i.e.$skip$) of time. Further, timeouts are not handled explicitly in CSFMs as in our model. In Kahn's Process networks [17, 37] the unbounded FIFO buffers are one way, i.e., they can be used only for receiving or sending, which is similar to our model. In the Temporal Agent Model (TAM) [19, 34] communication is asynchronous via time - stamped shared area called shunts.

The model proposed in [40] assumes a virtual watcher as an abstract entity which records the sequence of message as they are sent to peers, each peer is further modelled as a standard finite state automaton. Communication is similar to our model, i.e., over asynchronous - FIFO buffers, however the model does not specify if the buffers are bounded. Further the model describes a closed system where peer implementations are known a priori. This simplifies the modelling of peers considerably. A similar approach to closed systems can be found in [15]. The model assumes one message queue per peer and peer implementations are modelled as Mealy machines [14]. The model we propose describes an open system and encapsulates a *black box* view of service. An XML based notation with formal semantics in π-calculus, to model service composition is described in [32]. Although the prime objective here is an orderly sequencing

of messages, the notation does not provide any primitives for modelling services such as "timeout". Misra et al. [31, 24, 23] present a model "Orc" based on transactions. In this model a service is represented by a "Site". The model does not include features for time-out, synchronization, communication and the like. There is no notion of messages being passed or variables being shared. A Site call is analogous to calling a function. This view is analogous to legacy distributed computing paradigms such as CORBA [11] and DCOM [8]. Orc provides a design notation with operational semantics, analogous to ASDL with specification oriented semantics in our model. An interesting algebra for services is present by Cardeli and Davies et al. in [4] where the notion of a service is that of a HTTP information provider wrapped in error detecting and handling code. The idea is to provide service combinators that simulate the process of manual browsing on the web. Combinators, are defined to enable sequence, repetition and time out constructs amongst others.

ASDL can be considered as an algebra for services, quite analogous to widely known and fundamental process algebra models such as Hoare's Communicating Sequential processes (CSP) [21], Milner's Calculus of Communicating Systems and ACP [1]. These process algebras have been widely extended to include variations such as Timed CSP [9], pi-calculus [22, 33] and LOTOS amongst others. ASDL is however more expressive as it is enriched with explicit timing related constructs which is needed in order to reason about services communicating asynchronously. A "Process" in CSP is analogous to a "Service" in ASDL. The most significant difference between CSP and ASDL, is the notion of an "event" in CSP. ASDL does not define "event" explicitly, as when a service is viewed from a "Black box" perspective, the only visible events are the sending and receiving of messages between services, which we consider as primitive services in our model. Similarly with CCS, the notion of "action" is analogous to our primitives, *send*, *receive* and *empty*. An interesting comparison arises with π-calculus and the notion of "mobility". The primitive entities in π-calculus are processes, channels [1] and names where a name is used to refer to a channel. Processes are independent entities that are connected by channels. The idea of mobility with respect to services in a composition is that channel names can be passed between services. This allows the modeling of both static and dynamic services within a composition. For example, a buyer could specify channel information to be used for sending delivery information. The buyer could then send the channel information to the seller who then forwards it to the shipper. The shipper could then send delivery information directly to the buyer using the channel information originally supplied by the buyer. It is worth noting that in our computational model, mobility is not required as every service in a network is connected to every other service via a channel as illustrated in Figure 2. Further, services in ASDL are *named services*. Dynamic configuration in our model is simply achieved by sending a message along the channel, connecting the desired service. The significant advantages that ASDL offers over algebraic techniques of composition is the complete axiomatic system of ITL, which can be used to reason about system models designed using ASDL and their associated properties, and a unified semantic framework for

reasoning about models of services and their properties, as mentioned earlier.

6. CONCLUSIONS AND FUTURE WORK

A sound computational model and an underlying wide spectrum design language are crucial artifacts in the design of service composition and interaction protocols at an abstract level and then reasoning about properties of the composition.

In this paper, we have presented a computational model for the design of service oriented systems and developed a wide spectrum language that provides us with capabilities to design service composition and model their interactions at an abstract level. We base our model on asynchronous communication of messages between services via channels. We have proposed a design language, "ASDL", for the specification and validation of reactive Web services, based on our rigorous computational model. The language is rich in expressiveness and abstracts from several details found in high level specification languages.

At the theoretical level, we would like to extend the computational model and ASDL with "fairness". Fairness is especially desired when a composing agent has to make a non-deterministic choice from the same set of services, when it executes recursively i.e.

$$(\mathcal{S}_1 \sqcap \mathcal{S}_2 \sqcap \mathcal{S}_3)^*$$

In such situations, it might be the case that service S_1 is invoked for every iteration. We would like to extend ASDL with a fairness operator that allows a fair selection between services of similar capabilities i.e.

$$(\mathcal{S}_1 \sqcap_f \mathcal{S}_2 \sqcap_f \mathcal{S}_3)^*$$

We would also like to extend the model and ASDL to provide explicit transactional support that allows compensation, negotiation of commitments, relaxation of ACID properties, exception handling and enforcement of security policies.

7. REFERENCES

[1] J. C. M. Baeten and C. Verhoef. *Concrete Process Algebra*, pages 149–268. Oxford University Press, Oxford, UK, 1995.

[2] F. L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Bruckner, H. Partsch, P. Pepper, and H. Wossner. Towards a wide spectrum language to support program specification and program development. *SIGPLAN Not.*, 13(12):15–24, 1978.

[3] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.

[4] L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Trans. Softw. Eng.*, 25(3):309–316, 1999.

[5] A. Cau. ITL and (Ana)Tempura Home page on the web. http://www.cse.dmu.ac.uk/STRL/ITL/.

[6] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[7] R. Chinnic, H. Haas, A. Lewis, J. J. Moreau, D. Orchard, and S. Weerawarana. Web services

---

[1]Also referred to as links.

description language (wsdl) version 2.0 part 1: Core language w3c working draft 3 august 2005, 2005. http://www.w3.org/TR/2005/WD-wsdl20-20050803/.

[8] M. Corporation. The Component Object Model Specification, October 1995. Draft Version 0.9.

[9] J. Davies and S. Schneider. An Introduction to Timed CSP. Technical report, Oxford University, August 1989.

[10] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC/FSE-9: Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, New York, NY, USA, 2001. ACM Press.

[11] O. M. Group(OMG). The common object request broker: Architecture and specification(corba)rev 3.0.2. omg technical document, 2004.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[14] J. E. Hopcroft and J. D. Ullman. *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[15] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: a look behind the curtain. In *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–14. ACM Press, 2003.

[16] M. Imperato. *An Introduction to Z*. Chartwell-Bratt, 1991.

[17] G. Kahn. *The Semantics of a Simple Language for Parallel Programming*. Proc Information Processing, North Holland, 1974.

[18] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[19] Lowe and H. Zedan. Refinement of complex systems: a case study. *The Computer Journal*, 38(10), 1995.

[20] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[21] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[22] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, New York, NY, USA, 1999.

[23] J. Misra. Computation orchestration: A basis for wide-area computing. *Lecture Notes for NATO summer school*, 2004.

[24] J. Misra. A programming model for the orchestration of web services. In *SEFM*, pages 2–11, 2004.

[25] B. Moszkowski. *Executing Temporal Logic Programs*. Cambridge University Press, Cambridge, England, 1986.

[26] B. Moszkowski. *Programming Concepts, Methods and Calculi, IFIP Transactions, A-56.*, chapter Some Very Compositional Temporal Properties, pages 307–326. Elsevier Science B. V., North-Holland, 1994.

[27] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon. Web Services Choreography Description Language Version 1.0: *W3C Working Draft 17 December 2004*, 2004.

[28] C. Peltz. Web services orchestration and choreography. *IEEE: Computer*, 36(10):46–52, October 2003.

[29] W. Peng. Single-link and time communicating finite state machines. In *Proc. of 1994 International Conference on Network Protocol*, pages 126–133, Boston, October 1994.

[30] S. K. Rajamani and J. Rehof. A behavioral module system for the pi-calculus. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 375–394, London, UK, 2001. Springer-Verlag.

[31] Y. ri Choi, A. Garg, S. Rai, J. Misra, and H. M. Vin. Orchestrating computations on the world-wide web. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 1–20, London, UK, 2002. Springer-Verlag.

[32] S. J. Woodman, D. J. Palmerand S. K. Shrivastava, and S. M. Wheater. Notations for the specification and verification of composite web services. In *Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)*, September 20 - 24, 2004.

[33] D. Sangiorgi and D. Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[34] D. Scholefield. *A Refinement Calculus for Real Time Systems*. PhD thesis, University of York, 1992.

[35] M. Solanki. *A Compositional Framework for the Specification, Verification and Runtime Validation of Reactive Web Service*. PhD thesis, De Montfort University, Leicester, UK, October 2005.

[36] J. M. Spivey. Richer types for z. *Formal Asp. Comput.*, 8(5):565–584, 1996.

[37] K. Stølen, F. Dederichs, and R. Weber. Assumption/commitment rules for networks of asynchronously communicating agents. Technical Report TUM-I9303, Technische Univerität München, 1993.

[38] The OWL-S Coalition. OWL-S 1.1 Release., 2004. http://www.daml.org/services/owl-s/1.0/.

[39] Tony Andrews et al. Business Process Execution Language for Web Services, Version 1.1, 2003. http://www-106.ibm.com/developerworks/library/ws-bpel/.

[40] X. Fu T. Bultan and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA)*, pages 188–200, Santa Barbara, CA, USA, 2003.