A Framework for Generic Error Handling in Business Processes

Manuel Mazzara and Roberto Lucchi

Department of Computer Science, University of Bologna, Via Mura Anteo Zamboni 7 - 40127 Bologna, Italy
E-mail: {mazzara, lucchi}@cs.unibo.it

Abstract

Recently the term Web Services Choreography has been introduced to address some issues related to Web Services Composition. Several proposals for describing Choreography for Business Processes have been presented in the last years and many of these languages make use of concepts as long-running transactions and compensations for coping with error handling. BPEL4WS, the most accredited candidate for becoming a standard in Choreography, provides three different mechanisms for coping with abnormal situations: Exception Handling, Event Handling and Compensation Handling. The complexity of BPEL4WS makes it difficult to formally define this framework, thus limiting the formal reasoning about the designed applications. In this paper we advocate that three different mechanisms for error handling are not necessary and we formalize a novel choreography language, based on the idea of event notification as the only error handling mechanism. We can take advantages of this formal description in two ways. Firstly, this language represents by itself a proposal of simplification for BPEL4WS including an unambiguous specification. Secondly, an implementor of an actual BPEL4WS orchestration engine should implement simply this single mechanism providing all the remaining ones by compilation. Notably, the proposed framework is expressive enough to compare different solutions for managing long running transactions such as BPEL4WS and StAC.

1 Introduction

Web Services technology is a platform on which we can develop applications taking advantage of the Internet infrastructure. A Web Service, specifically, describes particular business functionalities that a company wants to expose through the Internet with the purpose of providing to other companies a way for using them. More formally, the official definition of W3C [14] says that:

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

> "A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols".

Recently the term *Web Services Choreography* has been introduced to identify Web Services Composition, that is the way of defining a complex service out of simpler ones. Several proposals for describing Choreography for business process have been presented in the last years: for example BPML [4], IBM's WSFL [8], Microsoft's XLANG ([13], [10]) or the more recent BPEL4WS [1] (which represents a trade-off between IBM and Microsoft).

Choreography is about business processes. It is important to remark that business processes are characterized by (1) being of long duration, (2) using asynchronous messages for communication, (3), manipulating sensitive business data in back-end databases. Error handling in such an environment is both difficult and critical. The use of ACID transactions can be limited to local updates because trust issues and because locks and isolation cannot be maintained for long periods. We refer to transactions with these different requirements as *long running* transactions. Error Handling in this context relies on the concept of *compensation*. Most of the Choreography languages listed at the beginning use long running transactions and compensations as a mechanism for describing loosely-coupled activities.

Compensations are application-specific activities which attempt to reverse the effects of a previous activity carried out as part of a larger unit of work which is being abandoned. While for ACID transactions in databases the transaction coordinator and the resource it controls know all the uncommitted updates and have the full control on the order in which they must be reversed, in the case of business transactions the compensation behavior is itself a part of the business logic and must be explicitly specified.

In this paper we want to address, in a formal way, the problem of composing Web Services with particular attention to Error Handling. In particular, we propose a basic language to deal with Choreography. The aim of this language is to provide a mean to express common Web Service requirements. Typically, in such an environment we need a way for addressing concurrency and message passing which form the basic paradigm of the distributed computation on the Internet.

In order to formally dealing with these requirements we chose to start from the π-calculus [12], a well known process algebra which has been widely studied during the last fourteen years. We chose the π-calculus rather than other calculi because the definition of XLANG (and then BPEL) has been strongly influenced by this calculus. The strong correlation between a so theoretical and academical area and a more business oriented one should not appear surprising as many people think. This is because a part of the people involved in the second one have been previously involved in the first one and, at the moment of choosing a valid paradigm for a Choreography language, they decided for an already deeply experimented one as, for example, the π-calculus. This appears completely natural as for the invention of the car: the more natural way for implementing such a mean of transportation was to enhance an already experimented one as the coach. Installing an engine on a basin would have resulted in a queer experiment, although pretty funny!

Unfortunately, the original π-calculus does not support any transaction mechanism. For this reason, in this paper we shall extend the basic calculus in such a way to include transactional facilities. Some other works have been presented in the past addressing similar issues. Anyway, all the past works committed only to ACID or Long-running semantics without providing a general framework for formalizing both the semantics. Instead, our attempt could be interpreted in this direction.

In this document we will proceed in the following way. In Section 2 we describe briefly the most accredited candidate for becoming a standard in Choreography and we explain and motivate our goals. In Section 3 we introduce the language syntax and in Section 4 its semantics. In Section 5 we shall informally show how to program the error handling mechanisms provided by BPEL4WS in our framework. Finally, in Section 6 we describe some related work and we report some conclusive remarks.

2 State of the Art in WS Choreography

The Business Process Execution Language for Web Services (BPEL4WS) is the fusion of IBM'S WSFL and Microsoft's XLANG and it is actually supported by both. So far, it represents the most accredited candidate for becoming a future standard in the field of Web Services Choreography. For this reason it deserves to be studied and considered as a touchstone for any further effort in this field.

BPEL4WS (BPEL for short) is, in practice, a layer on top of WSDL [7]. Roughly speaking, WSDL is used for defining message types and port types; such messages and ports are then used by BPEL for specifying the flow of actions. A BPEL document is an XML-based document that can be executed by an Orchestration Engine, which is the central coordinator. The engine will read the BPEL document and will invoke the necessary Web Services in the required order. The process itself will be offered as a Web Service and will be invoked in such a way.

A complete explanation of BPEL is beyond the purposes of this paper. Anyway, it is crucial to focus on the fact it provides three different mechanisms for coping with abnormal situations: Exception Handling, Event Handling and Compensation Handling. In this paper we advocate that three different mechanisms for error handling are not necessary and we formalize a novel choreography language based on the idea of event notification as the only error handling mechanism. We can take advantages of this formal description in two ways. Firstly, this language represents by itself a proposal of simplification for BPEL including an unambiguous specification. Secondly, an implementor of an actual BPEL orchestration engine should implement simply this single mechanism providing all the remaining ones by compilation.

It is worth to note that the BPEL Event Handling mechanism was not specifically designed for error handling. Although it is not a goal of this work, we consider that the proposed language still allows for all the remaining usages of the original mechanism. In order to prove the adequacy of the proposed mechanism, in Section 5 we shall informally show how to program the three mechanisms provided by BPEL in our framework. Any formal comparison is beyond the scope of this paper and it is left as future work. Anyway, it is worth noting that a more precise comparison with BPEL is an hard task because, presently, it lacks of any formal specification.

3 The Language Syntax

Let $\mathcal{N}$ be a numerable set of channel names and $\mathcal{T}$ be a numerable set of scope names. The set of processes, ranged over by P, Q, R, ... is defined by the following grammar:

$$\begin{array}{rll}
P ::= & \mathbf{0} & \texttt{Normal Termination} \\
\mid & \overline{x}\,\tilde{v}.P & \texttt{Output} \\
\mid & x(\tilde{u}).P & \texttt{Input} \\
\mid & (n)P & \texttt{New Name Creation} \\
\mid & P \mid P & \texttt{Parallel Execution} \\
\mid & A(\tilde{u}) & \texttt{Process Invocation} \\
\mid & \texttt{signal}(t) & \texttt{Raising of a Signal} \\
\mid & [P,Q].R & \texttt{Event Scope}
\end{array}$$

where n is a name in $\mathcal{N}$, $\tilde{v}$ represent a list of elements in $\mathcal{N} \cup \mathcal{T} \cup \{\texttt{this}\}$, $\tilde{u}$ represent a list of elements in $\mathcal{N} \cup \mathcal{T}$ and $t \in \mathcal{T} \cup \{\texttt{this}\}$. The term `this` is a keyword which represents the identifier of the current event scope. The keyword must be used uniquely inside the body a scope definition, we will define this notion of well formedness in the following. We are assuming a set of process constants, ranged over by A, in order to support process definition, whose definition follows:

Definition 3.1 (Process Definition) *A defining equation for a process identifier A is of the form*

$$A(\tilde{u}) \overset{def}{=} P$$

The first five operators are as usual: the $\mathbf{0}$ simply describes the normal termination of a process. The meaning of an *Output* $\overline{x}\ \tilde{v}.P$ is sending a list $\tilde{v}$, the *object* of the communication, through the channel x, the *subject*. The *Input* prefix $x(\tilde{u}).P$ represents the reception of the object $\tilde{u}$ through the channel x and it is a binder for the names $\tilde{u} \in \mathcal{N} \cup \mathcal{T}$ (these names can be channel names or scope names). The *New Name Creation* operator, instead, is a binder for the name $n \in \mathcal{N}$. The parallel operator represents the support for concurrency as the `flow` activity in BPEL. As in BPEL, the world here is modelled by concurrent activities which interact by message passing and event raising. BPEL allows for Web Services composition providing the `invoke` activity. In the same way, the process invocation à la π-calculus allows us to compose many different uncoupled services. So far the language is strictly similar to the π-calculus, it differs only for the last two operators. The first one is $\texttt{signal}(t)$ which produces a signal directed to the event scope identified by t. It is assumed that each event scope $[P, Q]$ is identified by a scope name (e.g., the scope $[P, Q]_t$ is identified by t), which is determined during the execution (more precisely, at the moment of the scope allocation). This will be clear in the following showing the semantics rules. For now, it is necessary to understand that an event scope defines a process P to be run during the normal execution and an event handler Q. During the execution, when a scope is opened, the defined handler is allocated. At that point, an identifier is associated with the scope. This information is provided by the system to the body of the scope which can refer this identifier with the keyword `this`. If some process running in parallel will raise a signal directed to it, the event handler will be eventually executed (the activation is asynchronous due to physical latency) and then the process itself terminated. Signals directed to nonexistent scope are lost. A straightforward consequence of this fact is that a scope can catch a signal only once.

We define free names $fn(P)$ of a process P as in the π-calculus with the necessary extension for $\texttt{signal}(t)$ and $[P, Q].R$:
$fn([P, Q].R) = fn(P)\ \cup\ fn(Q) \cup\ fn(R)$,
$fn(\texttt{signal}(t)) = \{t\}$.

In the next section we shall give the semantics for well formed processes, whose definition follows.

Definition 3.2 (Well-Formed Process) *A process is well-formed if the following conditions hold:*

(i) *For each process definition it contains* $A(\tilde{u}) \stackrel{def}{=} P$ *it holds* $fn(P) \subseteq \{\tilde{u}\}$ *and* $\tilde{u}$ *is composed by pairwise distinct names,*

(ii) *The keyword* `this` *never occurs outside the body* `P` *of a scope definition* `[P,Q].R`.

We use $\mathcal{P}$ to denote the set of processes that are well-formed.

4 The Language Semantics

In our language all the scopes are characterized by an unique identifier. The knowledge of this identifier is fundamental for a process to be able to interact properly with the previously allocated scope. When the main process inside a scope terminates also the relative event handler is removed and then it becomes unaccessible. Practically, processes have two way for interacting: (1) Message Passing and (2) Event Raising. It is not hard to understand the two mechanisms are mutually encodable. Anyway, here the goal is not about the minimality of the language but about providing mechanisms to describe, at an abstract level, all the important aspects a Web Service Choreography language needs. Pay attention to the fact that this sentence is not in contradiction with our previous claim about redundancy of BPEL's mechanisms for Failure Handling. Indeed, we are searching for the smallest set of operators which can meet the needed requirements for Web Services Choreography offering a reasonable simplicity to the application designers. Programming complex business processes with Failure Handling only in term of message passing does not seem reasonable. With this motivation we are claiming that — in this context — to add a mechanism of event handling is necessary and sufficient.

A program, statically, can be viewed as a tree of nested scopes. During its execution, instead, each scope is provided with an index identifying itself and it is extruded creating a flat structure. It is not provided with another index identifying the father in the ancestors's tree of nested scopes. Originally, we introduced also this information in the language; anyway it is worth to note that, avoiding the introduction of this information, is a significant advantage for the language because it simplifies the definitions of business processes and the semantics rules. Nevertheless, it does not seem to grow the complexity in the programming of Business Process phases. Before giving the semantics for the language, we have to formally define the concept of system state which is necessary for the dynamic handling of scope identifiers.

Definition 4.1 (System State) *The set $\mathcal{S}$ of the system states is defined as follows:*

$$S ::= \; P \mid \; [P,Q]_t \mid S|S$$

where $t \in \mathcal{T}$.

The intended meaning of $[P,Q]_t$ is to identify the scope $[P,Q]$ that at runtime has been associated to the identifier t. As it will be clear in the following (see Table 1), this allocation is obtained by substituting the keyword `this` with a fresh name t. We denote this special case of substitution as usual by $P\{t/\texttt{this}\}$ but the definition is slightly different: we define it as the replacement, in the body of the outer scope appearing in the process P, of each occurrence of the keyword `this` with the name t. The remaining occurrences of `this` will be substituted in the same manner when allocating the corresponding scopes. Now we extend the definition of free names on system states by adding only one rule: if S is of the form $[P,Q]_t$ we define

$fn([P, Q]_t) = (fn(P) \cup fn(Q)) \setminus \{t\}$.

Now we shall give the semantics for the language in two steps, following the approach of Milner [11]. This approach consists in separating the laws which govern the static relations between processes from the laws which rule their interactions. We shall achieve this defining firstly a static Structural Congruence relation over syntactic processes. A Structural Congruence relation for processes is introduced as a small collection of axioms that allow minor manipulation on the processes structure. This relation is intended to express some intrinsic meanings of the operators, for example the fact that parallel is commutative. Secondly, we shall define the way in which states evolve dynamically by means of a Labeled Transition System. Doing in this way we simplify the statement of the transition system just adding the (CONGR) rule in Table 1 which closes the transition relation under process order manipulation induced by Structural Congruence.

Definition 4.2 (Structural Congruence) *The structural congruence on processes $\equiv$ is the smallest equivalence relation satisfying the followings and closed with respect to α-renaming, parallel composition and restriction:*

(i) $(\mathcal{P}\ ,\ |\ ,\ \mathbf{0})$ *is an Abelian Monoid:*

$$\begin{array}{lll} P_1|P_2 & \equiv P_2|P_1 & \textit{Commutativity} \\ (P_1|P_2)|P_3 & \equiv P_1|(P_2|P_3) & \textit{Associativity} \\ P|\,\mathbf{0} & \equiv P & \mathbf{0}\ \textit{is nil element} \end{array}$$

(ii) $(n)\mathbf{0} \equiv \mathbf{0}$

(iii) $(n_1)(n_2)P \equiv (n_2)(n_1)P$

(iv) $(n)(P_1|P_2) \equiv P_1|(n)P_2$ *if* $n \notin \mathrm{fn}(P_1)$

(v) $A(\tilde{v}) \equiv P\{\tilde{v}/\tilde{u}\}$ *if* $A(\tilde{u}) \overset{def}{=} P$

In the above definition we use the usual notation for name substitution $P\{\tilde{v}/\tilde{u}\}$ which means the replacement, in the process P, of each occurrence of name in the ordered sequence $\tilde{u}$ with the correspondent name in the ordered sequence $\tilde{v}$. In the following definitions we shall consider the natural extension to states of the Abelian Monoid rules for processes.

Sometimes the semantics of a system is defined in term of a reduction relation which can result more concise. With this purpose, we also originally tried to define our semantics with a reduction relation. Anyway, subsequently we found a labeled transition system a more elegant way for describing raising of signals, scope spawning and inter-scope interactions. For this reason we decided to express the semantics in this way although it lacks of brevity. The transition relations over system states are labeled by the *actions*. We have four kind of actions as defined in the following:

Definition 4.3 (Actions) *The actions are given by*

$$\alpha ::= \overline{x}\ \tilde{v} \mid x(\tilde{u}) \mid \langle t \rangle \mid \tau$$

We shall write Act for the set of actions.

The first action is sending the tuple $\tilde{v}$ via the channel x, the second is receiving the tuple $\tilde{u}$ via x while the third is a signal of an event directed to the transaction t. Finally, τ represents an internal action. We omit the definition for $fn(\alpha)$, $bn(\alpha)$ and $subj(\alpha)$. They represent for actions respectively the set of free names, bound names and names occurring as subject in a communication. These definitions are as usual with a straightforward extension for the signal labels.

Definition 4.4 (Transition Relations) *The transition relations* $\{\xrightarrow{\alpha} | \alpha \in Act\}$ *on states* $\mathcal{S}$ *are defined by the rules in Table 1 where* $S \xrightarrow{\alpha} S'$ *means that the state* S *evolves in* S' *with the action* α.

Regarding the presented semantics three points deserve to be touched. Firstly, note that the mechanism in which we are extruding nested transactions is very similar to what happens in the Ambient Calculus [5] with the *out* operation. Also the *(Scope)* rule of the transition system reflects the one of the Ambient Calculus. The difference here lays in the fact that our extrusion is implicit and not explicitly programmed. Because this freedom of explicitly extruding transactions does not seem to have an added value in the context of Web Services, we decided to disallow it letting the system to control the evolution flattening transactions in such a way that, at runtime, they are all at the same level of nesting.

The second point is about localities. Here we decided to avoid the introduction of this notion in the language in order to be as close as possible to the notion of transaction in BPEL. In fact, it is important to recall that the notion of Long Running Transaction in BPEL is purely local and occurs within a single business process instance [1]. There is no distributed coordination regarding the outcome among multiple participants. The achievement of a distributed agreement is a problem outside the scope of BPEL. This problem can be solved by using the protocols described in the WS-Transaction [15] specification. It is worth to note that, since the need of composing WS-Transaction and BPEL is presently widely recognized and our notion of transaction seems to match quite well in a context of Process Algebras with localities [6], a next step in this research should consist in introducing a notion of locality in the language.

Finally, we want to remark that the execution of the rules *(ScopeAlloc)* and *(ScopeExtr)* generates fresh names in the whole system. Some reader could underline the fact that the relative side conditions could be expressed more rigorously. Anyway, our goal is to keep the specification as light as possible without lacking of precision and, since our semantics is implementation oriented, this solutions represents a reasonable tradeoff.

(Out) $\overline{x}\,\tilde{v}.P \xrightarrow{\overline{x}\,\tilde{v}} P$ (In) $x(\tilde{u}).P \xrightarrow{x(\tilde{u})} P$ (Signal) $\texttt{signal}(t) \xrightarrow{\langle t\rangle} \mathbf{0}$

(Local Com)
$$\frac{P \xrightarrow{\overline{x}\,\tilde{v}} P' \qquad Q \xrightarrow{x(\tilde{u})} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{\tilde{v}/\tilde{u}\}}$$

(Par)
$$\frac{S' \xrightarrow{\alpha} S''}{S' \mid S \xrightarrow{\alpha} S'' \mid S} \quad bn(\alpha) \cap fn(S) = \emptyset$$

(Res)
$$\frac{S \xrightarrow{\alpha} S'}{(n)S \xrightarrow{\alpha} (n)S'} \quad n \notin subj(\alpha)$$

(Congr)
$$\frac{S \equiv S' \qquad S' \xrightarrow{\alpha} S'' \qquad S'' \equiv S'''}{S \xrightarrow{\alpha} S'''}$$

(Catch)
$$\frac{R \xrightarrow{\langle t\rangle} R'}{[P,Q]_t \mid R \xrightarrow{\tau} Q \mid R'}$$

(Autoraising)
$$\frac{P \xrightarrow{\langle t\rangle} P'}{[P,Q]_t \xrightarrow{\langle t\rangle} Q}$$

(ScopeAlloc)
$$[P,Q].R \mid S \xrightarrow{\tau} [P\{t/\texttt{this}\},Q]_t \mid R \mid S \qquad t\ \textit{fresh}$$

(ScopeExtr)
$$[[P,Q].R \mid P',Q']_t \mid S \xrightarrow{\tau} [P\{t'/\texttt{this}\},Q]_{t'} \mid [R \mid P',Q']_t \mid S \qquad t'\ \textit{fresh}$$

(Scope)
$$\frac{P \xrightarrow{\alpha} P'}{[P,Q]_t \xrightarrow{\alpha} [P',Q]_t} \quad \alpha \neq \langle t\rangle$$

(NonLocal Com)
$$\frac{P \mid P' \xrightarrow{\tau} P'' \mid P'''\{\tilde{v}/\tilde{u}\}}{[P,Q]_t \mid [P',Q']_s \xrightarrow{\tau} [P'',Q]_t \mid [P'''\{\tilde{v}/\tilde{u}\},Q']_s}$$

Table 1
Labeled Transition System

5 Failures Handling Pragmatics

Mechanisms as Exception Handling, Nested Transactions with Compensation handlers and event handling can be programmed easily in this event based framework. In this section we shall show by examples how to program, in our language, the first two of these mechanisms while the third one is straightforward. At the end it should be immediate to figure out that implementing failure resilient business processes within this single error handling mechanism is both easy and effective.

5.1 Nested Transactions

In BPEL each transaction has a scope. In our language a transaction scope can be programmed using an event scope where the event handler represents the compensation handler. In this context, an `abort` can be programmed just as a `signal` directed to the scope itself. For example, a simple pattern is the following:

$$[\texttt{signal}(\texttt{this}), Q].\mathbf{0}$$

where the event handler Q has to provide the compensation mechanism. This mechanism must be decided solely by the programmer because is part of the business logic itself.

In our language we can program the compensation activation mechanism supported by BPEL where compensation handlers are invoked in the reversed order. In fact, it is possible to nest as many scope as we want providing an outer scope with the information about the scopes it contains using, e.g., a restricted channel for avoiding interferences. When the logic of the innermost event handler has been executed, it has to raise an event directed to the father. At that point, the father has to catch this event managing it with an adequate compensation handler which, at the end, has to raise the same event for its father and so on, propagating the signal until the root is reached. Let us consider the following basic pattern which allows the propagation of a signal from the inner transaction to the outer:

$$(x)([[signal(\texttt{this}), x(r).signal(r)].\mathbf{0} \mid \overline{x}\,\texttt{this}, Q].\mathbf{0})$$

In this example we have nested transactions where the inner transaction activates its handler which receive on a restricted channel the name of the outer transaction for activating its handler Q. It is straightforward to understand that, in this way, we can call the compensation handlers in the reversed order as in the usual semantics of nested transactions.

5.2 Exception Handling

The semantics of the exception handling mechanism `try P catch Q` can be programmed using event scopes as the scopes of exceptions. Usually, an exception is activated by a `thrown` primitive: `try P catch Q` means that, if a `thrown` is executed by P, then P is terminated and Q is performed. In our framework raising an exception inside the body of a scope means simply to signal an event to itself. The event will be triggered by rendezvous on a restricted channel. Here we program the `thrown` construct as $\overline{x}.x()$. The output $\overline{x}$ represents the trigger itself while the input $x()$ is necessary for blocking P avoiding an incorrect interleaving before the execution of $\texttt{signal}(\texttt{this})$. It is worth noting that the solution is correct only in the case P is a sequential process. The extension supporting parallel composition can be obtained simply by defining an event scope for each process composed in parallel and by implementing a synchronization among the event handlers in order to activate the exception handler. The pattern for the described behavior is as follows:

$$[(x)(P \mid x().\texttt{signal}(\texttt{this})), Q].\mathbf{0}$$

It is worth noting that nesting transaction allows us to catch more than one kind of exception.

6 Conclusions and Future Works

The specification of the BPEL language describes three different mechanisms for coping with abnormal situations: Exception Handling, Event Handling and Compensation Handling. All this complexity makes it difficult to formally define the language, thus limiting the formal reasoning about the designed applications. Having a small language instead makes it possible to accomplish the goal of formalization. In this paper we claim that three different mechanisms for error handling are not necessary and complicate too much the language definition. Thus we formalized a novel choreography language, extending the π-calculus, based on the idea of event notification as the only error handling mechanism. Our effort can be considered as a proposal of simplification for BPEL, thus advantaging implementors of the orchestration engine as well as application designers. In order to support this thesis, we presented some examples explaining how the mechanisms of BPEL can be programmed using our framework. Any formal comparison is left as future work.

We should underline the fact that even if our language tries to deal with the asynchrony of the communication typical of the context we refer to, the message passing mechanism is the one proposed in the synchronous version of the π-calculus. This is because we believe that asynchrony should be reflected at the level of event raising and not at the level of message passing. Anyway, it is straightforward to present an asynchronous variant of this language.

We considered that the proposed language shares some features with BPEL and StAC[3] and that they can all be viewed at the same level of programming abstraction. In the future we intend to investigate the expressivity of such proposals (such also the πt-calculus [2]) by exploiting our language that we consider flexible enough to encode, in a simple manner, the most interesting proposals available in the literature. We want to remark that a point particularly favorable to flexibility and elegance consists in the way in which we are identifying and handling dynamically the allocation of scopes. Our mechanism is an extension of the name mobility present in the $\pi - calculus$ in order to include also a runtime handling of scope identifiers. Other crucial problems we would like to tackle are verification and behavioral type systems [9]. It should be clear that the language we presented in this paper strongly commits to the π-calculus. This theory has been deeply investigated in the last years and it is widely recognized that the introduction of types for concurrency promise certain benefits in terms of program semantic analysis. In fact, a theoretical approach like the one we followed can lead to verifiers where the type system can check not just data types matching but also that a procedure using some channels (to send and receive data) is obeying a given protocol with these messages (and with these channels). Several works in literature showed that in this framework checking for critical properties in distributed programming is possible at compile time. Behavioral type systems for Process Algebras, in fact, permit us to ensure processes to have specific properties as being deadlock-free or obeying a particular protocol.

References

[1] Tony Andrews, Francisco Curbera et al. - *Businnes Process Execution Language for Web Services Specification, Version 1.1*, 5 May 2003.

[2] L.Bocchi, C.Laneve, G.Zavattaro - *A Calculus for Long Running Transactions.* FMOODS '03, Paris, December 2003. LNCS.

[3] Michael Butler, Carla Ferreira - *An Operational Semantics for StAC, a langage for Modelling Long-running Businness Transactions.* To appear in COORDINATION 2004.

[4] *Businness Process Modelling Language (BPML)* [www.bpmi.org].

[5] L.Cardelli, A.D.Gordon - *Mobile Ambients.* Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98.

[6] I.Castelani - *Process Algebras with Localities.* Handbook of Process Algebra - Edited by J.A. Bergstra, A.Ponse, S.A. Smolka - Elsevier 2001.

[7] E.Christenses, F.Curbera, G.Meredith, S.Weerawarana.
Web Services Description Language (WSDL 1.1) [www.w3.org/TR/wsdl], W3C, Note 15, 2001.

[8] F.Leymann - *Web Services Flow Language (WSFL 1.0)* [www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf].

[9] Atsushi Igarashi, Naoki Kobayashi - *A Generic Type System for the π-Calculus.* In Proceedings of ACM Symposium Conference on Principles of Programming Languages (POPL), 2001.

[10] Microsoft Corp. Biztalk Server - http://www.microsoft.com/biztalk.

[11] R. Milner - *Function as Processes.* Mathematical Structures in Computer Science, 2(2):119-141, 1992.

[12] Robin Milner, Joachim Parrow, David Walker - *A Calculus for Mobile Processes.* Journal of Information and Computation, 100:1-77. Academic Press, 1992.

[13] S.Thatte - *XLANG: Web Services for Businnes Process Design*
[www.gotdotnet.com/team/xml/wsspecs/xlang-c] Microsoft Corporation, 2001.

[14] World Wide Web Consortium (W3C) - http://www.w3.org.

[15] WS-Transaction Specification - http://www.w3.org.