Towards Semantics Driven Generation of Executable Web Services Compositions

Giusy Di Lorenzo, Nicola Mazzocca, Francesco Moscato,Valeria Vittorini
Dip.di Informatica e Sistemistica, Univ. of Naples Federico II
Email: {giusy.dilorenzo, francesco.moscato}@unina.it

Abstract— **Web services composition is a very active area of research due to the growing interest of public and private organizations in services integration and/or low cost development of value added services. The problem of building an executable web service from a service description has many faces since it involves web services discovery, matching, and integration according to a composition process.**

In this paper we propose a life cycle for the automated composition of web services which is based on the usage of Domain Ontologies for the description of data and services, and on workflow patterns for the generation of executable processes. In particular the paper focuses on the integration of the matching and composition phases. The approach aims at producing executable processes that can be formally verified and validated. This is achieved by exploiting formal definitions of composition rules and of BPEL4WS constructs. These definitions are expressed in operational semantics and are translated into Prolog programs in order to be throughout the composition process. A reference architecture for implementing the proposed life cycle is also described.

I. Introduction

The SOA (Service Oriented Architecture) foundation relies upon basic services, services descriptions and operations (publication, discovery, binding) [29]. One of the most promising benefits of SOA based web services is enabling the development of low cost solutions/applications by composing existing services. Web services composition is an emerging approach to support the integration of cross-organizational software components [9] whose effectiveness may be severely compromised by the lack of methods and tools to automate the composition steps. Given a description of a requested service and the descriptions of several available basic services, the ultimate goal is to be able:

a) to perform the automatic and dynamic selection of a proper set of basic services whose combination provides the required capabilities;
b) to generate the process model that describes how to implement the requested service;
c) to translate the process model into an executable definition of the services composition, in case the selection is successful;
d) to verify the correctness of the definition of the composition;
e) to validate the composite web service against the initial description.

Each of these steps has its intrinsic complexity. Services descriptions, relations among the involved data and operations, composition definitions should be unambiguously computer-interpretable to enable the automation of web services discovery, selection, matching, integration, and then the verification and validation of web services compositions [15], [22]. Most of the work on web services composition address these steps separately.

As for the description of web service capabilities and process model, the Semantic Web Community has created OWL-S [27], a web service ontology based on OWL which, in turn, is a logic based ontology language to describe web contents [21]. OWL-S has a well defined semantics and provides a starting point to automate tasks as discovery and composition [22]. The WSMO (Web Services Modeling Ontology) initiative has recently developed a conceptual framework and a formal language for “semantically describing all relevant aspects of web services in order to facilitate the automation of discovering, combining and invoking electronic services over the Web” [40]. These means provide the basis to reason about services integration in automated contexts. Nevertheless, they are not been proven expressive enough to solve the problem of generating executable compositions. For example, in [31] ad-hoc executable compositions (i.e., single-use cases) are described by means of OWL-S and executed by the Mindswap engine [23], while the OWL-S process model is shown to be not sufficiently expressive to characterize more general compositions (i.e., re-usable, multiple use-cases compositions). Ad-hoc composition and re-usable composition are also addressed in [25] and [6], respectively.

A different approach is based on orchestration languages. In particular, many industry efforts to build web services composition focus on the workflow to be realized to implement the requested service. The most popular flow language oriented to web services composition is the Business Process Execution Language for Web Services (BPEL for short). BPEL is now being standardized by OASIS (WSBPEL-TC); it enables the definition of business processes and interaction protocols to meet web services orchestration requirements [4].

The main drawback of the flow languages is their lack of formal semantics, hence the composition process described by means of these languages cannot be handled, queried and interpreted reliability by a computer based program. Several formalizations of BPEL have been proposed in the literature (e.g. in [10], [12]–[14], [26]), but at the best of our knowledge none of them is used for the automated development of web services compositions in working systems.

Moreover, automatic synthesis of composition processes is an hard problem to solve, both in theory and in practice. Partial solutions have been proposed, based on formal approaches (e.g. [2], [7]), on the peer-to-peer model [5], according to the semantic and workflow approaches [38], focusing on partial automation [30]. In particular, a technology for semi-automated composition of SOA components has been recently developed by IBM [1], [8], which uses AI planning-based techniques. One of the main difficulties to overcome is that dynamic composition of services requires component services to be compatible in order to achieve the composition goal. Matching algorithms, methods and tool have been developed aiming at selecting IOPE (Inputs, Outputs, Preconditions and Effects) compatible services [17], [18], [28], [35], but most of them really address only Inputs and Outputs matching since the specifications of precondition and effects is still an open issue of languages such as OWL-S [16].

Finally, very few work has been done to validate the results of the composition. This phase can be accomplished by means of formal techniques only if both the composition goal and the composite service capabilities may be formally expressed. Of course, the validation may be more easily performed if all the phases of the composition development process are supported by formal means.

In this paper we present a first step towards the definition of an unifying life cycle approach to web service composition development. The development process may start from a composition goal that describes a specific customer's request (ad-hoc request) or also from a composition goal that requires to build a web service able to satisfy a class of requests. The same objective (automatic generation of executable compositions) is pursued in [31] which investigates the integration of discovery/selection of compatible services and composition.

We propose a life cycle which uses domain ontologies to describe operations, data and services, and aims at producing BPEL executable processes that can be formally verified and validated. Operational semantics is the formal basis of all the life cycle phases: it is used to express the flow of the operations which realizes the composition goal, to identify the composition pattern described by the composition flow and automate its translation into a BPEL executable process, to formalize the BPEL constructs (so that the resulting executable process can be automatically verified), and to support the validation of the composition. We also define a reference architecture for implementing the proposed life cycle, whose components are partially developed [10], [24] or under development. An example of composition is described, in order to prove the effectiveness of the approach. Much more work has to be done on the theoretical framework and on the single steps of the composition, also integrating the results available in the literature. Non-functional aspects, such as quality of service and security requirements must also be considered [34], [42]. We want to show that an unified approach to the development phases of composite services can be investigated and successfully used.

The remainder of the paper is organized as follows. Section II contains an overview of the proposed life cycle. Section III focuses on the selection of candidate services and the generation of the process model of the composition. In Section IV an architecture is described which supports the generation of an executable composition according to the phases described in Section II and III. In Section V the proposed approach is applied to a case study. Finally Section VI contains some closing remarks and some hints about future work.

II. LIFE CYCLE

In this Section we describe the phases to automatically develop an executable web service composition from an initial specification of the requested service (*Request*, in the following). The proposed approach addresses the generation of both ad-hoc and re-usable compositions. Nevertheless, the discussion is here focused on the life cycle of re-usable compositions. The approach is based on the hypothesis that the Request refers to a well defined Domain and that a detailed Domain Ontology is available or can be created. The Domain Ontology must contain all the concepts (data and operations) to form the Request in a common (or accepted) Domain vocabulary. The approach exploits OWL for data and OWL-S for operations; here an operation is an atomic function described by the OWL-S Service Profile and Grounding[1]. Hence the proposed composition development process takes into account the fact that a service may provide more operations. The life cycle described in the following requires that a Knowledge Base $\mathcal{KB}$ and a set of Inference Rules $\mathcal{IR}$ are defined. A formal, explicit description of the Domain is given by OWL and OWL-S. The components of $\mathcal{KB}$ are axioms, derived from the ontology descriptions of the Domain and expressed by means of the Prolog language. They describe the properties of concepts and the relationships (*constraints*) among concepts. In particular, the relations among data and operations involve the semantic description of the operations, Pre-conditions and Effects, and the Input/Output of the operations. The creation of $\mathcal{KB}$ has to be performed once but it is specific to each Domain. The rules to reason on $\mathcal{KB}$ are expressed by an operational semantics and then written in Prolog. $\mathcal{IR}$ contains the rules to apply in order to build valid paths of operations in terms of IOPEs and according to the main workflow operators: sequence, AND, OR, XOR splits and joins. These inference rules are re-usable and they do not depend on the Domain.

Given an ontology description of the Domain, $\mathcal{KB}$ and $\mathcal{IR}$, the main phases to generate an executable Web Services composition according to the proposed approach are:

[1]The Service Profile says what an operation does by specifying its IOPE, and Grounding is a mapping from OWL-S to WSDL [35].

- Synthesis of the Operations Flow model ($\mathcal{OF}$);
- Generation of the Services Workflow model ($\mathcal{SW}$) from $\mathcal{OF}$;
- Synthesis and Verification of the executable process implementing $\mathcal{SW}$;
- Validation of the composite service.

They are briefly described below. Section III will detail the Synthesis of $\mathcal{OF}$ and the generation of $\mathcal{SW}$.

a) **Synthesis of the Operations Flow model.:** The generation of an executable Web Services composition is issued by a *Request*. The goal of this phase is to obtain a graph model of the flow of the operations that must be executed to satisfy the Request. This is accomplished according to the following steps:

a) Analysis of the Request. We suppose that the user's request contains a description of the service $\mathcal{W}$ to be provided, expressed by using the Domain Ontology concepts and relations. This description must specify the semantics of $\mathcal{W}$, its IOPE parameters and the conditions that it must verify. The conditions may be defined by means of the logical connectives of the propositional calculus. From this information, a Prolog Query $PQ_{\mathcal{W}}$ is generated.

b) Generation of the Operations Flow model. $PQ_{\mathcal{W}}$ is issued to determine possible operations flows that satisfy the *Request* (if any). In this phase operations compatibility is exploited by matching Pre-conditions and Effects and the inference rules in $\mathcal{IR}$ are applied to build $\mathcal{OF}_{\mathcal{W}}$. Input and Output compatibility is addressed in the next phase.

b) **Generation of the Services Workflow model.:** The goal of this phase is to transform the Operations Flow model in order to obtain a workflow model $\mathcal{SW}_{\mathcal{W}}$ of the composition. In this phase graph transformation techniques are applied and the workflow patterns which realize the service are identified. Notice that the operations specified by the flow model must be mapped to the available Web Services. We cope with I/O compatibility by using wrapping services, that can be involved in the composition if it is necessary.

c) **Synthesis and Verification of the executable process.:** The goal of this phase is to generate an executable and correct orchestration process from the workflow model. In particular, we address the automatic generation of BPEL processes from workflow models. This is achieved by exploiting previous results, in particular the automatic generation of a BPEL definition of the composition from $\mathcal{SW}_{\mathcal{W}}$ is performed by using the results described in [24], and the verification of the BPEL process is accomplished by using the results described in [10]. These works are based on a formalization of the control flow constructs of BPEL by means of operational semantics and Prolog.

d) **Validation of the composite service.:** The goal of this phase is to validate the composite service against the composition goal expressed by the Request. Since the overall development process is based on Prolog language, it is possible to validate the composite process by standard validation procedure, i.e. by trying to obtain the Request from the composite service.

III. From the Request to the Process Model

A. Synthesis of the Operations Flow model

In this phase a set of operations is selected and properly combined in order to produce the semantic behavior of the requested service.

According to the OWL-S standard definition, a web service operation is defined by its Inputs (*I*) and Outputs (*O*) parameters, and by its Pre-conditions (*P*) and Effects (*E*)(IOPE model) [28].

The synthesis of the operations flow model is achieved by analyzing the Request and the OWL-S definitions of the operations, and then issuing a Prolog query on $\mathcal{KB}$.

The inference rules in $\mathcal{IR}$ defines services compatibility in terms of pre-conditions and effects. Four possible types of semantical matchings can be exploited during the analysis of concepts in $\mathcal{KB}$ in order to establish operations compatibility:

- **Perfect Matching**: predicate concepts are the same;
- **Exact**: predicate concepts are equivalent;
- **PlugIn**: a predicate concept is a subconcept of another one;
- **Subsume**: a predicate concept is a superconcept of anothe one;
- **Fail**: no matchs among predicate concepts.

The analysis of inferential tree for retrieving the requested service leads to composite services. Compositions are defined in terms of precedence relations among operations. These relations define a graph that we call *Operation Flow* graph.

The workflow constructs we use to build OF are the following: *sequence*, *split* and *join*. Sequence allows for sequential activation of operations; split and join allows for concurrent execution of operations and synchronization. Choices and loops can be introduced by means of proper conditions on OF edges.

In the following the PE matching and flow rules are formally defined.

We remember that *P* and *E* are sets of predicates, as explained in Section II. Let $Predicate$ be the set of all predicates that appear in *P* and *E* sets of all operations in the domain and let σ be the set of evaluations of all predicates in $Predicate$ ($eval(Predicate)$ for short's where $eval$ is the function that associates to each element in the set $Predicate$ the couple $(predicate, value)$), where $value$ is the truth value of $predicate$).

In the following P_A (E_A) will denote the preconditions (effects) set associated to an operation A. In order to allow for A operation activation (i.e. to make A $activable$), all predicates in P_A must evaluate true. After a correct termination of the A operation, predicates in E_A evaluates true. In addition we call Act_A the activation of the operation A. Finally we indicate with Eff_A the set of all $Predicates$ that evaluates true after the Act_A:

$$Eff_A = \{p \in Predicates | eval(p) = (p, true) after Act_A\}$$

In the following the operational semantics of the *rules* we introduced before is reported. These definitions are translated into Prolog rules which are used during the synthesis phase. *Rules* are defined in terms of precondition that can enable the activation of a given operation composition and in terms of the changes in the σ set depending on *E* sets of component operations and composition operators.

The semantics of the activation of an operation A is the following:

$$\frac{eval(P_A)}{\sigma_A \overset{Act_A}{\longrightarrow} \sigma_A^{'}} \tag{1}$$

where $\sigma_A = eval(Predicate)$ before the activation of the operation A and $\sigma_A^{'}$ is the same $eval(Predicate)$ but after the activation of the operation. Notice that only predicates in E_A may change their evaluation, and then $\sigma_A^{'} = eval(\neg(Predicate \cap E_A) \cup E_A)$

In order to synthesize the requested composed service all possible combinations of services are analyzed by the means of a Prolog engine, that tries to find a services composition to which corresponds the requested *P* and *E* sets.

Proper pruning techniques are used to allows for termination of the inferences even when loops are created in the problem state space exploration.

1) Sequence: In a sequence, an operation is activable after the completion of another operation in the same process. Let A and B two web services operations where the B operation can be activated only after the completion of the A operation. We denote with $Seq(A,B)$ the sequential activation of A and B where the activity order inside the brackets is related to the activation order.

In order to activate the Sequence, the following conditions must happen:

$$\begin{gathered} Eff_A \supseteq P_B, eval(p) = (p, true) \\ \forall p \in P_A \longrightarrow activable(Seq(A,B)) \end{gathered}$$

Obviously $P_{Seq} = P_A$ and $E_{Seq} \supseteq E_B$ because E_{Seq} contains all predicates in the last sequence operation, but also predicates that belong to the E sets of the other operations which maintain their truth values during the execution of the whole sequence (i.e., in sequence with two operations, the $p \in E_A$ such that $eval(p) = (p, true)$ even after the B execution). It is possible to prove that $E_A \cup E_B \supseteq E_{Seq}$. The relation between the sets is not an equality since it the B operation should request the invalidation of a previous effect. For example a service that first request an authentication for a session, can also request the end of the authentication session after the execution of a given task. The predicate *hasAuthentication* is an effect of the first operation, but not of the last one (and thus it is not one of the sequence).

It is also true that $Eff_{Seq} = Eff_B$.

Notice that associative property can be applied to Sequence operator, and it is possible to state that $Sequence(A, Seq(B,C)) = Seq(Seq(A,B),C) = Seq(A,B,C)$. In the case of multiple sequence component operations, the previous definition can be extended by recursion.

If L_n denotes a list of n operations $L_n = (A_1, \cdots, A_n)$ to execute in a sequence then:

$$\begin{aligned} &Seq(L_n) = Seq(Seq(L_{n-1}), A_n) \\ &Seq(A) = A \end{aligned}$$

The semantics of a sequence composition is the following one:

$$\frac{\begin{gathered}\sigma_{Seq(L_{n-1})} \overset{Act(Seq(L_{n-1})}{\longrightarrow} \sigma_{Seq(L_{n-1})}^{'}, \\ Eff_{Seq(L_{n-1})} \supseteq P_{A_n}, \sigma_{Seq(L_{n-1})}^{'} \overset{Act(L_n)}{\longrightarrow} \sigma_{Seq(L_n)}^{'}\end{gathered}}{\sigma_{Seq(L_n)} \overset{Act(Seq(L_n))}{\longrightarrow} \sigma_{Seq(L_n)}^{'}} \tag{2}$$

(1) and (2) are the rules which define the execution of sequential web services operations. They state that in order to allow for sequential execution of a list of operations, the last one (A_n) has to be activable and the other ones have to be previously activated. In order to allows for last operation activation, it must be $Eff_{Seq(L_{n-1})} \supseteq P_{A_n}$.

For example, the rules are recursively applied for $Seq(A,B)$ in the following manner:

$$\frac{\dfrac{eval(P_A)}{\sigma_A \overset{Act(A)}{\longrightarrow} \sigma_A^{'}}, Eff_A \supseteq P_B, \dfrac{eval(P_B)}{\sigma_a^{'} \overset{Act(B)}{\longrightarrow} \sigma_{Seq(A,B)}^{'}}}{\sigma_{Seq(A,B)} \overset{Act(Seq(A,B))}{\longrightarrow} \sigma_{Seq(A,B)}^{'}}$$

Notice that $Eff_A \supseteq P_B \Longrightarrow eval(P_B)$, $\sigma_{Seq(A,B)} = \sigma_A$ and $\sigma_{Seq(A,B)}^{'} = \sigma_B^{'}$.

2) Split: A Split is a point in the OF with a single incoming control flow path and multiple outgoing paths (Fig. 1). Three types of splits are defined in order to describe different kind of outgoing paths executions: AND, XOR and OR splits.

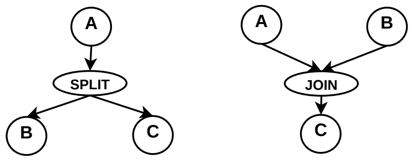

Figure 1. Split and Join

AND splits allow for parallel execution of outgoing paths. All preconditions needed to enact all paths executions have to be verified in order to consider the OF activable. XOR splits allow for the enactment of only one of the outgoing paths. The preconditions of only one outgoing path have to be be verified in order to consider the OF activable. If more paths can be enabled, the system must choose one to activate. OR splits allow for the enactment of one or more of the outgoing paths at any time, as soon as paths enabling preconditions are evaluated true. Obviously any type of split with one outgoing edge has to be considered a sequence. For brevity's sake we will show in the following only the AND split activation conditions and semantics. Complex splits can be achieved by associating predicates called *conditions* on each outgoing path and discriminating their activation depending on these predicates values.

Let us consider the AND split in Fig. 1 and let us indicate a split with $Split^{AND}_{(A,B,C)}$ where the first operation in the brackets is related to the incoming split path and the other ones to the outgoing paths . The condition that makes activable the AND split is the following one:

$$\begin{gathered} Eff_A \supseteq P_B \cup P_C \\ eval(p) = (p, true) \forall p \in P_A \end{gathered}$$

Let us consider for simplicity's sake (here and in the other following constructs) that the E sets of split outgoing operations have no predicates that appear in one of the other set in the negate form. We can say that $P_{Split^{AND}_{(A,B,C)}} = P_A$ and, $E_{Split^{AND}_{(A,B,C)}} = E^*_A \cup E_B \cup E_C$, where E^*_A is the set of E_A predicates which do not appear in a negative form in E_B and E_C . The equality is because we assume that parallel outgoing operations cannot act concurrently executing conflicting operations.

More generally we denote with Inc the operation on the incoming path and with $Out_n = (O_1, \cdots, O_n)$ the operations on the outgoing paths, indicating the split with $Split^{AND}_{(Inc,Out_n)}$. Let be $Out_{n-1} = (O_1, \cdots O_{n-1})$, We can thus recursively define $Out_n = (Out_{n-1}, O_n)$, with $O_0 = \emptyset$ and $Split^{AND}_{(Inc,\emptyset)} = Inc$

Since we assume no conflicts on outgoing operations, the AND Split operator can be considered commutative on Out_n list. Thus if we denote with $Perm(Out_n)$ the set of all possible permutations on Out_n list, and with $Perm_i$ an element of this set,

$$Split^{AND}_{(Inc,Out_n)} = Split^{AND}_{(Inc,Perm_i)} \forall i \in \{1, n!\}$$

It is possible to associate the position in the list O_n with the order of operation completion. With the previous relation we state that the AND Split execution is independent on the completion order of outgoing operations.

It is now possible to describe the semantics of the AND Split:

$$\frac{Eff_{Inc} \supseteq P_{O_{n-1}}, \quad \sigma_{Split^{AND}_{(Inc,Out_{n-1})}} \overset{Act(Split^{AND}_{(Inc,Out_{n-1})})}{\longrightarrow} \sigma'_{Split^{AND}_{(Inc,Out_{n-1})}}}{\sigma_{Split^{AND}_{(Inc,Out_n)}} \overset{Act(Split^{AND}_{(Inc,Out_n)})}{\longrightarrow} \sigma'_{Split^{AND}_{(Inc,Out_n)}}}$$

Notice that $\sigma'_{Split^{AND}_{(Inc,Out_{n-1})}}$ is the set of all predicates evaluations of all the Split operations evaluations except the operation O_n. This may resume the case of having all operations terminated but the O_n. Thanks to the commutative property previously described, this is not a loss of generality and the rule can be applied independently on outgoing operation terminations: the final state will be $\sigma'_{Split^{AND}_{(Inc,Out_n)}}$ in every case.

Furthermore,

$$\begin{gathered} Eff_{Inc} \supseteq \bigcup_{i=1,\cdots,n} P_{O_i} \Rightarrow \\ Eff_{Inc} \supseteq P_{O_i} \forall i \in \{1, \cdots, n\} \end{gathered}$$

and the precondition $Eff_{Inc} \supseteq P_{O_{n-1}}$ is true at any level of inference tree.

We do not report the rules of other Split types due to the lack of space.

3) Join: A Join in the OF is a point with multiple ingoing paths and a single outgoing path (Fig. 1). It is usually a synchronization point of concurrent or parallel activities. Three types of joins are defined in order to describe different kind of synchronization: AND; XOR and OR joins.

An AND join is a point where all operations on incoming paths have to terminate their execution in order to activate the outgoing control flow path. An XOR join allows for activation of the outgoing path whenever the operations of one of the incoming paths terminate their executions. Finally an OR join allows for activation of the outgoing path every time an incoming path operations terminate; the outgoing path can be activated more than once. In addition complex synchronization patterns can be defined by associating predicates (*conditions*) on incoming paths. The joins rules in such case apply only to paths with conditions that evaluate true. In the following for brevity's sake, we will describe only AND Join semantics.

With reference to the Fig. 1 we denote with $Join^{AND}_{(A,B,C)}$ the activity with operation A and B on incoming join path and with operation C (the last in the join list) on the outgoing path. The conditions that make activable the AND join are the following:

$$\begin{gathered} Eff_A \cup Eff_B \supseteq P_C \\ eval(p) = (p, true) \forall p \in P_A \cup P_B \end{gathered}$$

In order to activate the C operation, the paths with A and B operations must terminate. This implies that join can be activated only when all precondition of operations on incoming paths evaluate true and it is possible to state that: $P_{Join^{AND}_{(A,B,C)}} = P_A \cup P_B$. Further, if no conflicting operations on incoming paths, it can be trivially proved that $E_{Join^{AND}_{(A,B,C)}} = E^*_A \cup E^*_B \cup E_C$ where E^*_X is the set of all E_X predicates except for predicates that appear in the negative form in E_C. The definitions can be adapted to a generic number of operations on incoming paths (in a list Inc_n) and one on outgoing path (Out) like as we did previously with Split.

In order to define the semantics of Join, let us extend the semantics of the Act function to a list of operations. Let be Inc_n a list of n operations $(I_i, \cdots, I_n)$. We can extend the Act semantics as follows:

$$\frac{\sigma \xrightarrow{Act(Inc_{n-1})} \sigma'_{n-1}, \sigma'_{n-1} \xrightarrow{Act(I_n)} \sigma'}{\sigma \xrightarrow{Act(Inc_n)} \sigma'}$$

Where σ is the state before the activation of all operations, σ' is the state after the activation of all operations and σ'_i is the state after the activations of the first i activities in the Inc_n list. In brief the activation of n operations in a list evolves by activating component operations in turn.

With this definition and thanks to the commutative properties on incoming paths operations, it is possible to define the semantics of a Join:

$$\frac{\sigma_{Join^{AND}_{Inc_n,Out}} \xrightarrow{Act(Inc_n)} \sigma'_{Inc_n}, \quad \bigcup_{i\in\{1,\cdots,n\}} Eff_{I_i} \supseteq P_{Out}, \sigma'_{Inc_n} \xrightarrow{Act(Out)} \sigma'_{Join^{AND}_{(Inc_n,Out)}}}{\sigma_{Join^{AND}_{(Inc_n,Out)}} \xrightarrow{Act(Join^{AND}_{(Inc_n,Out)})} \sigma'_{Join^{AND}_{(Inc_n,Out)}}}$$

B. Generation of the Services Workflow model

Since the previous synthesis process does not take in account of operation I/O descriptions, the OF graph may generate compositions that, even if semantically correct, may be incorrect in terms of Input and Output. At this purpose, we introduce in the OF the execution of operations that execute I/O format translations (*wrappers*). In order to establish if a wrapper has to be inserted in the OF, a proper I/O matching algorithm for operations is used.

Once the rules explained before have been applied in order to build the OF graph, it is necessary to translate this representation into a control flow graph which elements are organized in workflow patterns [33], [41]. We will call this graph *Service Workflow* graph ($\mathcal{SW}$).

The main concepts of the translation from OF to SW will be introduced in the following.

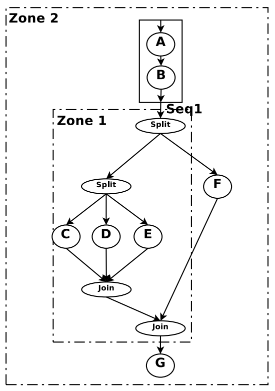

Figure 2. OF Zones

Let us consider the OF in Fig.2. It is composed by two operations in sequence (A and B in Seq_1) and two parallel paths after an AND split point, that synchronize in an AND join point. The path on the right contains a single operation while the path on the left is in turn a splijoin composed by three parallel branches (each containing a single operation).

The first step of the translation algorithm consists in identifying OF graph regions we call *zones*. Briefly each zone is identified by a single split-join and by activities that comes before and after it.

Zones are identified by visiting the OF graph with a depth first policy, identifying zones recursively also by using techniques of compiler optimization and program synthesis [3], [11]. The graph may contain loops that will be addressed in future works.

In Fig.2 it is possible to identify two zones ($Zone_1$ and $Zone_2$, where the first is included in the second one)

Depending on split and join types (AND, XOR, OR) in a zone, and on which kind of conditions are defined over the edges in OF graph, it is possible to translate a zone subgraph in a component of SW (which obviously is a workflow pattern).

Table I shows the relations among some workflow patterns, split-join types and conditions. Notice that the *Sequence* pattern is trivially identified on OF Graph and translated.

Pattern	**Split**	**Join**	**Conditions**
Parallel Split	AND	-	No
Synchronization	-	AND	No / Yes iff they are the same on all join incoming edges
Exclusive Choice	XOR, OR	-	Mutually exclusive on all split outgoing edges
Simple Merge	-	XOR	No
Multi Choiche	OR	OR, XOR	On split outgoing edges
Structured Synchronization Merge	*	AND	Yes
Multiple Merge	*	OR	Yes
Structured Discriminator	*	XOR	Yes
Interleaved routing	AND	AND	No
Interleaved Parallel Routing	*	AND	No
Deferred Choice	OR	OR, XOR	Yes with exeternal event (trigger) specification

TABLE I.
SW RULES

A workflow patterns description can be found in [33], [41]. Fig.3 depicts the SW related to the OF in Fig.2.

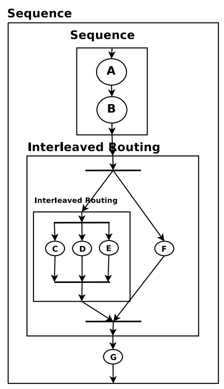

Figure 3. An Example of SW Model

The SW is a composition of sequences and interleaved routing patterns. The whole composition must be included in a sequence pattern.

The composition of WF patterns will be used in the next phases in order to produce, verify and validate executable composed processes. The executable processes are defined in BPEL4WS language as described before.

C. Synthesis and Verification of the executable process

In the phase of the executable process synthesis the $\mathcal{SW}$ is analyzed in order to build a BPEL process ($\mathcal{BP}$). The $\mathcal{BP}$ is then processed in order to verify its correctness.

Both synthesis and verification phases are based on a formal definition of BPEL4WS constructs in terms of operational semantics. The operational semantics is translated into Prolog rules (*BPEL Semantic Rules*) which are used as explained in the following.

1) BPEL4WS basics: BPEL4WS defines a language for specifying Web Services compositions in the form of business processes. In the following we refer to the BPEL4WS specifications published in May 2003 [4].

BPEL4WS (or BPEL for short) provides a model and an XML-based grammar for describing the control logic and the message exchanges required to coordinate the Web Services involved in a composition. A BPEL composition is called **process** and the web services participating in the process are called **partners**.

Once the partners are defined, how messages are exchanged and how to sequence the operations is specified by means of a set of primitives called **activities**. BPEL provides *basic* and *structured* activities.

Basic activities are used to define interactions with something external to the process itself. They mainly are instructions that handle the communication between the process and the partners. A typical scenario is the following.

The BPEL executable process waits a message from a requestor (which represents a client) by using the **receive** activity, when the message arrives an instance of the process is created. Then the process instance might executes one or more partner by means of the **invoke** activity and finally it sent back a response to the requestor by using the **reply** activity.

Since an executable process may specify more than one *receive* activity, it is necessary to distinguish between the messages received from a partner during a conversation. This is done by a mechanism called **correlation**.

Data flow is realized by sharing data containers that may exchange data using the **assign** activity to copy internal variables.

Structured activities are used to manage the process flow. They can be combined to write complex algorithms:

- **sequence**: specifies an ordered sequence of activities;
- **while**: defines a loop;
- **pick**: captures events (messages) in order to execute one of several alternative paths;
- **switch,case,otherwise**: defines a conditional branching;
- **scope**: groups a set of activities in a single transaction;
- **flow**: specifies a set of activities to be executed in parallel. The **link** construct may be used to define precedence relations among activities executing in parallel. Particular conditions (called transition conditions) may be defined over links. A link produces the activation of an activity only when its transition condition evaluates true.

Each activity has optional standard attributes: a name, a join condition, and an indicator whether a join fault should be suppressed if it occurs. A join condition is a Boolean expression on all incoming links of an activity and is used to specify requirement about concurrent paths reaching at an activity.

2) BPEL4WS Semantics: In the following the state of an activity a will be denoted by the symbol σ_a and the state of a link k will be denoted by the symbol σ_{L_k}.

Furthermore we will denote structured activities by:

$$A = \top s_0 a_1 s_1 \cdots\cdots s_{N-1} a_N s_M \bot$$

where:

- $\top$ activity is the activity that precedes the construct in the process,
- $\bot$ activity specifies that the construct is terminated and no more activities have to be processed within the construct,
- the symbols $s_i \in S$ denote a construct;
- $a_i \in L_A$ is the list of the activities specified within the construct.

The state σ_a of the basic activity a, can be:

- $ready$: a is ready to start;
- $exec$: a is started;
- $terminated$: a is ended; depending on the cause of the termination, this state can assume the values:
  - $noexec$: if the activity completes correctly without faults;
  - $undefined$: if the activity is terminated in an abnormal way or if it generates a not handled fault.

Notice that the state of a structured activity depends on the states of its component activities;

The state of a link k (σ_{L_k}) can assume the following values:

- $undefined$: the state of the link before the evaluation of its *Transition Condition*;
- $positive$: if the *Transition Condition* associated to the link is evaluated *true;*
- $negative$: if the *Transition Condition* associated to the link is evaluated *false*;

3) Derivation Rules: In the following the operational semantics for some of the BPEL language constructs are reported.

For brevity's sake in this paper we will only report the semantics of the sequence and flow (with links) constructs. The sequence is simple enough to let us explain how the semantic rules are defined and derived; flow with links is complex enough to describe the semantics of complex BPEL processes and to make possible the definition of a non trivial example.

In order to define the semantics of the BPEL constructs, it is needed to introduce some rules that are related to constructs but not explicitly provided by the BPEL language.

Implicit constructs

Let us introduce two basic transitions:

$$\frac{}{\langle a, \sigma_a = ready \rangle \xrightarrow{\mu} \langle a, \sigma'_a = exec \rangle}$$

$$\frac{}{\langle a, \sigma_a = exec \rangle \xrightarrow{\tau} \langle a, \sigma'_a \in terminated \rangle}$$

the μ and τ that respectively enable the execution and the termination of an activity:

Now let us introduce two operators frequently used in workflow languages: *split* and *join*. *Split* operator is used, when an activity terminates, to choose the next activity that may be activated depending on some boolean *conditions* defined over outgoing transitions. *Join* operator is similar to *split* operator but it applies to incoming transitions. It is important to notice that BPEL does not support explicitly these operators, but their behaviors are obtained by using links. For these reasons, here we define the semantics rules for *join* and *split* operator.

As for the join operator, let L_A be a list of activities, in the $ready$ state, to analyze and L_L the list of the links defined in the process.

In the following, the recursive definition of L_A list will be used: *a list is either the empty list, or it is a head followed by a tail, where the head is an element and the tail is a list*. With this definition, it is possible to define the following rule:

$$\frac{}{L_A \overset{first}{\rightarrow} l_H \cdot L_T}$$

The $first$ transition extracts the first activity l_H from the list of the activities L_A. The L_T list is composed by the remaining activities (the tail).

With this definition it is possible to introduce the basic rules for join semantics:

Rule 1 (Join1):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \dfrac{\left\langle a_H, L_L \right\rangle \overset{vC}{\rightarrow} true}{\sigma_{a_H} \overset{\mu}{\rightarrow} \sigma_{a_H}^{exec}}}{\left\langle L_A, L_L, \epsilon_A, \sigma_A \right\rangle \overset{join}{\rightarrow} \left\langle L_{A_T}, L_L, a_H^{exec}, \sigma'_A \right\rangle}$$

A join of some activities in the state σ_A is enacted, only if for at least one of the activities inner the L_A list has been determinated the status of all its incoming links and the Join condition can be evaluated. This check is performed by the rule vC that is omitted for brevity's sake. If the evaluation is $true$ in rule **Join1**, the join operator, thought the μ transition, allows the activation of the activity; changing the state σ_A into σ'_A that is the same of the state σ_A except for the state of the component activity $\sigma_{a_H}^{exec}$ that becomes *exec*.

If the join condition is $false$ the **Join2** and **Join3** rules must be applied and the sJF value is evaluated. sJF is used to establish if the *death-path-elimination* must be applied or if a standard fault must be propagated. *Death-path-elimination* does not allows the activation of other activities which are on the same path of the activity whose join condition evaluates false, assigning a negative status to its outgoing links.

Rule 2 (Join2):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \dfrac{\left\langle \sigma_{a_H}, L_L \right\rangle \overset{vC}{\rightarrow} false, sJF{=}no}{\sigma_{a_H} \overset{\mu}{\rightarrow} \sigma_{a_H}^{undefined}}}{\left\langle L_A, L_L, \sigma_A \right\rangle \overset{join}{\rightarrow} \left\langle L_{A_T}, L_L, \sigma'_A \right\rangle}$$

Rule 3 (Join3):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \dfrac{\left\langle \sigma_{a_H}, L_L \right\rangle \overset{vC}{\rightarrow} false, sJF{=}yes}{\sigma_{a_H} \overset{dPE}{\rightarrow} \left\langle \sigma_{a_H}^{undefined}, L_L^{New} \right\rangle}}{\left\langle L_A, L_L, \sigma_A \right\rangle \overset{join}{\rightarrow} \left\langle L_{A_T}, L_L^{New}, \sigma'_A \right\rangle}$$

The **Join3** rule handles join failure suppressions, performing death path elimination while the **Join2** rule put the activity with false transition condition into an undefined state.

The $Split$ operator is similar but concerning the activation of outgoing links of terminated activities and is omitted for brevity's sake. Another important construct, that we have defined to handle the termination of the activities in BPEL is the $TauCostruct$ transition. The semantics of $TauCostruct$ transition is:

Rule 1 (Tau 1):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \dfrac{\sigma_{a_H} \overset{tau}{\rightarrow} \sigma_{a_H}^{noexec}}{\left\langle a_H, L_L \right\rangle \overset{Split}{\rightarrow} L_{L_{New}}}}{\left\langle L_A, L_L, \sigma_A \right\rangle \overset{TauCostruct}{\rightarrow} \left\langle L_{A_T}, L_{L_{New}}, \sigma'_A \right\rangle}$$

Rule 2 (Tau 2):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \dfrac{\sigma_{a_H} \overset{tau}{\rightarrow} \sigma_{a_H}^{undefined}}{\left\langle a_H, L_L \right\rangle \overset{Split}{\rightarrow} L_{L_{New}}}}{\left\langle L_A, L_L, \sigma_A \right\rangle \overset{TauCostruct}{\rightarrow} \left\langle L_{A_T}, L_{L_{New}}, \sigma'_A \right\rangle}$$

L_A is the list of activities to analyze, L_L is the list of the **links** and $L_{L_{New}}$ is the list of links activated after a $split$ operation. A **TauCostruct** transition is activated when at least one activity is no more in execution and applies the split transition to a_H outgoing links to verify if some other activity can be execute. This operation changes the state of the process links and may activate some other activities. Since this transition terminates an activity, its state becomes $noexec$, if the activity termination is normal, $undefined$ otherwise. The state of process activities changes from σ_A to σ'_A where σ'_A is the same of σ_A except for the state of the activity σ_{a_T} (because the τ transition changes the state of the a_T activity).

It is now possible to introduce the sequence and flow constructs:

Sequence

Let be :

$$S = \top \cdot a_1 \cdot a_2 \cdot a_3 \cdots a_n \cdot \bot$$

the definition of a sequence of activities a_i. $\top$ activity is the activity that precedes the sequence in the process and the $\bot$ activity specifies that S is terminated and no more activities have to be processed.

Let L_A be the activity list of S and σ_s the state of S. A sequence activity may be in execution (state $exec$) or not (state $noexec$). Let be σ_a the state of the a activity in the sequence ($exec$ or $noexec$). The state of the sequence depends on the state its component activities:

$$\begin{aligned}\sigma_s &= exec \Leftrightarrow \forall i,j \in \{1,\cdots,n\}\sigma_{a_i} = exec, \sigma_{a_j|j\neq i} = ready;\\ \sigma_s &= noexec \Leftrightarrow \forall i \in \{1,\cdots,n\}\sigma_{a_i} = noexec\end{aligned}$$

The operational semantics specification of the *sequence* construct consists of the following four rules:

Rule 1 (S1):

$$\frac{\sigma_S = ready, L_A \overset{first}{\rightarrow} \top \cdot L_{A_T}, \sigma_S \overset{\mu}{\rightarrow} \sigma_S^{exec}}{\left\langle L_A, \sigma_S \right\rangle \overset{sequence}{\rightarrow} \left\langle L_{A_T}, \sigma_S^{exec} \right\rangle}$$

This rule applies when the sequence is not started yet. In such case the first activity of the sequence is the $\top$ activity. The *first* transition put in the L_{A_T} list the remaining sequence activities and the state of the sequence through the application of the μ becomes $exec$. The new state of the sequence is then σ_S^{exec} and the remaining activities to process (L_{A_T}) were pruned of the $\top$ activity.

Rule 2 (S2):

$$\frac{\sigma_S = exec, L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \sigma_{a_H} \overset{\mu}{\rightarrow} \sigma_{a_H}^{exec}}{\left\langle L_A, \sigma_S \right\rangle \overset{sequence}{\rightarrow} \left\langle L_A, \sigma'_S \right\rangle}$$

This rule applies when the sequence is already started and the *first* activity in the sequence is executed:

Rule 3 (S3):

$$\mathbf{S3.1}\ \frac{\sigma_S = exec, L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \sigma_{a_H}^{exec} \overset{\tau}{\rightarrow} \sigma_{a_H}^{noexec}}{\left\langle L_A, \sigma_S \right\rangle \overset{sequence}{\rightarrow} \left\langle L_{A_T}, \sigma'_S \right\rangle}$$

$$\mathbf{S3.2}\ \frac{\sigma_S = exec, L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \sigma_{a_H}^{exec} \overset{\tau}{\rightarrow} \sigma_{a_H}^{undefined}}{\left\langle L_A, \sigma_S \right\rangle \overset{sequence}{\rightarrow} \left\langle L_A, \sigma'_S = undefined \right\rangle}$$

In this case the activity a_H is terminated and its state can evolve:

1) from $exec$ to $noexec$, if the termination is normal;
2) from $exec$ to $undefined$, if a fault has occurred.

With this rule the *first* activity processes the activity a_H and the L_A list is pruned of this activity. The remaining activity list is called L_{A_T}. Notice that, in the first case the list of activities to process remains unchanged and only the sequence of states is changed; in the second case the $sequence$ termination is abnormal and its state is $undefined$.

Rule 4 (S4):

$$\frac{\sigma_S = exec, a = \bot, \sigma_S \overset{\tau}{\rightarrow} \sigma_S^{noexec}}{\left\langle a, \sigma_S \right\rangle \overset{sequence}{\rightarrow} \left\langle a, \sigma_S^{noexec} \right\rangle}$$

This rule applies when the next activity in the sequence to process is $\bot$. In such case the sequence state becomes $noexec$ and the sequence activity ends.

Axiom 1 (SA1)

$$\frac{}{\left\langle \bot, \sigma_S \right\rangle \overset{sequence}{\rightarrow} END}$$

this axiom states that if no more activities are in the list of activities to process, the sequence ends.

Axiom 2 (SA2):

$$\frac{\sigma_S = undefined}{\left\langle L_A, \sigma_S \right\rangle \overset{sequence}{\rightarrow} END_{undefined}}$$

This axiom states that the state of $sequence$, σ_S is $undefined$, because a fault has occurred.

The rules for a sequence construct are applied as follows:

```
S: Sequence
end=false;     fault=false;
while (!end && ! fault)
{
        if (first (S)==⊤ )
        {
                apply S1;
        }
        else if (first(S)==⊥)
        {
                apply S4;
                end=true;
                apply SA1;
         }
        else
        {
                apply S2;
                if (fault_occurred)
                {
                        apply S3.2;
                        fault=true;
                        apply SA2
                }
                else
                {
                        apply S3.1;
                }
        }

}
```

S1 is applied to start a sequence; no component activity of the sequence is yet started. **S2** is applied the first time to start the first activity in the sequence. **S3** is applied when the activity executed in the previous step is terminated. If this activity does not complete correctly the **S3.2** and **SA2** rules are applied and the sequence terminates in an *undefined* state, otherwise the other rules can be applied to the new list of activities to process that is the previous one pruned of the first activity. If the current list of activities to process contains only the $\bot$ activity the **S4** and **SA1** rules are applied and the process ends, else the **S2** rule at point is applied on the remaining activities.

D. Flow

Let be :

$$F = \top \| a_1 \| a_2 \| a_3 \cdots \| a_n \| \bot$$

Since it is no possible to start all activities simultaneously, a way to define an order of activation must be provided. In our definition the order is derived from the value of the index i. Thus the *first* transition can be applied also to flow rules. In addition it is necessary to remember all activities enacted and all activities that need to be started. The activities may terminate their execution only when all of them were started. Likewise for sequence, the state of the flow depends on the state of each component activity:

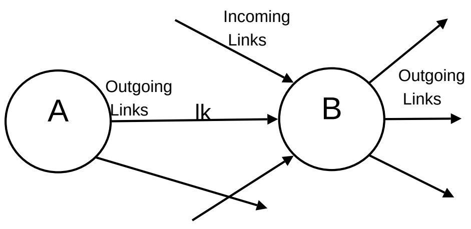

Figure 4. Links

$$\begin{aligned}\sigma_F &= exec \Leftrightarrow \exists i \in \{1,\cdots,n\} : \sigma_{a_i} = exec\\ \sigma_F &= noexec \Leftrightarrow \forall i \in \{1,\cdots,n\} : \sigma_{a_i} = noexec\end{aligned}$$

If the $flow$ is defined with the $links$, the following semantics rules apply:

Rule 1 (F1):

$$\frac{\sigma_F = ready, L_A \overset{first}{\rightarrow} \top || L_{A_T}, \sigma_F \overset{\mu}{\rightarrow} \sigma_F^{exec}}{\left\langle L_L, L_A, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_L, L_A, \sigma_F^{exec} \right\rangle}$$

This rule applies when the $flow$ is not yet started. In such case the first activity of the $flow$ is the $\top$ activity. The *first* transition put in the L_{A_T} list the remaining $flow$ activities and the state of the $flow$ through the application of the μ becomes $exec$. The new state of the $flow$ is then σ_F^{exec} and the remaining activities to process (L_{A_T}) were pruned of the $\top$ activity.

Rule 2 (F2):

$$\textbf{F2.1}\ \frac{\sigma_F = exec, \left\langle L_A, L_L, \sigma_F \right\rangle \overset{join}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = exec \right\rangle}{\left\langle L_L, L_A, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} \right\rangle}$$

$$\textbf{F2.2}\ \frac{\sigma_F = exec, \left\langle L_A, L_L, \sigma_F \right\rangle \overset{join}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = undefined \right\rangle}{\left\langle L_L, L_A, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{undefined} \right\rangle}$$

This rules applies when the $flow$ is already started. At this point, the $Join$ transition can be applied. It starts only the activities whose $TransitionCondition$ is evaluated true. The state of $flow$ changes to:

- σ_F^{exec}: this state is the same of the state σ_F except for the state of some activities that become $exec$;
- $\sigma_F^{undefined}$: if a fault has occurred and it not is handled.

Rule 3 (F3):

$$\textbf{F3.1}\ \frac{\sigma_F = exec, \left\langle L_A, L_L, \sigma_F \right\rangle \overset{TauCostruct}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = exec \right\rangle}{\left\langle L_L, L_A, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} \right\rangle}$$

$$\textbf{F3.2}\ \frac{\sigma_F = exec, \left\langle L_A, L_L, \sigma_F \right\rangle \overset{TauCostruct}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = undefined \right\rangle}{\left\langle L_L, L_A, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{undefined} \right\rangle}$$

This rules applies to terminate at least one of the activities in execution. The $TauCostruct$ must be applied.

Rule 4 (F4):

$$\frac{\sigma_F = exec, \left\langle L_L, L_A \right\rangle \overset{AnalisysStatusLink}{\rightarrow} true, \sigma_F \overset{tau}{\rightarrow} \sigma_F^{noexec}}{\left\langle L_L, L_A, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_L, L_A, \sigma_F^{noexec} \right\rangle}$$

Rule 5 (F5):

$$\frac{\sigma_F = exec, \left\langle L_L, L_A \right\rangle \overset{AnalisysStatusLink}{\rightarrow} false, \sigma_F \overset{tau}{\rightarrow} \sigma_F^{undefined}}{\left\langle L_L, L_A, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_L, L_A, \sigma_F^{undefined} \right\rangle}$$

The $AnalisysStatusLink$ checks if the flow can terminate correctly depending on link status and transition conditions. In case of $DeathPathElimination$ application, dead links conditions are propagated in the flow construct.

These rules respectively apply if: F4) all started activities inside the $flow$ are terminated and the $Join$ transition does not start any activity; F5) Errors occur in the flow definition. In this case the state of the flow activity becomes $undefined$.

Rule 6 (F6):

$$\frac{a = \bot, \sigma_F = exec, \sigma_F \overset{tau}{\rightarrow} \sigma_F^{noexec}}{\left\langle L_L, a, \sigma_F \right\rangle \overset{flow}{\rightarrow} \left\langle L_L, \bot, \sigma_F^{noexec} \right\rangle}$$

This rule applies if the only activity in the flow to start is the $\bot$ activity. In such case the $flow$ state becomes $noexec$ through the τ transition.

Axiom 1(FA1):

$$\frac{}{\left\langle L_L, \bot, \sigma_F = noexec \right\rangle \overset{flow}{\rightarrow} END}$$

This axiom states that if no more activities are in the list of activity to process, the $flow$ ends.

Axiom 2(FA2):

$$\frac{}{\left\langle L_L, L_A, \sigma_F = noexec \right\rangle \overset{flow}{\rightarrow} END}$$

This axiom states that $flow$ ends correctly.

Axiom 3(FA3):

$$\frac{\sigma_F = undefined}{\left\langle L_L, L_A \right\rangle \overset{flow}{\rightarrow} END_{undefined}}$$

This axiom states that the state of $flow$ is σ_F = undefined, because a fault has occurred.

The rules for a flow construct are applied as follows:

```
F:Flow
end=false; fault= false;

while (!end && !flow)
{
        if (first(F)==⊤
                apply F1;
        else
        {
                apply F2.1 or F2.2 to all activities ready in F;
                apply F3.1 or F3.2 for all activities that can terminate;
                if (fault_occurred)
                {
                        fault=true;
                        apply F5,FA2;
                }
                else if ( all activities were executed)
                {
                        end= true;
                        apply F6,FA3;
                }
                else if (F ended due to Links States Analysis)
                {
                        end=true;
                        apply F4;FA1;
                }
        }
}
```

F1 is applied to start a flow when no component activity of the Flow is yet started. **F2** (2.1 and 2.2) are then recursively applied when starting new activities in the flow depending on the execution states of the other activities and **F3** (3.1 and 3.2) are recursively applied when terminating the activities in the flow. **F4** is applied when terminating the flow activity in presence of general activities; **FA1** is applied too and the derivation process ends. **F5** is applied when terminating the flow activity incorrectly; **FA2** is applied too and the derivation process ends. **F6** is applied when all activities in the flow were terminated correctly and the $\perp$ activity is examined; the **FA3** is applied and the process ends;

1) Synthesis of BP: In order to allow for the BP synthesis, the patterns in the SW graph are analyzed and associated into proper Prolog Query.

Each pattern is defined in terms of states and activations sequence of its components activities. A Prolog query is composed depending on these informations. Inferences on *BPEL Semantics Rules* allows for determining if any composition of BPEL constructs can implement the given pattern. Tracing the execution of the inferential engine also produces informations that are used to build the BP.

2) BP Verification: The analysis of the BP by the means of derivations on the previously described *BPEL Semantics Rules*, establishes if a given BPEL process is correct and can help to discover faults before the process is executed. The BP is again described in terms of a prolog Query, where its correct termination is requested. The Prolog Inferential Engine checks if the given BP can terminate without faults.

Some fault can happen only in presence of some particular inputs or conditions: this makes difficult their detection.

This verification phase also allows for retrieving the cause of the fault (if any), making possible the definition of compensation actions if needed.

In particular, it is possible to detect:

- wrong usage of BPEL constructs;
- undesirable behaviors of BPEL processes due, for example, to bad **links** use or bad constructs combinations;
- faults due to undiscovered semantics errors;
- wrong usage of fault handlers.
- reachability of any state of the executable process.

E. Validation of the composite service

This phase is not yet implemented. We aim to formally define this step as future work.

IV. ARCHITECTURE

The life cycle described in the previous section is supported and automated by the architecture shown in Figure 5.

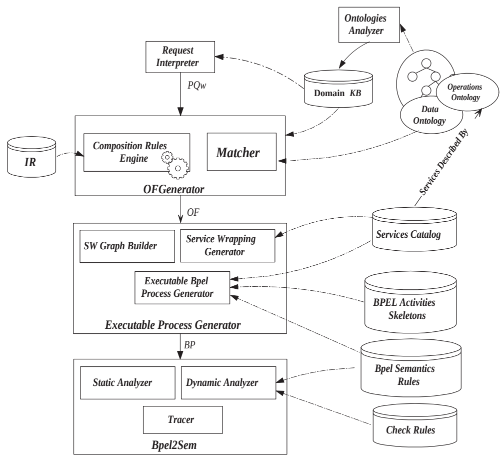

Figure 5. Architecture

First of all the Domain Ontologies have to be analyzed in order to build $\mathcal{KB}$. The *Ontologies Analyzer* is the component in charge of populating $\mathcal{KB}$ with Prolog axioms.

The *Request Interpreter* analyzes the user request and produces a description of the required service. The description is defined by using the IOPE and $\mathcal{KB}$. This module translates the required service description in the Prolog query $PQ_{\mathcal{W}}$.

The *OF Generator* performs the automatic generation of the Operational Flow Graph. It is based on a Prolog inference engine (*Composition Rules Engine*) and it uses the inference *Rules* ($\mathcal{IR}$) described in section III in order to build the OF graph while visiting an inference tree. For operation matching purposes, a proper component (the *Matcher*) implements the PE matching, needed to select the candidate operations as explained in Section III.

During the OF graph generation,more than one solution (OFs) can be selected to implement the required service. The OF generator will chose the first OF whose effects completely satisfy the request.

The *Service Wrapping Generator* modifies the OF graph if its services are not compatible in terms of I/O. It introduces in the OF the activation of proper wrapper operations (retrieved from the *Service Catalog* in order to perform I/O types translations for operations if needed). The new OF graph is analyzed by the *SW Graph Builder*. It implements the algorithm for the SW graph building described in section III. The output of this component is a SW graph containing the composition of operations in terms of workflow patterns. This graph is then analyzed in order to be translated (if possible) in a BPEL executable process by the *Executable BPEL Process Generator* as described in the previous section. This is achieved by substituting proper BPEL activities skeletons in place of patterns inside the SW. This last component also uses the methodology described in [24] in order to establish if a given pattern can be defined in the BPEL language.

Finally, *Bpel2sem* [10] is a tool for the automatic validation of BPEL executable processes, based on the Prolog language.

The *Static Analyzer* analyzes the BP produced in the previous steps and translates it into an internal representation. Furthermore, this module performs different kinds of analysis on the BPEL process like:

- Checks if the BPEL definition contains at least one activity able to start the process (the creation of a process instance in BPEL4WS is done by setting the ”createIstance” attribute of some receive or pick activity to ”yes”);
- Checks if the elements links are defined in the flow activity;
- Checks if every link declared within a flow activity has exactly one source activity and one target activity;
- Determines, if, for each target activity of a link,the source activity is declared, and vice versa;

so that, the *Dynamic Analyzer* can elaborate a correct process relatively the data necessary for the semantic analysis.

The *Dynamic Analyzer* aims to explore the full state space of a BPEL process. This analyzer uses the BPEL semantics rules translated into *prolog* rules and stored into a knowledge base which contains the *BPEL Semantic Rules* and rules used to detect errors and retrieve their causes, when the analyzed process is checked to end in an undefined state (*Check Rules*).

Derivations leads inevitably to an explosion of states to analyze. Proper pruning techniques are implemented into the Dynamic Analyzer in order to cope with the state space explosion problem.

Finally, the *Tracer* uses information generated during the analysis phases in order to produce information about process execution (traces) both when the semantics incorrectness are present in the process definition or not.

V. EXAMPLE

In this section an example is reported which shows how the lifecycle steps described in section II are exploited in order to produce a requested service by composition.

Let us suppose that a user requests for a service that returns the meaning and synonyms, both in the Italian language, of an Italian word. Furthermore let us assume that only the following services are available:

- an English Dictionary;
- an Italian Synonyms Dictionary;
- a multi-language translator.

The requested service can be obtained by composition of available services as we will explain in the following.

The descriptions of Ontology Domain and of available services operations are stored in the $\mathcal{KB}$. The Domain is described by Wordnet [43] ontologies while operations are described by proper OWL-S documents. The operations IOPE models are shown in Fig. 6, 7, 8

In the following some components of the wordnet schema used in this case study are descibed.
The WordNet schema has three main classes: Synset, WordSense and Word. In the Wordnet ontology, a *word form* represents a characters sequence such as ”man”, ”cat”; the same *word form* can have different meaning in several languages. A *Word*(element of Word class) is a word form in a language; in this Section we denote with $Word_E$, $Word_I$ a word in English and Italian language respectively. The property *LexicalForm* relates a word to its word form. A word can have different senses. A word sense(***WordSense*** class) represents a word used in particular sense; like the word class we denote with $WordSense_E$ the sense of a word in a English language etc. The property *sense* relates a word a their senses (the property *word* is its inverse). The synsets (***SynSet class***) are a synonyms set; a set of words that are interchangeable in some context. The property *containsWordSense* relates synset to their word sense (the property *hasSynset* is its inverse).

Let us describe the available services.

EnglishDictionary service is defined by

Input: **inputED** type **String**
Output: **outputED** type **String**
Precondition: **LexicalForm**($Word_E$, inputED)
Effect: **sense**(inputED, $WordSense_E$)

Figure 6. English Dictionary Service

The **EnglishDictionary** service, returns the meaning of the input word(**sense**(inputED, $WordSense_E$)) only if the input word form is a word belonging to the English dictionary (**LexicalForm**($Word_E$, inputED)).

SynonymsDictionary service is defined by

Input: **inputIS** type **String**
Output: **outputIS** type **String**
Precondition: **LexicalForm**($Word_I$, inputIS)
Effect: **sense**(inputIS,$SenseWord_I$)
$\wedge$ **hasSynset**($SenseWord_I$, Synset)

Figure 7. Italian Synonyms Dictionary Service

The Italian Synonyms Dictionary returns a set of Synonyms (Synset) of the inserted word only if the input word form is an Italian word (**LexicalForm**($Word_I$, inputIS)).

Translator service is defined by

Input: **inputT** type **String**
  inputLanguage type **String**
  outputLanguage type **String**
Output: **outputT** type **String**
Precondition: [**canBeTraslated**(inputLanuage,outputLanguage)]
$\wedge$ **LexicalForm**($Word_{IL}$, inputT)
Effect:[**LexicalForm**($Word_{OL}$, outputT)]$\wedge$
semanctics($Word_{IL}$,$Word_{OL}$)

Figure 8. Translator Operation

The translator is able to translate a word from a language to another. The translation is allowed only if the input-output languages are supported by the service (**canBeTranslated***(InputLanguage, OutputLanguage)*). Furthermore the input word has to be a valid word in the input language (**LexicalForm***($Word_{IL}$, inputT)*). The translator service must preserve the (semantics) meaning of the input word after the translation (**semantics***($Word_{IL}$, $Word_{OL}$)*).

The first step of the life cycle needs to translate the user request into the $PQ_{\mathcal{W}}$ query. In Fig. 9 the request for service in terms of IOPE model is reported.

ItalianDictionary service is defined by

Input: **InputWord** type **Word**
Output: **OutputWord1** type **SenceWord**
OutputWord2 type **Synset**
Precondition: LexicalForm($Word_I$, inputWord)
Effect:[**sense**(inputWord,SenseWordI)
$\wedge$**hasSynset**(SenseWordI, Synset)]
$\wedge$ [**sense**(inputWord,$SenseWord_I$]

Figure 9. $\mathcal{W}_{\mathcal{ID}}$ Service

The second step of the life cycle is the generation of the OF by exploiting $\mathcal{KB}$ and $\mathcal{IR}$. Composition Rules Engine and Matcher components act together to accomplish this task. The output of this phase is depicted in Fig. 10.

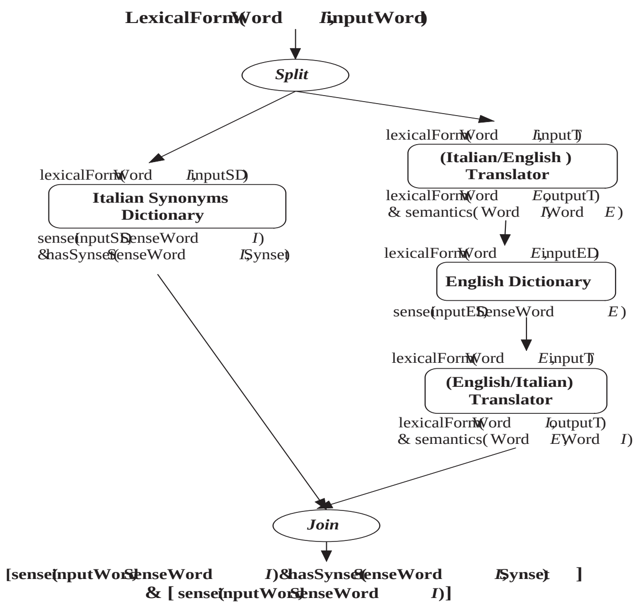
LexicalForm(Word I, inputWord)
Split
lexicalForm(Word I, inputSD)
Italian Synonyms Dictionary
sense(inputSD, SenseWord I)
&hasSynset(SenseWord I, Synset)
lexicalForm(Word I, inputT)
(Italian/English) Translator
lexicalForm(Word E, outputT)
& semantics(Word I, Word E)
lexicalForm(Word E, inputED)
English Dictionary
sense(inputED, SenseWord E)
lexicalForm(Word E, inputT)
(English/Italian) Translator
lexicalForm(Word I, outputT)
& semantics(Word E, Word I)
Join
[sense(inputWord, SenseWord I)&hasSynset(SenseWord I, Synset)]
& [sense(inputWord, SenseWord I)]

Figure 10. Operation Flow Model

In Figure 10,the squares represent the services, the expressions above the services represent the preconditions and the expression below the services represent the effects.

For example the matching between the (Italian/English) Translator and the English Dictionary is an exact match. It is possible to execute the services in a sequence. In fact, the P set of the second service (**lexicalForm**$Word_E$, *inputED*) is included in the set Eff of the translator service (**lexicalForm**$Word_E$, *outputT* $\wedge$ **semantics**$Word_I$,$Word_E$).

Furthermore the whole composed service has in the Eff set all predicates that appear below operations boxes and in the P set the request P predicates. Notice that $sense(inputED, SenseWord_{ED})$ and $sense(inputWord, SenseWord_{ID})$ are equivalent because the semantics of the translated words are the same after translations thanks to the $semantics(\cdots)$ properties. Hence the E set of the requested service is a subset of the Eff set of the composed service.

Then the OF model has to be translated into the SW model. Some wrapper services are needed in order to allow for I/O matching. A wrapper ($Wr1$ Service) is used since the Translator needs more parameters (the input language and the output language selections) than the ones available as Input. Hence $Wr1$ is a wrapper able to complete this list of the parameter required for the Translator service (it invokes the service with the constant languages: Italian and English).

The SW graph obtained after the application of the algorithm described in section III, is reported in Fig.11

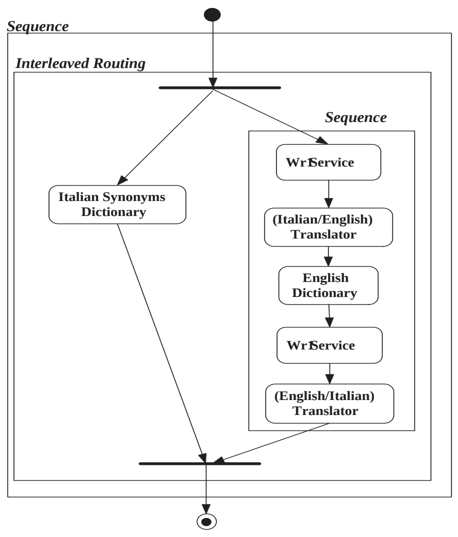
Sequence
Interleaved Routing
Sequence
Italian Synonyms Dictionary
Wr1Service
(Italian/English) Translator
English Dictionary
Wr1Service
(English/Italian) Translator

Figure 11. Services Workflow model

The SW graph is analyzed by the Executable Bpel Process Generator in order to establish if the SW process definition is implementable in the BPEL language. The BPEL process is the output of step. We do not show the whole BPEL process definition due to the lack of space.

A scratch of the BPEL process synthesized from the SW graph is reported in the following.

It is possible to identify in the process the SW pattern implementation. The Interleaved Routing pattern is implemented by using the *flow* BPEL construct with links, while sequences are trivially implemented by using the *sequence* construct. Links allow for definition of proper synchronization among activities present in the *flow* construct.

The process is also verified with success by the Bpel2Sem component. We do not report more informations about the enactment of derivation rules in verification due to the lack of space. More informations can be found in [10].

VI. Conclusion

In recent years many efforts have been made towards the automated composition of web services. Several re-

```
<?xml version="1.0" encoding="UTF-8"?>
<bpel:process ...>
   ...
  <bpel:variables>
     <bpel:variable messageType="ns1:inputIDMessage"
      name="inputIDMessage"/>
     <bpel:variable messageType="ns1:outputIDMessage"
      name="outputIDMessage"/>
      ...
  </bpel:variables>
  <bpel:sequence>
       <bpel:receive createInstance="yes" operation="request"
       partnerLink="dictionaryItalianLinkType"
       portType="ns1:dictionaryItalianPT"
       variable="inputIDMessage">
        </bpel:receive>
     <bpel:flow>
        <bpel:links>
           <bpel:link name="L3"/>
           <bpel:link name="L4"/>
        </bpel:links>
        <bpel:sequence name="Sequence1" >
           <bpel:invoke inputVariable="inputWR1Message" name="Wr1" .../>
           <bpel:assign>
              <bpel:copy>
                        ...<!-- Assign right Translation Languages-->
              </bpel:copy>
           </bpel:assign>
           <bpel:invoke inputVariable="inputTMessage" name="Translator"
                        operation="transalte"
                        outputVariable="outputTMessage"
                        partnerLink="translatorLinkType"
                        portType="ns3.translatorPT"/>
           <bpel:invoke inputVariable="inputEDMessage"
            name="English_Dictionary" operation="meaning" .../>
           <bpel:invoke inputVariable="inputWR1Message" name="Wr1" ... />
            <bpel:assign>
              <bpel:copy>
                        ...<!-- Assign right Translation Languages-->
              </bpel:copy>
           </bpel:assign>
          <bpel:invoke inputVariable="inputTMessage" name="Translator"
                operation="transalate" .../>
              <bpel:sources>
              <bpel:source linkName="L4"/>
           </bpel:sources>
        </bpel:invoke>
      </bpel:sequence>
        <bpel:sequence name="Sequence2" >
        <bpel:invoke inputVariable="inputIDMessage" name="Synonyms_ID"
                operation="SDictionary" ...>
           <bpel:sources>
              <bpel:source linkName="L3"/>
           </bpel:sources>
        </bpel:invoke>
       </bpel:sequence>
     <bpel:assign>
           <bpel:copy>
                        ...<!-- Outputs are copied into the reply message-->
              </bpel:copy>
        <bpel:targets>
              <bpel:target linkName="L3"/>
              <bpel:target linkName="L4"/>
           </bpel:targets>
           </bpel:assign>
           </bpel:flow>
     <bpel:reply operation="reply" .../>
  </bpel:sequence>
</bpel:process>
```

Figure 12. $\mathcal{W}_{\mathcal{ID}}$ Service BPEL Process

sults are described in literature, addressing different aspects of this problem. In this paper a life cycle approach to the automatic generation of executable web services compositions is presented, in which many of the available results may be integrated. The core of the process consists of the automated synthesis of two graph models of the composite process: the operations flow model (OF) and the services workflow model (SW). The construction of OF requires: a) to select a set of (available) service operations that are semantically compatible (matching of Pre-conditions and Effects); b) to determine a composition of such operations which semantically matches Pre-conditions and Effects of the requested composite service. The OF model is then translated into the SW model. The construction of SW requires: a) to check and provide the Input/Output matching of the selected operations; b) the application of graph transformation techniques in order to express the composition by means of workflow patterns. SW is then used to generate an executable BPEL process. An architecture to implement the proposed approach is also described. Future work will include: investigating the integration in this reference architecture of more matching and graph transformation algorithms and tools, supporting further workflow patterns and loops, addressing quality of services and security requirements.

REFERENCES

[1] G. Agarwal, K. Chafle, N. Dasgupta, A. Karnik, S. Kumar, V. Mittal, B. Srivastava, Synthy: A System for End to End Composition of Web Services, Journal of Web Semantics, 3(4):311-339, 2005.

[2] R. Agarwal, K. Verma, J. Miller, W. Milnor, Constraint Driven Web Service Composition in METEOR-S, IEEE Intl. Conf. on Services Computing (SCC 2004):23–30, 2004.

[3] Alfred V. Aho , Ravi Sethi , Jeffrey D. Ullman, Compilers: principles, techniques, and tools Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1986

[4] T. Andrews, F. Curbera, H. Dholakia, et al., Business Process Execution Language for Web Services, Version 1.1, 2003. http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

[5] B. Benatallah, Q.Z. Sheng, M. Dumas, The SELF-SERV Environment for Web Services Composition, IEEE Internet Computing:40–48, 2003.

[6] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of e-Services that Export their Behavior, in 1st Intl. Conference on Service Oriented Computing (ICSOC 2003), LNCS 2910:4-358, 2003.

[7] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, M. Mecella, Towards Automatic Web Service Discovery and Composition in a Context with Semantics, Messages, and Internal Process Flow, Position paper at W3C Workshop on Frameworks for semantic web services, 2005.

[8] G. Chafle, G. Das, K. Dasgupta, A. Kumar, S. Mittal, S. Mukherjea, B. Srivastava, An Integrated Development Environment for Web Service Composition, In Proc. of the IEEE Int. Conf. on Web Services (ICWS 2007). Also as IBM Research Report RI 06009, November 2006.

[9] Jen-Yao Chung, Kwei-Jay Lin, Irvine Richard G. Mathieu, *IEEE Computer, Special Issue on Web Services Computing*, 36(10), 2003.

[10] G. Di Lorenzo, F. Moscato, N. Mazzocca, V. Vittorini, Automatic Analysis of Control Flow in Web Services Composition Processes, in PDP 2007:299-306, 2007.

[11] R. Eshuis , R. Wieringa, Verification support for workflow design with UML activity graphs, Proc. of the 24th Int. Conf. on Software Engineering, May 19-25, 2002, Orlando, Florida

[12] D. Fahland and W. Reisig, ASM-based semantics for BPEL: The negative Control Flow, in 12th Intl. Workshop on Abstract State Machines (ASM 2005), 131–151, 2005.

[13] R. Farahbod, U. Glässer, and M. Vajihollahi, A formal semantics for the business process execution language for Web Services, in third Workshop on Web Services: Modeling, Architecture and Infrastructure (WSMDEIS 2005):122–133, 2005.

[14] A. Ferrara, Web services: a process algebra approach, in 2nd Intl. Conf. on Service Oriented Computing, 242–251, 2004.

[15] M. Grüninger, R. Hull, S. McIlraith, A First-Order Ontology for Semantic Web Services, Position paper at W3C Workshop on Frameworks for semantic web services, 2005.

[16] M.C. Jaeger, L. Engel and K. Geihs, A Methodology for Developing OWL-S Descriptions, in First Intl. Conf. on Interoperability of Enterprise Software and Applications Workshop on Web Services and Interoperability, 2005.
[17] M. Klein, B. König-Ries, M. Müssig, What is Needed for Semantic Service Descriptions? A Proposal for Suitable Language Constructs, *Int. Journal of Web and Grid Services*, 1(3):328–364, 2005.
[18] M. Klusch, B. Fries, M. Khalid, K. Sycara, OWLS-MX: Hybrid OWL-S Service Matchmaking, in 1st Intl. AAAI Fall Symposium on Agents and the Semantic Web, 2005.
[19] F. Lécué, A. Léger, Semantic Web Service Composition through a Matchmaking of Domain, in ECOWS'06 2006:171–180, 2006.
[20] D. Martin, M. Paolucci, S. McIlraith, et al., Bringing Semantics to Web Services: The OWL-S Approach, in SWSWPC 2004, 26–42, 2004.
[21] D.L. McGuinness and F. van Harmelen, OWL Web Ontology Language Over-view, World Wide Web Consortium (W3C) Candidate Recommendation. August 18, 2003. http://www.w3.org/TR/owl-features/
[22] S. McIlraith, T. Son, and H. Zeng, Semantic Web Services, *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, 16(2):46–53, 2001.
[23] Mindswap, Maryland Information and Network Dynamics Lab Semantic Web Agents Project. http://www.mindswap.org
[24] F. Moscato, N. Mazzocca, V. Vittorini, G. Di Lorenzo, P. Mosca, M. Magaldi, Workflow Pattern Analysis in Web Services Orchestration: The BPEL4WS Example, in HPCC 2005:395-400, 2005.
[25] S. Narayanan and S. McIlraith, Simulation, Verification and Automated Composition of Web Services, in 11th Intl. World Wide Web Conference (WWW 2002):77–88, 2002.
[26] C. Ouyang, E. Verbeek, W.M. van der Aalst, S. Breutel, M. Dumas, and A.H. ter Hofstede, Formal Semantics and Analysis of Control Flow in WS-BPEL, Technical Report 2174, Queensland University of Technology, 2006. http://eprints.qut.edu.au/archive/00002174/01/BPM-05-15.pdf
[27] OWL-S Coalition. OWL-S 1.0 Release. http://www.daml.org/services/owl-s/1.0/
[28] M. Paolucci, T. Kawamura, T.R. Payne, K. Sycara, Semantic matching of web services capabilities, in First Intl. Semantic Web Conference, LNCS 2342:333-347, 2002.
[29] M.P. Papazoglou and D. Georgakopoulos, Service-Oriented Computing, *Communication of the ACM (Special Issue on Service-Oriented Computing)*, 46(10), 2003.
[30] J. Pathak, S. Basu, V. Honavar, Modeling Web Services by Iterative Reformulation of Functional and Non-Functional Requirements, in 4th Intl. Conf. on Service Oriented Computing (ICSOC-2006), LNCS 4294:314-326, 2006.
[31] M. Pistore, P. Roberti, P. Traverso, Process-Level Composition of Executable Web Services: "On-the-fly" Versus "Once-for-all" Composition, in ESWC 2005: 62–77, 2005.
[32] D Roman, U. Keller, H. Lausen, et al., Web Service Modeling Ontology, *Applied Ontology*, 1(1):77–106, 2005.
[33] N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar, Workflow Control-Flow Patterns: A Revised View. BPM Center Report BPM-06-22 , BPMcenter.org, 2006.
[34] H. Skogsrud, B. Benatallah, F. Casati, F. Toumani, Managing Impacts of Security Protocol Changes in Service-Oriented Applications, IEEE Proc. of the 29th Int. Conf. on Sof. Eng. (ICSE 2007):468–477, 2007.
[35] E. Sirin, B. Parsia, J. Hendler, Filtering and Selecting Semantic Web Services with Interactive Composition Techniques, *IEEE Intelligent Systems*, 19(4):42–49, 2004.
[36] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, Y. Katz, Pellet: a practical owl-dl reasoner, Submitted for publication to *Journal of Web Semantics*. http://www.mindswap.org/papers/PelletJWS.pdf
[37] SWSL Committee, Semantic web service ontology (swso). http://www.daml.org/services/swsf/1.0/swso/
[38] P. Traverso, M. Pistore, Automated Composition of Semantic Web Services into Executable Processes, in Intl. Semantic Web Conference 2004: 380–394.
[39] Workflow Management Coalition, XPDL definition, http://www.wfmc.org/standards/xpdl.htm
[40] WSMO working group, Web Service Modeling Ontology (WSMO). http://www.wsmo.org/
[41] NW.M.P van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, Workflow Patterns. Distributed and Parallel Databases, 14(3), pages 5-51, July 2003.
[42] L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, H. Chang, QoS-aware middleware for Web services composition, IEEE Trans. on Soft. Eng, 30(5), pp. 311–327, 2004.
[43] Wordnet http://www.w3.org/TR/wordnet-rdf/

AUTHOR BIOGRAPHIES

Giusy di Lorenzo is a Ph.D. student in Computer Engineering at the Department of Computer Science and Systems of the University of Naples Federico II. She obtained her Master degree in Computer Engineering in 2005 from the University of Naples Federico II. Her current research activities include composite web-services analysis and verification.

Nicola Mazzocca is full professor of Calcolatori Elettronici at the University of Naples Federico II. He graduated in electronic engineering from the University of Naples, Italy, in 1987, and received his Ph.D. from the same university. His scientific activity involves methodologies and tools for performance evaluation of computing systems, computer networks, communication protocols, general and special purpose parallel architectures and applications. Since 1998 he partecipated in various research projects as coordinator.

Francesco Moscato is reseach assistant at the Department of Computer Science and Systems of the University of Naples Federico II. He obtained his Master degree in Computer Engineering in 2002 from the University of Naples Federico II and his Ph.D. in Electronic Engineering in 2005 at the Second University of Naples (SUN). His research interests include: complex system modeling by multi-formalism techniques, formal verification of reactive systems, composite web-services composition and verification.

Valeria Vittorini is associate professor at the Department of Computer Science and Systems of the University of Naples Federico II, Italy. She graduated in Mathematics at the University of Naples in 1990 where she received her Ph.D. degree in Computer Science in 1996. Her research interests include distributed system, systems modelling and formal methods in system specification and design.