

## D2.2 Semantic Querying, Discovery, and Composition Framework

Schahram Dustdar<sup>1</sup>, Jörg Hoffmann<sup>2</sup>, Ta'id Holmes<sup>1</sup>, Adina Sirbu<sup>2</sup>, Huy Tran<sup>1</sup>, and Uwe Zdun<sup>1</sup>

*<sup>1</sup>Distributed Systems Group  
Information Systems Institute  
Vienna University of Technology  
Austria  
{lastname}@infosys.tuwien.ac.at*

*<sup>2</sup>Digital Enterprise Research Institute  
Leopold – Franzens Universität Innsbruck  
Austria  
{firstname.lastname}@deri.at*

20. August 2007



## **Abstract**

The business process modeling ontology (BPMO) contributes to bridging the gap between the business level perspective and the technical implementation level in business process management (BPM) by semantic descriptions of business processes. In order to benefit from the semantical description of business processes, semantically enabled techniques have to be developed for querying, discovering and composing business processes accordingly. This deliverable introduces a comprehensive semantic business process framework with the basic components of semantic query, discovery and composition.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Semantic Query &amp; Discovery</b>	<b>3</b>
2.1	Process Reasoner . . . . .	4
2.2	Process Discovery . . . . .	4
<b>3</b>	<b>Functional Composition</b>	<b>5</b>
3.1	Introduction . . . . .	5
3.2	Functional Composition with Partial Matches . . . . .	6
3.3	Forward Effects . . . . .	8
3.4	Conformant-FF . . . . .	9
3.5	From Conformant-FF to Forward Effects . . . . .	10
3.5.1	Adapting the generation of formulas . . . . .	10
3.5.2	Reusing knowledge . . . . .	12
<b>4</b>	<b>Syntactic Composition</b>	<b>14</b>
4.1	Workflow Languages . . . . .	14
4.2	Process Engine . . . . .	15
4.3	VDE Provider . . . . .	17
4.3.1	Validation . . . . .	17
4.3.2	Deployment . . . . .	17
4.3.3	Execution . . . . .	18
4.4	Modeling Framework . . . . .	18
4.4.1	Overview of the modeling framework . . . . .	19
4.4.2	View-based modeling framework . . . . .	20
<b>5</b>	<b>Conclusion</b>	<b>27</b>
5.1	Further Work . . . . .	27

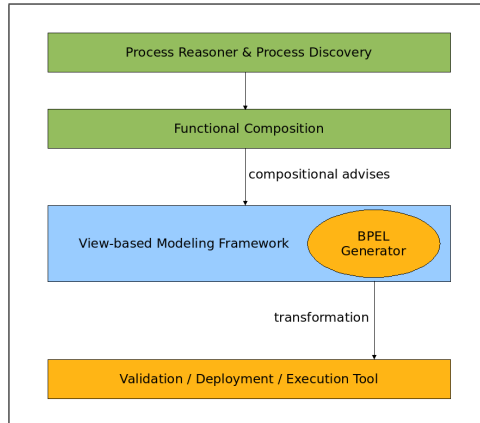


Figure 1: Overview of the overall framework

## 1 Introduction

The aim of the business process modeling ontology (BPMO) is to define a precise and sufficient semantic description model for business processes that encompasses business process description on a higher level of abstraction in order to enable business level BPM<sup>1</sup> facilities as well as connection to the technical implementation level for supporting automated execution of business processes.

In order to benefit from the semantical description of business processes, semantically enabled techniques have to be developed for querying, discovering and composing business processes accordingly.

This paper is structured as follows: in this section we have explicated the need for a comprehensive semantic business process framework. We will then present the components for semantic query and discovery in Section 2. Functional composition as independent from a specific workflow language will be covered in Section 3. The modelling framework that can map the results to a certain workflow language will be presented in Section 4. Together these components constitute the overall composition framework. Section 5 will summarize this work by giving a conclusion and referring to further work.

Figure 1 gives an overview of the overall framework.

## 2 Semantic Query & Discovery

Semantic query and discovery are two approaches for data retrieval. Semantic query refers to query answering with respect to a BPMO repository, using not only the data stored explicitly, but as well the ontological aspect of the data. With semantic discovery of BPMO business processes, we automatically locate the semantic descriptions of business processes that meet certain functional criteria. This section treats them together,

---

<sup>1</sup>business process management

because in both cases the main effort consists in adapting existing WSMO solutions to BPMO.

## 2.1 Process Reasoner

The semantic query functionality necessary within the SemBiz BPMSuite [39] is achieved by means of ontology reasoning. The requirements come from the BPMO Process Editor, the BPMO Process Validator and Process Discovery.

For this purpose, we adapt to BPMO the WSML2Reasoner <sup>2</sup> framework, using the Integrated Rule Inference System (IRIS) <sup>3</sup> as underlying datalog engine. The WSML2Reasoner framework is a highly modular architecture which combines various functionalities (validation, normalization and transformation), essential to the translation of ontologies described using WSML [23] to the appropriate syntax of several underlying reasoning engines. Additionally, it provides a Facade for easy integration for different reasoners (for example MINS, IRIS, Pellet and KAON2). IRIS uses a reasoning mechanism based on deductive database algorithms like semi-naive algorithms, dynamic filtering, and well-founded evaluation with alternating fixed point computation. Together, WSML2Reasoner and IRIS support query answering for WSML-Core and WSML-Flight. Reasoning support for WSML-Flight has been one of the main requirements, since BPMO is defined as a WSML-Flight ontology.

## 2.2 Process Discovery

As already mentioned in [26], we have taken as starting point the existing work in the area of Semantic Web service discovery, in particular the WSMO discovery [22]. Several approaches to Web service discovery have been identified within the WSMO framework, delivering results of different accuracy. The syntactic approach, also known as keyword-based discovery, has the advantage that it can quickly filter a possibly huge set of available Web services. However, this approach has a low precision, as it returns also a large number Web services that are irrelevant to the request.

The semantic-based approaches have a higher precision, but require certain expressivity in the semantic description of the Web services. These approaches reduce matching of goals and Web services to standard reasoning tasks in the language used for semantic representation.

Discovery based on simple descriptions of services considers Web services and goals in terms of the offered, respectively requested, sets of objects. More precisely, at this level, a Web service is represented as a set of objects that it delivers to its domain of value, while a goal is represented as a set of elements which are relevant to the client as the outcome of a service execution. The descriptions of these sets refer to ontologies that capture general knowledge about the domain under consideration.

At the highest level of abstraction is the discovery based on rich semantic descriptions of services. The Web services considered by this approach must be expressed in terms of pre-state and post-state constraints that describe the intended execution of the service.

---

<sup>2</sup><http://tools.deri.org/wsml2reasoner/>

<sup>3</sup><http://iris-reasoner.org/>

One of the initial tasks is therefore adapting the WSMO discovery to BPMO. We chose to adapt the WSMX implementation of the WSMO discovery, because it allows for Web service discovery to be performed using any of the presented approaches, or an appropriate combination. The idea is to take advantage of the various tradeoffs offered by each approach, in a way that it is suitable for the considered search space.

Adapting WSMO discovery to BPMO is straightforward, since BPMO business processes and business goals have their capabilities described semantically, in a very similar way to WSMO Web services and goals.

A future task that will involve considerable effort is adapting the type of discovery to also suit the needs of functional composition (see Section 3). First of all, the composition we are designing can handle *partial matches*, where capabilities of BPMO business processes may deal with only a part of the requirement. This type of matching capabilities must of course be retrieved by semantic discovery. Also, in order to support composition, the discovery should be able to determine, in a second phase, not only the capabilities of BPMO business processes that match the user goal, but also those that match the preconditions required by the capabilities that are already in the discovery result set. This operation must be performed until a fixed point is reached.

### 3 Functional Composition

This section describes the SemBiz approach to semantic composition of BPMO business processes. Because for the automatic processing of semantic descriptions this is the main contribution of this deliverable, and also because it represents an innovative approach, we will present in detail the formalism we consider, followed by a special case that we identified, and also, the advances made in designing a tool for the special case.

#### 3.1 Introduction

The functional composition of BPMO business processes addresses composition at the *capability* level, where only their overall behavior is taken into account: the inputs and outputs, as well as the preconditions and postconditions, all formulated with respect to a domain ontology (also referred to as background ontology).

When designing the functional composition of BPMO business processes, we have taken as reference point the existing work in the area of Semantic Web service composition (WSC). Like any task that involves the automatic processing of semantic descriptions of service-related resources, the composition of BPMO business processes needs to include support for background ontologies, which specify the behavior of the underlying domain.

In the case of WSC, almost all existing solutions compile the problem into AI Planning formalisms. The motivation is that planning tools have become many times more scalable in recent years, through the use of heuristic functions and other search techniques, e.g., [27, 18, 15]. The problem here is that those tools cannot handle background ontologies. The following example illustrates the importance of those:

**Example 3.1** A service shall be composed that inputs a constant of concept  $A$ , and outputs a constant of concept  $C$ . (E.g.,  $A$  may be a trip request and  $C$  a ticket.) The ontology defines the concepts  $A, B, B_1, \dots, B_n, C$ , and states that each  $B_i \subseteq B$ , and  $\bigcup_1^n B_i \supseteq B$ . (E.g.,  $B$  may be a geographical region and the  $B_i$  its parts.) An available service  $ws_{AB}$  transforms  $A$  into  $B$ , and for each  $i$  an available service  $ws_{B_iC}$  transforms  $B_i$  into  $C$ . A solution first applies  $ws_{AB}$ , and then applies the  $ws_{B_iC}$  in conjunction.

In Example 3.1, reasoning over the background ontology is necessary to (1) understand which services can be used, and to (2) test whether a given composition is actually a solution. Such reasoning can be modeled through the *background theories* explored in some planning works, e.g., [13, 16]. However, incorporating this notion into the modern scalable planning tools poses serious challenges, and has not yet even been tried. Due to the background theory, even computing a state transition – which is now a form of belief revision – is a computationally very hard task. The existing planning tools dealing with background theories, e.g., [13, 16], map the problem into generic deduction, which is well known for its lack of scalability. The existing planning tools dealing with WSC, e.g., [30, 6], ignore the ontology and assume *exact matches*. In Example 3.1, this would require  $B = B_i$  instead of  $B \cap B_i \neq \emptyset$ . Obviously, this renders the example unsolvable.

We identify an interesting special case of functional composition. We exploit the fact that capabilities (of Web services, as well as BPMO business processes) may output new constants.<sup>4</sup> Now, if all ramifications concern only propositions involving at least one new constant, then a belief revision is not necessary. We term this special case functional composition with *forward effects*: the effects are “forward” in the sense that no backwards-directed belief revision is necessary. E.g., the services in Example 3.1 have forward effects. The same holds for many scenarios from the literature, and from real case studies. A simple example is the wide-spread “virtual travel agency”, where Web services must be linked that book travel and accommodation, generating new constants corresponding to tickets and reservations.

We introduce a framework for planning with background theories. We show that testing whether an action sequence is a solution – *solution testing* – is  $\Pi_2^P$ -complete in general, but only **coNP**-complete with forward effects. Of course, **coNP** is still hard. However, *planning under uncertainty* has the same complexity of solution testing, and scalable tools for this case have already been developed. Adapting them for functional composition with forward effects is our current step. In particular, the Conformant-FF tool [17] is based on CNF reasoning, which can be naturally extended to our setting.

We will first introduce our formalism, in Section 3.2. Section 3.3 discusses forward effects. Section 3.4 presents the main features of Conformant-FF, and Section 3.5 provides some details regarding our developments.

## 3.2 Functional Composition with Partial Matches

We introduce a framework, called *WSC*, that borrows notation from AI Planning, and corresponds to functional composition (of either BPMO business processes, or Web services). The (planning) terminology of our formalism corresponds to functional composition as follows. BPMO business processes (respectively, Web services) are planning

---

<sup>4</sup>Namely, the outputs model the generated data.

”operators”; their input/output behavior maps to input/output parameters on which preconditions and effects are specified [3, 11, 14]. The background ontology is the background ”theory”. The precondition in the goal is equivalent to sets of ”initial literals” and ”initial constants”. The effect in the goal is the ”goal condition”.

We assume supplies of logical predicates  $p, q$ , variable names  $x, y$  and constant names  $a, b, c, d, e$ ; (*ground*) *literals* are defined as usual. For variables  $X$ ,  $\mathcal{L}^X$  is the set of literals using only variables from  $X$ . We write  $l[X]$  for a literal  $l$  with variable arguments  $X$ . For a tuple  $C$  of constants substituting  $X$ , we write  $l[C/X]$ . In the same way, we use the substitution notation for any construct involving variables. Positive ground literals are *propositions*. A *clause* is a disjunction of literals with universal quantification on the outside, e.g.  $\forall x.(\neg p(x) \vee q(x))$ . A *theory* is a conjunction of clauses. An *operator*  $o$  is a tuple  $(X_o, \text{pre}_o, Y_o, \text{eff}_o)$ , where  $X_o, Y_o$  are sets of variables,  $\text{pre}_o$  is a conjunction of literals from  $\mathcal{L}^{X_o}$ , and  $\text{eff}_o$  is a conjunction of literals from  $\mathcal{L}^{X_o \cup Y_o}$ . The intended meaning is that  $X_o$  are the inputs and  $Y_o$  the outputs, i.e., the new constants created by the operator. For an operator  $o$ , an *action*  $a$  is given by  $(\text{pre}_a, \text{eff}_a) \equiv (\text{pre}_o, \text{eff}_o)[C_a/X_o, E_a/Y_o]$  where  $C_a$  and  $E_a$  are vectors of constants; for  $E_a$  we require that the constants are pairwise different. *WSC tasks* are tuples  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$ . Here,  $\mathcal{P}$  are predicates;  $\mathcal{T}$  is the theory;  $\mathcal{O}$  is a set of operators;  $C_0$  is a set of constants, the initial constants supply;  $\phi_0$  is a conjunction of ground literals, describing the possible initial states;  $\phi_G$  is a conjunction of literals with existential quantification on the outside, describing the goal states, e.g.,  $\exists x, y.(p(x) \wedge q(y))$ .<sup>5</sup> All predicates are from  $\mathcal{P}$ , all constants are from  $C_0$ , all constructs are finite.

Assume we are given a task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$ . *States* in our formalism are pairs  $(C_s, I_s)$  where  $C_s$  is a set of constants, and  $I_s$  is a  $C_s$ -*interpretation*, i.e., an interpretation of all propositions formed from the predicates  $\mathcal{P}$  and the constants  $C_s$ . We need to define the outcome of applying actions in states. Given a state  $s$  and an action  $a$ ,  $a$  is *applicable* in  $s$  if  $I_s \models \text{pre}_a$ ,  $C_a \subseteq C_s$ , and  $E_a \cap C_s = \emptyset$ . We allow *parallel actions*. These are sets of actions which are applied at the same point in time. The result of applying a parallel action  $A$  in a state  $s$  is  $\text{res}(s, A) :=$

$$\{(C', I') \mid C' = C_s \cup \bigcup_{a \in A, \text{appl}(s, a)} E_a, I' \in \min(s, C', \mathcal{T} \wedge \bigwedge_{a \in A, \text{appl}(s, a)} \text{eff}_a)\}$$

Here,  $\min(s, C', \phi)$  is the set of all  $C'$ -interpretations that satisfy  $\phi$  and that are minimal with respect to the partial order defined by  $I_1 \leq I_2$  :iff for all propositions  $p$  over  $C_s$ , if  $I_2(p) = I_s(p)$  then  $I_1(p) = I_s(p)$ . This is a standard semantics where the ramification problem is addressed by requiring minimal changes to the predecessor state  $s$  [37]. Note that  $\text{res}(s, A)$  allows non-applicable actions. This realizes partial matches: a BPMO business process (respectively, a Web service) can be applied as soon as it matches at least one possible situation.

We also allow the effects of the actions in  $A$  to be contradictory; this is safe if the contradictory effects never occur together. We say that  $A$  is *inconsistent* with a state  $s$  if  $\text{res}(s, A) = \emptyset$ ; this can happen in case of conflicts between the effects of the actions that are applicable in  $s$ .

We refer to the set of states possible at a given time as a *belief*. The *initial belief* is  $b_0 := \{s \mid C_s = C_0, s \models \mathcal{T} \wedge \phi_0\}$ . A parallel action  $A$  is inconsistent with a belief  $b$  if it

<sup>5</sup>The existential quantification is needed to give meaning to the creation of new constants.

is inconsistent with at least one  $s \in b$ . In the latter case,  $res(b, A)$  is undefined; else, it is  $\bigcup_{s \in b} res(s, A)$ . This is extended to action sequences in the obvious way. A *solution* is a sequence  $\langle A_1, \dots, A_n \rangle$  s.t. for all  $s \in res(b_0, \langle A_1, \dots, A_n \rangle) : s \models \phi_G$ .

When assuming *fixed arity* – a constant upper bound on the arity of all variable vectors (e.g., used in predicates) – transformation to a propositional representation is polynomial. Even in this case, solution testing is  $\Pi_2^p$ -complete in  $\mathcal{WSC}$ . Further, we have proven that polynomially bounded solution existence is  $\Sigma_3^p$ -complete.

**Theorem 3.2 (Solution testing in  $\mathcal{WSC}$ )** *Assume a  $\mathcal{WSC}$  task with fixed arity, and a sequence  $\langle A_1, \dots, A_n \rangle$  of parallel actions. It is  $\Pi_2^p$ -complete to decide whether  $\langle A_1, \dots, A_n \rangle$  is a solution.*

### 3.3 Forward Effects

The high complexity of  $\mathcal{WSC}$  motivates the search for interesting special cases. As stated, here we define a special case where every change an action makes to the state involves a new constant. A  $\mathcal{WSC}$  task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$  has *forward effects* iff:

- For all  $o \in \mathcal{O}$ , and for all  $l[X] \in \text{eff}_o$ , we have  $X \cap Y_o \neq \emptyset$ . In words, the variables of every effect literal contain at least one output variable.
- For all clauses  $cl[X] \in \mathcal{T}$ , where  $cl[X] = \forall X.(l_1[X_1] \vee \dots \vee l_n[X_n])$ , we have  $X = X_1 = \dots = X_n$ . In words, in every clause all literals share the same arguments.

The set of all such tasks is denoted with  $\mathcal{WSC}|_{fwd}$ . The second condition implies that effects involving new constants can only affect literals involving new constants. Given a state  $s$  and a parallel action  $A$ , define  $res|_{fwd}(s, A) :=$

$$\{(C', I') \mid C' = C_s \cup \bigcup_{a \in A, \text{exec}(s, a)} E_a, I'|_{C_s} = I_s, I' \models \mathcal{T} \wedge \bigwedge_{a \in A, \text{exec}(s, a)} \text{eff}_a\}$$

Here,  $I'|_{C_s}$  denotes the restriction of  $I'$  to the propositions over  $C_s$ .

**Proposition 3.3 (Semantics of  $\mathcal{WSC}|_{fwd}$ )** *Assume a  $\mathcal{WSC}|_{fwd}$  task, a state  $s$ , and a parallel action  $A$ . Then  $res(s, A) = res|_{fwd}(s, A)$ .*

Hence, the action semantics becomes a lot simpler with forward effects, no longer needing the notion of minimal changes with respect to the previous state.

In difference to  $\mathcal{WSC}$ , we can make sure that no inconsistent actions will occur by filtering out certain actions in a preprocessing step. An action  $a$  is called *contradictory* if  $\mathcal{T} \wedge \text{eff}_a$  is unsatisfiable.

**Proposition 3.4 (Inconsistency in  $\mathcal{WSC}|_{fwd}$ )** *Assume a  $\mathcal{WSC}|_{fwd}$  task, a belief  $b$  reachable in the task, and a parallel action  $A$ . If  $A$  is inconsistent with  $b$ , then there exists  $a \in A$  so that  $a$  is contradictory.*

Obviously, a contradictory action will never yield a successor state; it can be filtered out prior to planning, without affecting solution existence. Thanks to this, and thanks to the simpler semantics as per Proposition 3.3, solution testing is much easier in  $\mathcal{WSC}|_{fwd}$  than in  $\mathcal{WSC}$ .

**Theorem 3.5 (Solution testing in  $\mathcal{WSC}|_{fwd}$ )** *Assume a  $\mathcal{WSC}|_{fwd}$  task with fixed arity, and without contradictory actions. Assume a sequence  $\langle A_1, \dots, A_n \rangle$  of parallel actions. It is **coNP**-complete to decide whether  $\langle A_1, \dots, A_n \rangle$  is a solution.*

**Theorem 3.6 (Polynomially bounded solution existence in  $\mathcal{WSC}|_{fwd}$ )** *Assume a  $\mathcal{WSC}|_{fwd}$  task with fixed arity, and a natural number  $b$  in unary representation. It is  $\Sigma_2^P$ -complete to decide whether there exists a solution using at most  $b$  actions.*

### 3.4 Conformant-FF

Planning under uncertainty, or conformant planning, is the task of generating plans given uncertainty about the initial state, and without any sensing capabilities during plan execution. There exists a correspondence between functional composition with partial matches and planning under uncertainty: in both cases the world is described in terms of a formula and the composed solution should work for every situation satisfying that formula. Very encouraging for our work is that solution testing for planning under uncertainty is also **coNP**-complete, and scalable tools have already been developed.

An important contribution of our work is the adaptation of one of these tools, namely Conformant-FF [17]. It is known that conformant planning can be transformed into a search problem in belief space, the space whose elements are sets of possible worlds. The key observation introduced with Conformant-FF is that, for a setting where the goal and action preconditions are reduced to simple conjunctions of propositions, it is sufficient to know the propositions that are true in the intersection of worlds contained in a belief state. Based on this observation, Conformant-FF trades space for time. Instead of maintaining complete knowledge of a belief state  $s$  in memory, the planner maintains for  $s$  only the known propositions and the path leading to it (the initial belief state and the action sequence that leads to  $s$ ). To check for fulfillment of the goal and action preconditions, the planner checks that every proposition is implied by a CNF formula that captures the semantics of the action sequence. This approach can be naturally extended to functional composition with forward effects, since forward effects beliefs can be represented as propositional CNFs. This is not possible in the general case, for complexity reasons.

Making this approach efficient is a challenging task. Preconditions and effects are more complex in our case, making it more difficult to apply the optimizations used in Conformant-FF, for example caching and re-using of results, or identifying sufficient conditions for cases that do not require reasoning. Moreover, Conformant-FF (and every existing planning tool) does not allow the generation of new constants. It is critical to devise heuristics for identifying which new constants are important, since exponentially many constants may be generated in general.

Practical scenarios are expected to involve large sets of business processes (respectively, Web services), and huge search spaces (of partial compositions). To overcome large search spaces, we need to design heuristic functions and filtering techniques. Here again, we start from ideas underlying Conformant-FF: for each belief, Conformant-FF computes a relaxed solution using approximate SAT reasoning, and uses it to guide the search. The length of this solution is the heuristic function: an estimate of how much more effort is required to complete the search state. The actions contained in the relaxed solution provide the filtering technique: an estimate of which actions are most relevant to complete the search state. We differ from Conformant-FF in the different structure of CNFs, necessitating different approximate reasoning, and in that we will explore typical forms of ontologies (e.g., subsumption hierarchies) to obtain better distance estimates.

### 3.5 From Conformant-FF to Forward Effects

This section introduces the procedure for generating a CNF formula representing the semantics of an action sequence in the case of functional composition with forward effects. We continue by shortly describing the optimizations applied to our composition tool, some of them inspired by Conformant-FF, others characteristic to our setting. We conclude the section with an overview of the future steps in designing our tool.

#### 3.5.1 Adapting the generation of formulas

Assume a  $\mathcal{WSC}|_{fwd}$  task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$  with fixed arity and without contradictory actions and a sequence  $act = \langle A_1, \dots, A_n \rangle$  of parallel actions. Assume also that any two actions  $a, a' \in act$  are compatible, i.e. either  $E_a \cap E_{a'} = \emptyset$ , or  $eff_a = eff_{a'}$ . In order to determine the known and negatively known propositions in a belief state, and therefore be able to test the satisfiability of the goal formula, we construct a CNF corresponding to the semantics of the respective action sequence.

We introduce a new predicate  $exists(constant, time)$ , useful for keeping track of the generated constants at different points along the execution of the action sequence. Further on, in order to keep the number of generated clauses polynomial, we introduce a predicate  $appl(action, time)$ . By definition,  $appl(a, t) = \bigwedge_{l \in pre_a} l \wedge \bigwedge_{c \in C_a} exists(c, t) \wedge \bigwedge_{e \in E_a} \neg exists(e, t)$ , its truth value being therefore completely determined by the action preconditions and exists predicates. Except for these predicates, there is no need for timestamps, because once an action has generated its outputs, the properties remain fixed.

Our CNF, denoted  $\phi(act)$  is obtained as follows. We initialize  $\phi(act)$  with the CNF that results from instantiating the background theory  $\mathcal{T}$  with  $C$ , where  $C$  is the union of  $C_0$  and all output constants appearing in  $\langle A_1, \dots, A_n \rangle$ . Next, we insert the initial ground literals from  $\phi_0$ . Since we know which constants exist at time 0, we add to our formula  $\bigwedge_{c \in C_0} exists(c, 0) \wedge \bigwedge_{c \in C \setminus C_0} \neg exists(c, 0)$ .

For each parallel action  $A_t$ ,  $1 \leq t \leq n$ , and for each action  $a \in A_t$ , we insert the clauses defining applicability. Therefore, we add:  $\bigvee_{l \in pre_a} \neg l \vee \bigvee_{c \in C_a} \neg exists(c, t-1) \vee \bigvee_{e \in E_a} exists(e, t-1) \vee appl(a, t)$ . For the opposite direction, we insert for each literal  $l \in$

pre<sub>a</sub> a clause  $\neg appl(a, t) \vee l$ , for each constant  $c \in C_a$  a clause  $\neg appl(a, t) \vee exists(c, t-1)$ , and for each constant  $e \in E_a$  a clause  $\neg appl(a, t) \vee \neg exists(e, t-1)$ .

Further, for each literal  $l \in \text{eff}_a$  we add the effect axiom clause  $\neg appl(a, t) \vee l$ . For each output constant  $e \in E_a$ , we add an axiom clause  $\neg appl(a, t) \vee exists(e, t)$ . These clauses should be read as implications: if action  $a$  is applicable at moment  $t-1$ , then  $\text{eff}(a)$  is known to hold, respectively each  $e \in E_a$  is known to exist at  $t$ .

Frame axioms are required for the *exists* predicate. For every constant  $c \in C \setminus C_0$ , we insert a negative frame axiom clause stating that if  $c$  did not exist at  $t-1$ , and none of the actions  $a \in A_t$  that could create it was applicable at  $t$ , then it does not exist at  $t$ :  $exists(c, t-1) \vee \bigvee_{a \in A_t, c \in E_a} appl(a, t) \vee \neg exists(c, t)$ .

We complete the time step by inserting for each constant  $c \in C$  a positive frame axiom clause  $\neg exists(c, t-1) \vee exists(c, t)$  (once created, every constant will exist until time step  $n$ ).

In the following, we will prove that testing whether a sequence  $act = \langle A_1, \dots, A_n \rangle$  of parallel actions is a solution for a given  $\mathcal{WSC}|_{fwd}$  task can be reduced to a single satisfiability test.

Given a  $\mathcal{WSC}|_{fwd}$  task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$  with fixed arity and without contradictory actions, a possible initial world  $w_0$  is defined by the pair  $(C_{w_0}, I_{w_0})$ , where  $C_{w_0}$  is the set of initial constants  $C_0$ , and  $I_{w_0}$  is an interpretation of the predicates  $\mathcal{P}$  over the constants  $C_{w_0}$ .  $w_0$  is possible, and therefore  $I_{w_0} \models \mathcal{T} \wedge \phi_0$ , with the quantifiers restricted to  $C_{w_0}$ .

**Proposition 3.7** *Assume a  $\mathcal{WSC}|_{fwd}$  task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$  with fixed arity and without contradictory actions, and a sequence  $act = \langle A_1, \dots, A_n \rangle$  of parallel actions. Assume a possible initial world  $w_0$  defined by the pair  $(C_{w_0}, I_{w_0})$ . Then there exists a single satisfying assignment  $\sigma$  of  $\phi(act)|_{w_0}$  ( $\phi(act)$  where the truth values of propositions over  $\mathcal{P}$  and  $C_0$  has been set to the one in  $I_{w_0}$ ).*

**Proof** Given fixed values for all propositions over  $\mathcal{P}$  and  $C_0$ , the axioms explicitly enforce a truth value for all predicates that appear in the formula (including *exists* and *appl* predicates) at each point in time. In particular, at time step 0 the values are given by  $I_{w_0}$  and  $C_0$ . For all other time steps  $1 \leq t \leq n$ , the value can either be set by an effect axiom, or (in case of the *exists* predicate) it can remain the same due to the frame axioms.

An execution of  $act$  is by definition a pair  $(\bar{C}, \bar{I})$ , where  $\bar{C}$  is an array of sets of constants, and  $\bar{I}$  is an array of interpretations, each  $I_t$  ( $0 \leq t \leq n$ ) being an interpretation of the predicates  $\mathcal{P}$  over the corresponding set of constants  $C_t$ .

**Proposition 3.8** *Assume a  $\mathcal{WSC}|_{fwd}$  task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$  with fixed arity and without contradictory actions, and a sequence  $act = \langle A_1, \dots, A_n \rangle$  of parallel actions. For any execution of  $act$  given by a pair  $(\bar{C}, \bar{I})$ , it is sufficient to consider only  $I_n$ , since  $I_0 \subseteq I_1 \subseteq \dots \subseteq I_n$ . We rename  $I_n$  to  $I$ . An execution of  $act$  is therefore the pair  $(\bar{C}, I)$ .*

**Proof** The proof that  $I_0 \subseteq I_1 \subseteq \dots \subseteq I_n$  follows directly from the definition of a result state in the forward effects case, since  $I_{t+1}|_{C_t} = I_t$ ,  $0 \leq t < n$ .

**Proposition 3.9** *Assume a  $\mathcal{WSC}|_{fwd}$  task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$  with fixed arity and without contradictory actions, and a sequence  $act = \langle A_1, \dots, A_n \rangle$  of parallel actions. Given a possible initial world  $w_0$ , and a literal  $l$  corresponding to a predicate in  $\mathcal{P}$ , if  $\phi(act)|_{w_0} \wedge (\neg l \vee \bigvee_{c \in Arg(l)} \neg exists(c, n))$  is unsatisfied then  $l$  holds after all executions  $(\bar{C}, I)$  that originate from  $w_0$ , or in other words  $l$  holds in all interpretations  $I$ , and all constants appearing in  $l$  are included in  $C_n$ .*

**Proposition 3.10** *Assume a  $\mathcal{WSC}|_{fwd}$  task  $(\mathcal{P}, \mathcal{T}, \mathcal{O}, C_0, \phi_0, \phi_G)$  with fixed arity and without contradictory actions, and a sequence  $act = \langle A_1, \dots, A_n \rangle$  of parallel actions, each  $A_t$  representing a compatible set of actions. Then  $act$  is a plan iff  $\phi(act) \wedge \neg \exists X. (\bigwedge_{l \in \phi_G[X]} l \wedge \bigwedge_{x \in X} exists(x, n))$  is unsatisfiable.*

### 3.5.2 Reusing knowledge

Both entailment of the goal and of the applicability of an action can be checked using a single SAT call. As an alternative, these tests can be decomposed into several tests for individual propositions. The advantage is that the knowledge about the propositions can be stored along the path and reused. The disadvantage is that the tests themselves become more costly.

Similar to Conformant-FF, the knowledge about propositions can be used for simplifying the formulas by inserting the values of the propositions known to be true or false. In case of  $\phi(act)$ , we further determine proposition values using the unit propagation procedure.

More precisely, for each ground literal  $l$  known to be true and, in case of  $\phi(act)$ , for each unit clause (a clause that contains a single ground literal  $l$ ), the formula is simplified by applying the following rules:

1. every clause containing  $l$  is removed
2. in every clause that contains  $\neg l$  this literal is deleted

Obtaining an empty clause through this process is equivalent to an unsatisfiable formula, and has different consequences, depending on the purpose of the formula to be simplified. For example, if an empty clause is obtained while simplifying  $\phi(act)$ , then the corresponding search state is invalid and therefore aborted.

**Decomposing tests** In the following, we describe separately the test decomposition for the goal and the action applicability.

In case of the goal, when using a single SAT call, the CNF to be tested is the one introduced in Proposition 3.10:

$$\phi(act) \wedge \forall X. (\bigvee_{l \in \phi_G[X]} \neg l \vee \bigvee_{x \in X} \neg exists(x, n)),$$

where  $n$  is the last timestep of  $act$ .  $act$  is a solution if the formula is unsatisfiable, and not a solution otherwise.

When storing knowledge about propositions, this formula is reduced according to the simplification rules. For the part that corresponds to the negated goal two straightforward cases may occur: if an empty clause is obtained the formula is unsatisfiable and we have reached a plan; if an empty formula is obtained, the entire formula is satisfiable, and  $act$  is not a plan.

The reduced formula that results can be decomposed in the following manner. For each variable substitution  $S$  of variables in the goal with constants belonging to  $C$  ( $C$  being the union of initial constants  $C_0$  and output constants appearing in  $act$ ), the original test contains a clause of the form:  $\bigvee_{l \in \phi_G[S]} \neg l \vee \bigvee_{c \in S} \neg exists(c, n)$ . The reduced formula will contain only a subset of these clauses, and for each of those, only the literals whose value is unknown. For each ground literal  $g$  in such a reduced clause, whether it is a  $\neg l$  or an  $\neg exists(c, n)$ , we test  $\phi(act)_r \wedge g$ :

- if satisfied, no new information is derived. Since the whole clause is satisfied, we continue with the next clause.
- if unsatisfied,  $\neg g$  is known to hold (and the value of the corresponding proposition is stored). If there is a further ground literal in the clause, we continue, otherwise the entire clause is unsatisfied and therefore have found a plan.

One complete test of the simplified formula must be performed, due to the fact that the actions to be composed are partial matches, and therefore are allowed to cover only parts of the goal. Obviously,  $act$  is a plan if this formula is unsatisfiable.

In the case of applicability of an action  $a$ , the single CNF corresponds to  $\phi(act) \wedge \bigwedge_{l \in \text{pre}(a)} l \wedge \bigwedge_{c \in C_a} exists(c, n) \wedge \bigwedge_{e \in E_a} \neg exists(e, n)$ , where  $n$  is either the step before the last step of  $act$ , if  $a$  is added in parallel, or the last step, if  $a$  is added in sequence. The action is applicable if the formula is satisfiable; not applicable otherwise.

Similar to the case of goal testing, we first simplify the formula. For the part that represents the action applicability, we check again for the trivial cases: if an empty clause is obtained the action is not applicable; if an empty formula is obtained, the action can be applied. If none of the straightforward cases occur, we perform the following tests. For each unknown ground literal  $g$  from  $\text{pre}(a)$ ,  $exists(c, n)$  or  $\neg exists(e, n)$ , we test  $\phi(act)_s \wedge g$ .

- if satisfiable, no new information is derived. If there is a further ground literal, we continue; otherwise, the action is applicable.
- if unsatisfiable,  $\neg g$  is known to hold. The action is not applicable and we abort the test.

Here as well, we need to perform one last complete test of the simplified formula.

**Computing formulas incrementally** The properties of a search state defined by an action sequence  $act$  are always computed in an incremental fashion. This is of particular

importance for the CNF corresponding to the search state,  $\phi(act)$ , and the propositions that are known to be true or false in the state.

For practical reasons, we do not compute  $\phi(act)$  in the standard manner described in section 3.5.1, except for the initial state. The initial state corresponds to an empty  $act_0$ , therefore  $\phi(act_0)$  will contain only the instantiation of background theory  $\mathcal{T}$  with  $C_0$ , the initial literals and for each  $c \in C_0$  an exists statement.

The CNF for a search state  $S_{i+1}$  that results from applying a parallel action  $A$  to state  $S_i$ , is created by appending to the CNF of  $S_i$  the following clauses:

- the theory instantiated with tuples that contain at least one new constant
- frame axioms for all new constants for every step  $t < \text{current timestep}$
- action applicability clauses and effect clauses for every action  $a$  in the parallel action  $A$
- frame axioms for all constants for the current timestep

For each state, including the initial state, we also simplify the formula using the propositions whose values are known. These values of propositions are stored along the path that corresponds to  $act$ . As already mentioned, in case of  $\phi(act)$  unit propagation procedure is applied, and therefore the simplification process may result in new knowledge.

## 4 Syntactic Composition

After successful semantic query, discovery and functional composition, the results are fed into a BPMO view of a framework that, after transformation to different views, synthesises and deploys a corresponding, executable process. Figure 2 gives a simplified overview of the syntactic composition framework.

Within this section we will introduce the syntactic composition framework and its components in a bottom-up order. This is to say we will start with the foundations of service-oriented systems. We will first discuss the topic of workflow languages briefly, look at appropriate runtime environments for process execution and finally discuss a model driven approach for syntactic composition.

### 4.1 Workflow Languages

Web services have become widely accepted as the de-facto standard for distributed business applications [7].

They bring maximum interoperability and use an open and flexible architecture. The implementation and complexity of a web service can be hidden towards a caller. Layered on top of these services, BPEL, the de-facto standard for orchestration of web services that evolved out of the Web Services Flow Language (WSFL) [19] and XLang [31], formally describes processes [10].

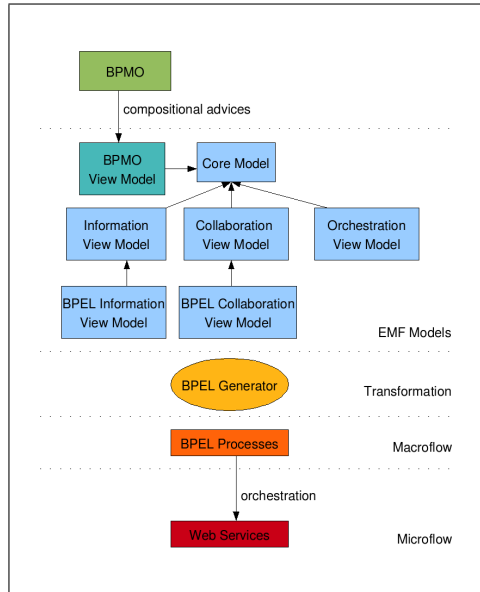


Figure 2: Overview of the Composition Framework

While external activities within BPEL correspond to web services, human interaction within a process is not covered by BPEL [4]. As a consequential service oriented separation of concerns, IBM and SAP introduced the WS-BPEL Extension for People (BPEL4People) [2] as an extension for BPEL defining human participant integration points together with WS-HumanTask [1]. Figure 3 shows a stack of different web service technologies and standards on respective layers.

## 4.2 Process Engine

The process engine, as the final runtime environment, is responsible for executing the resulting process. It manages states, permits monitoring and persists relevant data for recovery or analysis. Several commercial as well as open source BPEL engines are available and can serve as the runtime environment for BPEL processes.

Most common BPEL engines do not yet support BPEL4People and WS-HumanTask out of the box. In order to permit and realize human interaction within business processes our motivation for an implementation thus was to design a system that manages human aspects while transparently interacting with a traditional BPEL engine. Indeed as BPEL4People really is an extension to BPEL, we chose to design VieBOP<sup>6</sup>, a system that can interact with arbitrary BPEL engines while hosting BPEL4People specific information and managing BPEL processes on a *people* level.

Figure 4 shows VieBOP within the composition framework.

<sup>6</sup>Vienna BPEL for People

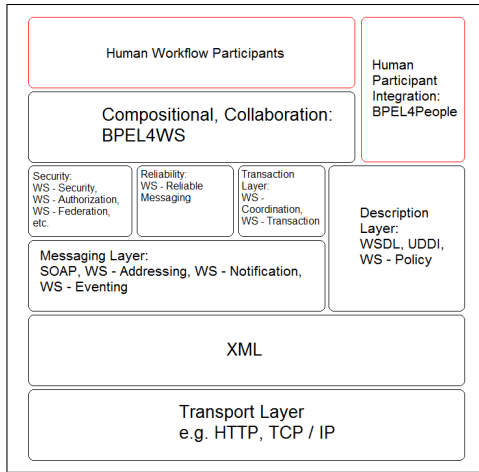


Figure 3: A Web Service Stack

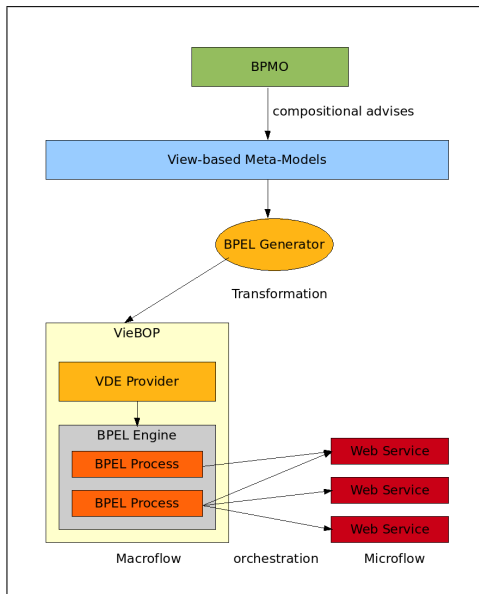


Figure 4: VieBOP within the Composition Framework

### 4.3 VDE Provider

After the composition and the code generation of the process, interaction with the runtime environment is realized by a Validation / Deployment / Execution (VDE) Provider. Such a provider offers services for validating processes on a semantic level, for deploying the processes on a process engine and for executing it. An abstract VDE provider contains generic code that can be used by inherited providers that are specialized for specific process engines.

We have chosen BPEL as the workflow language of our choice and intend to develop VDE providers for the open source ActiveBPEL Engine [5] and Apache Orchestration Director Engine (Apache ODE) [8].

#### 4.3.1 Validation

Post validation of BPEL processes on a semantic level is not covered by the SemBiz project as the resulting generated processes are expected to be semantical correct after semantic discovery and composition. A lot of research is dedicated to this topic and various approaches exist in order to validate workflows. For example semantic validation of petri nets can take place after transforming BPEL accordingly [40]. A foundational unifying framework based on the  $\pi$ -calculus that could be integrated into the VDE component of VieBOP has been developed in [24] and [25]. The results can be used both for a better understanding of the BPEL semantics and behavior and for developing conceptual and software tools able to detect process equivalences leading to flow design simplification.

#### 4.3.2 Deployment

In order to deploy processes, customized solutions have to be realized due to the lack of standardized deployment descriptors. We present how deployment can be realized on two different open source BPEL engines.

**ActiveBPEL** The ActiveBPEL engine uses a deployment archive file that after creation can be deployed by calling a web service or by copying it to a directory of the process engine in the filesystem.

The deployment archive is a java archive with the extension of `.bpr` and contains the following subdirectories:

- `bpel`
- `META-INF`
- `wSDL`
- `partners`

At the root level of the archive a process descriptor with the extension of `.pdd` is placed. `META-INF` holds a `wsdlCatalog.xml` that references WSDLs and XML Schemata. The BPEL Process is stored in the `bpel` directory and the optional partners directory may hold `.pdef` files with partner link definitions.

**Apache ODE** The Apache ODE engine requires a different approach for deploying processes. The following steps will be performed by a VDE Provider for Apache ODE:

- Creation of a temporary service unit directory for the BPEL processes.
- Placement of the relevant `.bpel`, `.wsdl` and `.xsd` files into the temporary directory.
- Creation of an ODE deployment descriptor (`deploy.xml`) and placement into the temporary directory.
- (Optional) Compilation of the BPEL processes.
- Compression of the contents of the temporary directory into a service unit archive.
- Creation of a temporary service assembly directory.
- Placement of the service unit archive in a temporary service assembly directory.
- Update of the service assembly's `jbi.xml` descriptor.
- Compression of the contents of the service assembly directory into a service assembly archive.
- Copy of the service assembly archive into the appropriate deploy directory.

### 4.3.3 Execution

As mentioned in Section 4.2 the process engine, that hosts the process to be executed, acts as the runtime environment for its execution. At deployment time it reads the XML based BPEL document and offers a web service for process invocation. After invocation the workflow executes and necessary web services are called as external activities in the specified order.

Process execution thus takes place when calling the deployed BPEL processes web service.

## 4.4 Modeling Framework [32]

Inspired by the concept of *architectural views*, that are representations of a system from the perspective of a related set of *concerns* [20], we suggest a view-based approach to modeling of process-driven SOAs. Namely, perspectives on business process models and service interactions – as the most important concerns in process-driven SOA – are used as central views in our approach. The approach is extensible with all kinds of other views. In particular, our approach offers separated views, in which each of them

represents a certain part of the processes and services, such as the collaboration view, the information view, the orchestration view, etc. These views can be viewed separately to get a better understanding of a specific concern, or they can be integrated to produce a richer view or a thorough view of the processes and services.

Technically, our concepts are realized using the model-driven software development (MDSD) paradigm [36]<sup>7</sup>. We have chosen this approach to integrate the various view models into one model, and to automatically generate platform-specific or executable code in WSDL, BPEL, or Java. MDSD is also used to separate these platform-specific views from the platform-neutral views and the integrated views, so that business experts do not have to deal with platform-specific details. The code generation process is driven by model transformations from view models or integrated models into executable code.

#### 4.4.1 Overview of the modeling framework

In this section, we briefly introduce the view-based modeling framework. The framework consists of modeling elements such as a *meta-meta-model*, *meta-models*, and *views* (see Figure 5(a)). As mentioned in the previous section, a view is a representation of a process from the perspective of related *concerns*. In our framework, a view is specified using an adequate framework's meta-model. Each meta-model is a (semi-)formalized representation of a particular business process concern. Therefore, the meta-model specifies entities and their relationships that can appear in the correspondent view. The meta-models, in turn, are defined on top of the *meta-meta-model*. Figure 6(a) shows the relevant excerpt of the meta-meta-model of the Eclipse Modeling Framework [12] (i.e., Ecore meta-model) that we used to define our meta-models.

In our approach we categorize distinct activities – in which the modeling elements are manipulated (see Figure 5(b)):

- *Design* is used to define new architectural views.
- *Extend* is used to create a new meta-model by adding more features to an existing meta-model, or by developing it from scratch (e.g., to add a new formalization of a certain business process concern to the framework).
- *Integrate* is used to combine views to produce a richer view or a thorough view of a business process.
- *Transform* is used to generate executable code from one or many architectural views.

Before generating outputs, *Transform* and *Integrate* validate the input views against relevant meta-models. *Extend* and *Integrate* are the most important activities used to broaden our view-based model-driven framework toward various dimensions. Existing meta-models can be enhanced using the *extension mechanisms* or can be combined using the meta-model-level *integration mechanisms* as we will see.

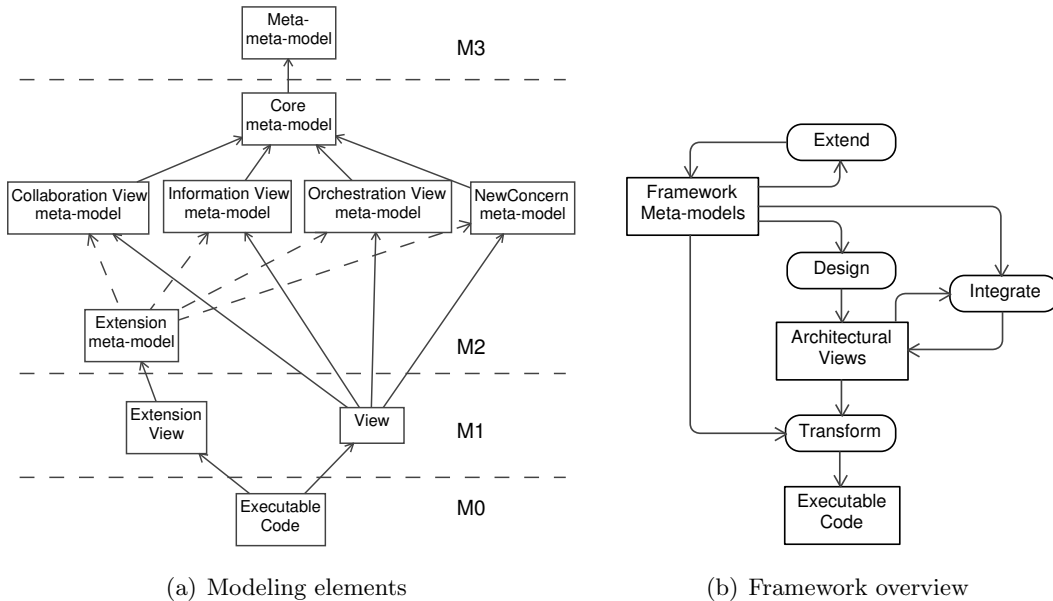


Figure 5: View-based model-driven framework

#### 4.4.2 View-based modeling framework

A business process often contains various concerns that require support of modeling approaches. We firstly concentrate on modeling of the basic concerns of a business process, namely, *orchestration*, *information*, and *collaboration* (see Figure 5(a)). However, our view-based modeling framework is not only bound to the above-mentioned concerns but also open and extensible to allow other concerns such as transactions, event handling, security, quality of service, etc., to be plugged in using the same approach. In the next sections, we present in detail (semi-)formalized representations of the process's concerns summarized above in terms of relevant meta-models along with the discussion of extensibility mechanisms, namely, *extension* and *integration*.

**The Core meta-model** To enhance the extensibility, we devise a basic meta-model, namely, the *Core* meta-model as a foundation for the other meta-models (see Figure 6(b)). Each of the other meta-models is defined by extending the *Core* meta-model. Therefore, the meta-models are independent of each other. The *Core* meta-model is the place where the relationships among the meta-models are maintained. Accordingly, the relationships in the *Core* meta-model are needed for both view- and meta-model-level integrations as described in Section 4.4.2.

The *Core* meta-model consists of a number of abstract meta-classes such as *View*, *Process*, *Service*, and *Element*. These entities are cornerstones of our modeling framework. Each of them can be extended further. At the heart of the *Core* meta-model is the *View* meta-class that captures the central view concept. Each specific view (i.e. each

<sup>7</sup>Please note that the OMG's MDA proposal is one specific MDSD approach.

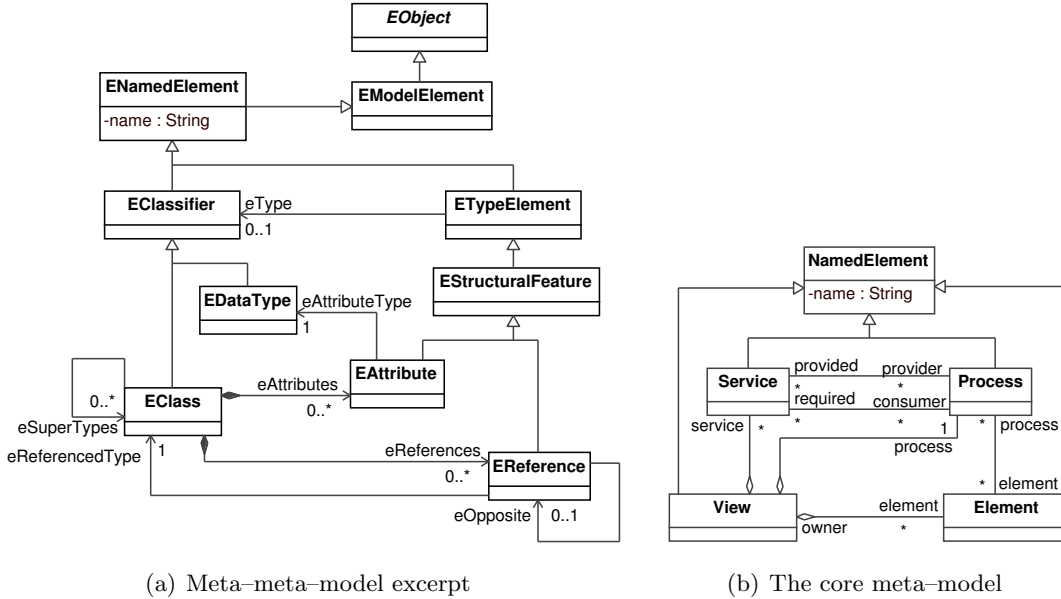


Figure 6: Meta-meta-model and the *Core* meta-model

instance of the View meta-class) represents a perspective on one *Process*. It consists of a number of *Services* representing the external functions the business process provides or requires, and a number of *Elements* representing the objects that appear inside the process. Because the meta-models represent concerns of a business process, they are mostly derived from the core meta-model, and the *Service* and *Element* meta-classes are the most important extension points. Moreover, the hierarchical structures in which those meta-classes are roots can be used to define the *integration points* used to combine meta-models (see Section 4.4.2).

**Orchestration view meta-model** Orchestration is one of the most important concerns of a SOA process. An orchestration view comprises many activities and control structures. The activities are process tasks such as service invocations, or data handling, while control structures describe the execution order of the activities to achieve a certain goal. Each orchestration view is specified based on the orchestration view meta-model.

There are several approaches to modeling process’s orchestration such as state-charts, block structures [28], activity diagrams [21], Petri-nets [33], and so on. Despite this diversity in control flow modeling, it is well accepted that existing modeling languages share five basic patterns: *sequence*, *parallel split*, *synchronization*, *exclusive choice*, and *simple merge* [35, 38, 34]. Thus, we adopted these patterns as the building blocks of our orchestration meta-model. Other, more advanced patterns can be added later by using *extension mechanisms* discussed in Section 4.4.2 to augment the orchestration model.

The control structures of BPEL [28], such as *sequence*, *flow*, and *switch*, are more or less equivalent to the aforementioned patterns. The issue here is that the semantics of

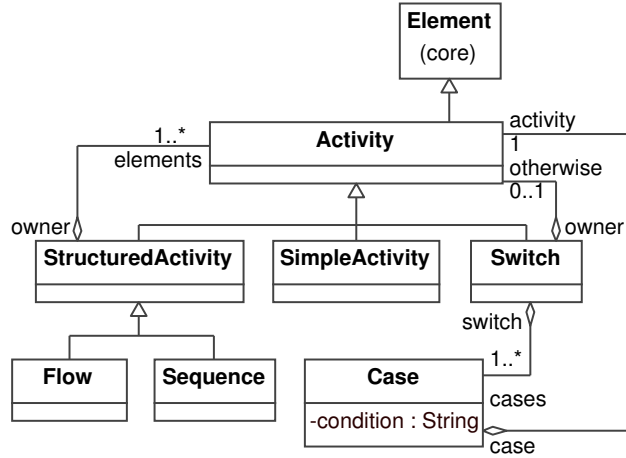


Figure 7: Orchestration view meta-model

Structure	Description
Sequence	An activity is only enabled after the completion of another activity in the same sequence structure. The sequence structure is therefore equivalent to the semantics of the <i>Sequence</i> pattern.
Flow	All activities of a flow structure are executed in parallel. The subsequent activity of the flow structure is only enabled after the completion of all activities in the flow structure. The semantics of the flow structure is equivalent to a control block starting with the <i>Parallel Split</i> pattern and ending by the <i>Synchronization</i> pattern.
Switch	Only one of many alternative paths of control inside a switch structure is enabled according to a condition value. After the active path finished, the process continues with the subsequent activity of the switch structure. The semantics of the switch structure is equivalent to a control block starting with the <i>Exclusive Choice</i> pattern and ending by the <i>Simple Merge</i> pattern.

Table 1: Semantic of control structures

BPEL’s structures is not as clear and precise as the semantics of the patterns. Therefore, instead of re-inventing a new orchestration meta-model we built our meta-model on the basic BPEL control structures, and define their semantics more strictly (see Table 1).

The primary entity of the orchestration meta-model is the *Activity* meta-class (see Figure 7) which is the base class for other meta-classes such as *Sequence*, *Flow*, and *Switch*. Another important entity in the orchestration meta-model is the *SimpleActivity* meta-class that represents a concrete action such as a service invocation, a data processing task, etc. The actual description of each *SimpleActivity* is modeled in another specific view. For instance, a service invocation is described in a *collaboration view*, while a data processing action is specified in an *information view*. Each *SimpleActivity* is a placeholder or a reference to another activity, i.e., an interaction, or a

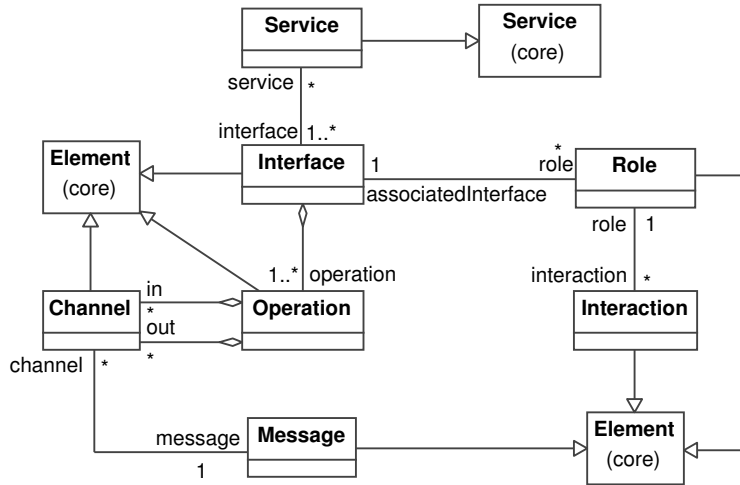


Figure 8: Collaboration view meta-model

data processing task. Therefore, every SimpleActivity becomes an integration point to combine an orchestration view with an information view, or with a collaboration view (see *integration mechanisms* in Section 4.4.2).

**Collaboration view meta-model** A business process is often developed by composing the functionality provided by various parties such as services or other processes. Other partners, in turn, might use the process. All business functions required or provided by the process are exposed in terms of standard interfaces (e.g., WSDL portTypes). We captured these concepts in the *Core* meta-model by the relationships between the two elements *Process* and *Service*. The collaboration view meta-model extends the *Core* meta-model to represent the interactions between the business process and its partners.

In the collaboration view meta-model, the *Service* meta-class from the *Core* meta-model is extended by a tailored *Service* meta-class that exposes a number of *Interfaces*. Each *Interface* provides some *Operations*. An *Operation* represents an action that might need some inputs and produces some outputs via correspondent *Channels*. The details of each data element are not defined in the *collaboration* view but in the *information* view. Therefore, a *Channel* holds a reference to a *Message* entity. Each *Message* becomes an *integration point*, that can be used to combine a specific collaboration view with an information view (see Section 4.4.2).

The ability and the responsibility of an interaction partner are modeled by the *Role* meta-class. Every partner – who provides the relevant interface associated with a particular role – can play that role. An interaction between the process and any partner is represented by the *Interaction* meta-class that associates with a specific *Role* of that partner.

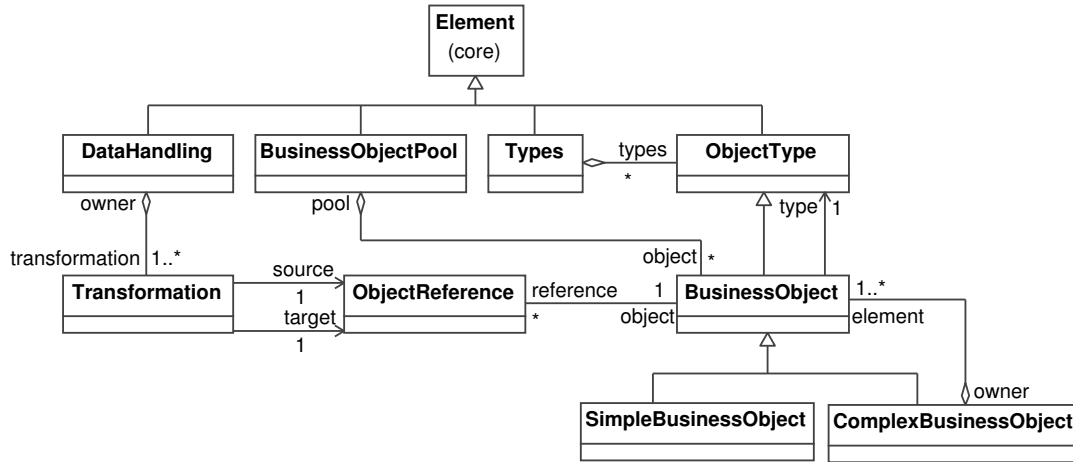


Figure 9: Information view meta-model

**Information view meta-model** The third basic concern we considered in modeling a business process is *information*. This concern is (semi-)formalized by the *information view meta-model* (see Figure 9). This meta-model involves the representation of data object flows inside the process, and message objects traveling back and forth between the process and the external world.

In the information view meta-model, the *BusinessObject* meta-class, which has the type *ObjectType*, is the abstraction of any piece of information, for instance, a purchase order received from the customer, or a request sent to a banking service to verify the customer’s credit card, etc. Each piece of information might be a *SimpleBusinessObject*, or a *ComplexBusinessObject* that consists of a number of *BusinessObjects*. We define the *BusinessObjectPool* meta-class as a generic container for a number of *BusinessObjects*.

Messages exchanged between the process and its partners, or data flowing inside the process might go through some *Transformations* that convert or extract existing data to form new pieces of data. The transformations are performed inside a *DataHandling* object. The source or the target of a transformation is an *ObjectReference* entity that holds a reference to a certain *BusinessObject*.

**Extension mechanisms** The aforementioned meta-models are the cornerstones to create architectural views like orchestration-, collaboration-, and information-views. Our framework is not limited to these concerns but it allows other concerns to be plugged in via *extension points*. An extension point is any entity that can add additional features (e.g., attributes or relations) to construct a new entity. Using relationships, such as *generalization*, *extend*, etc., we can gradually *refine* an existing meta-model toward another meta-model at a lower abstraction level. For instance, the *orchestration view*, *collaboration view*, and *information view* meta-models are mostly extensions of the Core meta-model using the generalization relation. We also demonstrate the extensibility of the collaboration view meta-model by an enhanced meta-model, namely, the *BPEL-*

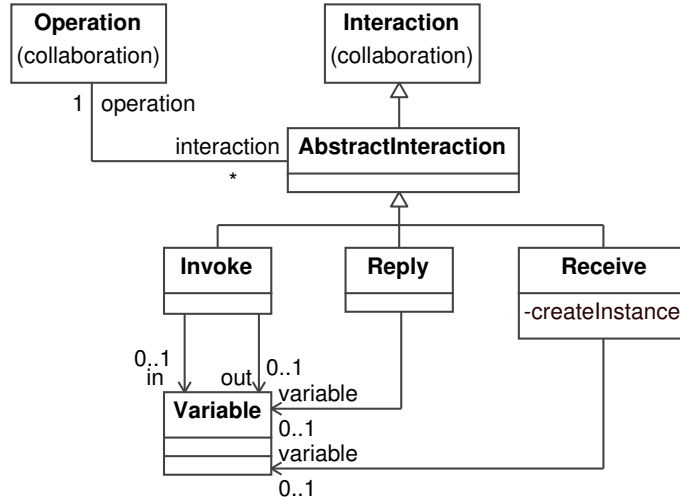


Figure 10: An extension of the collaboration view

*Collaboration* extension (see Figure 10). Similar BPEL specific view extensions have also been developed for the information and orchestration view (omitted here for space reasons). In the same way, more specific meta-models for other technologies can be derived. In addition, any other business process concern, such as transactions, event handling, and so on, can be (semi-)formalized by a new meta-model derived from the common meta-meta-model using the same approach as used above.

**Integration mechanisms** In our approach, the orchestration view – as the most important concern in process-driven SOA – is often used as the central view. Views can be integrated via *integration points* to provide a richer view or a thorough view of the business process (see Algorithm 1).

**Definition** Let  $M_1, M_2$  be two meta-models (i.e., derived from the *Core* meta-model). If the entities  $m_1 \in M_1$  and  $m_2 \in M_2$  extend the same entity of the *Core* meta-model,  $m_1$  and  $m_2$  are **conformable**.

**Definition** Given  $M_1, M_2$  are two meta-models and  $V_1, V_2$  are two views conforming to  $M_1$  and  $M_2$ , respectively. An **integration point** between  $V_1$  and  $V_2$  is a tuple  $I(v_1, v_2 | v_1 \in V_1, v_2 \in V_2, v_1 = \text{instanceOf}(m_1), v_2 = \text{instanceOf}(m_2))$ , and  $m_1$  and  $m_2$  are *conformable*, such that  $V_1$  can be merged with  $V_2$  – at the position of  $v_2$  into that of  $v_1$ .

The *GetIntegrationPoint* function receives as input an entity  $v_1 \in V_1$  and a view  $V_2$ . It looks for  $v_2 \in V_2$  such that  $(v_1, v_2)$  is an *integration point* between  $V_1$  and  $V_2$ . This function can be implemented based on name-based matching, class hierarchical structures, or ontology-based structures. The name-based matching mechanism might be effectively used at the view level (or model level) because from a modeler’s point of

---

**Algorithm 1:** View integration algorithm

---

**Input:** View  $V_1$ , view  $V_2$   
**begin**  
  **foreach** *Entity*  $v_1 \in V_1$  **do**  
     $v_2 = \text{GetIntegrationPoint}(v_1, V_2)$ ;  
    **if** ( $v_2 \neq \text{NULL}$ ) **then**  
       $v_1.\text{add}(v_2.\text{eAttributes})$ ;  
       $v_1.\text{add}(v_2.\text{eReferences})$ ;  
    **end**  
  **end**  
**end**

---

---

**Algorithm 2:** Name-matching algorithm

---

**Input:** Entity  $v_1 \in V_1$ , view  $V_2$   
**Output:** Entity  $v_2 \in V_2$  or NULL  
**begin**  
   $Found = \text{FALSE}$ ;  
  **while** *NOT Found* **do**  
     $v_2 = \text{getNextEntity}(V_2)$ ;  
    **if**  $v_2.\text{name} == v_1.\text{name}$  **then**  $Found = \text{TRUE}$   
  **end**  
  **if**  $Found$  **then** return  $v_2$  **else** return NULL  
**end**

---

view, it makes sense and is reasonable to give the same name to the modeling entities which pose the same functionality and semantics. To demonstrate the view integration idea, we present a simple implementation of the name-based matching mechanism (Algorithm 2) for the *GetIntegrationPoint* function.

To create an integrated view – as the result of view integration – a correspondent meta-model of the view has to be defined first. That meta-model is also used later to validate or transform the integrated view into code. Therefore, an adequate integration at the meta-level is needed for any view integration or integrated view transformation. We can use the same approach as used for view integration. However, at the meta-model level, name-based matching is not sufficient. The reason is that the relationships between meta-classes are mostly hierarchical, and the meta-classes that have the same name might not be *conformable*. Therefore, class hierarchical structures are used at the meta-level to define the integration points in our framework. We proposed the *meta-level integration mechanism* using the class hierarchical relationship to define the *meta-level integration points*.

**Definition** Given  $M_1, M_2$  are two meta-models based on a common meta-meta-model. A tuple  $MI(m_1, m_2 | m_1 \in M_1, m_2 \in M_2)$  is a **meta-level integration point** iff  $m_1$  and  $m_2$  are instances of the same entity of the meta-meta-model and  $M_1$  can be integrated with  $M_2$  by merging the model structure at the position of  $m_2$  into that of  $m_1$ .

**Model transformations** There are two basic types of model transformations: model-to-model and model-to-code. A model-to-model transformation maps a model conforming to a given meta-model to another kind of model conforming to another meta-model. Model-to-code, so-called code generation, produces executable code from a certain model.

In our framework, the model transformations are mostly model-to-code that take as input one or many views and generate code in executable languages, for instance, Java, BPEL, WSDL, etc. In the literature there are numerous code generation techniques such as templates+filtering, template+meta-model, inline generation, code weaving, etc. [36]. In our prototype, we used the template+meta-model technique – which is realized in the openArchitectureWare framework (oAW) [29] to implement the model transformations. But any of above-mentioned techniques can be utilized in our framework with reasonable modifications.

## 5 Conclusion

We have presented semantic query, discovery and functional composition of BPMO processes and introduced a view based modeling framework for composition.

Concerning functional composition of BPMO processes, we have introduced a formalism for functional composition, and identified a special case for which no belief revision is necessary. In our current step we are developing a tool for the special case, inspired by Conformant-FF, a scalable tool from the area of planning under uncertainty. More precisely, at this point, we have adapted the Conformant-FF search space representation to the special case. We will continuously refine this tool. One of the most challenging tasks will be to guide the search using heuristics and filtering techniques.

Moreover, a view-based modeling framework has been presented where results of the functional composition will be fed into a BPMO excerpt view. A model to model transformation then will populate different views like the orchestration, information and collaboration view. BPEL and WSDL is generated by model to code transformation using these populated models. Final deployment is performed by a VDE provider that interacts with a concrete BPEL engine, the runtime environment for executing the processes on a macroflow scale. External activities on a microflow scale can be integrated into the processes as standalone web services.

### 5.1 Further Work

**Query & Discovery** Adopting the type of discovery for suiting the needs of functional composition as mentioned in Section 2.2 poses a crucial challenge.

**Functional Composition** One of the immediate next steps in refining the composition tool is to design and implement algorithms for creating new actions. In the current phase, in every search state all possible actions are generated. This is however very unfeasible, and corresponding algorithms can be devised to prune the irrelevant actions.

One of the most challenging tasks moreover will be to devise heuristics and filtering techniques, based on a relaxation of the planning problem. An important issue is that, in difference to us, Conformant-FF (and indeed every existing planning tool) does not allow the generation of new constants. We will devise new heuristics for dealing with this. Note that this is critical: as already indicated, exponentially many constants may be generated in general, so one needs heuristics identifying which are important. Those heuristics need to be clever: if they remove too many constants, then the solutions may be cut out; if they remove too few constants, then the search space may explode.

**Syntactic Composition** The BPMO view for taking the results of the functional composition as well as the model-to-model transformation for populating the other views, so that model-to-code transformation can take place, will be developed and questions related to this will be addressed in [9].

## References

- [1] *Web Services Human Task (WS-HumanTask), Version 1.0*, June 2007.
- [2] *WS-BPEL Extension for People (BPEL4People), Version 1.0*, June 2007.
- [3] A. ANKOLENKAR ET AL. DAML-S: Web service description for the semantic web. In *ISWC (2002)*.
- [4] A JOINT IBM-SAP WHITEPAPER. *WS-BPEL Extension for People*, August 2005.
- [5] ACTIVE ENDPOINTS. Activebpel engine. <http://www.activebpel.com>.
- [6] AKKIRAJU, R., SRIVASTAVA, B., ANCA-ANDREEA, I., GOODWIN, R., AND SYEDA, T. Semaplan: Combining planning with semantic matching to achieve web service composition. In *ICWS (2006)*.
- [7] ANZBÖCK, R., DUSTDAR, S., AND GALL, H. Software configuration, distribution, and deployment of web-services. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering* (New York, NY, USA, 2002), ACM Press, pp. 649–656.
- [8] APACHE SOFTWARE FOUNDATION. Apache ode. <http://ode.apache.org>.
- [9] CONSORTIUM, S. D2.3 prototype implementation. SemBiz Deliverable, April 2008.
- [10] CURBERA, F., KHALAF, R., MUKHI, N., TAI, S., AND WEERAWARANA, S. The next step in web services. *Commun. ACM* 46, 10 (2003), 29–34.
- [11] D. MARTIN ET AL. OWL-S: Semantic markup for web services. In *SWSWPC (2004)*.
- [12] ECLIPSE. Eclipse Modeling Framework. <http://www.eclipse.org/emf/>, 2006.

- [13] EITER, T., FABER, W., LEONE, N., PFEIFER, G., AND POLLERES, A. A logic programming approach to knowledge-state planning, II: The DLVK system. *AI 144*, 1-2 (2003), 157–211.
- [14] FENSEL, D., LAUSEN, H., POLLERES, A., DE BRUIJN, J., STOLLBERG, M., ROMAN, D., AND DOMINGUE, J. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer-Verlag, 2006.
- [15] GEREVINI, A., SAETTI, A., AND SERINA, I. Planning through stochastic local search and temporal action graphs. *J. Artificial Intelligence Research 20* (2003), 239–290.
- [16] GIUNCHIGLIA, E., LEE, J., LIFSCHITZ, V., MCCAIN, N., AND TURNER, H. Non-monotonic causal theories. *AI 153*, 1-2 (2004), 49–104.
- [17] HOFFMANN, J., AND BRAFMAN, R. Conformant planning via heuristic forward search: A new approach. *AI 170*, 6–7 (May 2006), 507–541.
- [18] HOFFMANN, J., AND NEBEL, B. The FF planning system: Fast plan generation through heuristic search. *JAIR 14* (2001), 253–302.
- [19] IBM SOFTWARE GROUP. *Web Services Flow Language (WSFL) version 1.0*, 2001.
- [20] IEEE. Recommended Practice for Architectural Description of Software Intensive Systems. Tech. Rep. IEEE-std-1471-2000, IEEE, 2000.
- [21] ISO/IEC. *Unified Modeling Language (UML), v1.4.2*, April 2005.
- [22] KELLER, U., LARA, R., LAUSEN, H., POLLERES, A., AND FENSEL, D. Automatic location of services. In *Proceedings of the 2nd European Semantic Web Symposium (ESWS2005)* (Heraklion, Crete, 5 2005).
- [23] LAUSEN, H., DE BRUIJN, J., POLLERES, A., AND FENSEL, D. WSML - A Language Framework for Semantic Web Services. In *Proceedings of the W3C Workshop on Rule Languages for Interoperability* (April 2005).
- [24] LUCCHI, R., AND MAZZARA, M. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming 70*, 1 (2007), 96–118.
- [25] MAZZARA, M., AND LANESE, I. Towards a unifying theory for web services composition. In *Web Services and Formal Methods* (2006), no. 4184, Springer Berlin / Heidelberg, pp. 257–272.
- [26] MAZZARA, M., URBANEC, A., HOFFMANN, J., AND SIRBU, A. D2.1 state of the art and requirements on semantic querying, discovery, and composition. SemBiz Deliverable, February 2007.
- [27] MCDERMOTT, D. V. Using regression-match graphs to control search in planning. *Artificial Intelligence 109*, 1-2 (1999), 111–159.
- [28] OASIS WEB SERVICES BUSINESS PROCESS EXECUTION LANGUAGE (WSBPEL) TC. *Web Service Business Process Execution Language Version 2.0*, January 2007.

- [29] OPENARCHITECTUREWARE.ORG. openArchitectureWare project. <http://www.openarchitectureware.org>, 08 2002.
- [30] PONNEKANTI, S., AND FOX, A. SWORD: A developer toolkit for web services composition. In *WWW (2002)*.
- [31] THATTE, S. *Web Services for Business Process Design*. Microsoft Corporation, 2001.
- [32] TRAN, H., ZDUN, U., AND DUSTDAR, S. View-based and model-driven approach for reducing the development complexity in process-driven soa. In *Int. Conf. on Business Process and Services Computing (BPSC)* (Leipzig, Germany, Sept. 2007).
- [33] VAN DER AALST, W., DESEL, J., AND OBERWEIS, A., Eds. *Business Process Management: Models, Techniques, and Empirical Studies - Lecture Notes in Computer Science*, vol. 1806. Springer-Verlag, 2000.
- [34] VAN DER AALST, W. M., DUMAS, M., TER HOFSTEDÉ, A. H., AND WOHED, P. Pattern Based Analysis of BPMN (and WSCI). Tech. rep., FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.
- [35] VAN DER AALST, W. M. P., TER HOFSTEDÉ, A. H. M., KIEPUSZEWSKI, B., AND BARROS, A. P. Workflow patterns. *Distributed and Parallel Databases 14*, 1 (2003), 5–51.
- [36] VÖLTER, M., AND STAHL, T. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.
- [37] WINSLETT, M. Reasoning about actions using a possible models approach. In *AAAI (1988)*.
- [38] WOHED, P., VAN DER AALST, W. M., DUMAS, M., AND TER HOFSTEDÉ, A. H. Pattern Based Analysis of BPEL4WS. Tech. rep., FIT-TR-2002-04, Queensland University of Technology, Brisbane, 2002.
- [39] YAN, Z., CIMPIAN, E., AND MAZZARA, M. D3.1 sembiz bpmsuite requirements analysis and design. SemBiz Deliverable, February 2007.
- [40] YANG, Y., TAN, Q., AND XIAO, Y. Verifying web services composition based on hierarchical colored petri nets. In *IHIS '05: Proceedings of the first international workshop on Interoperability of heterogeneous information systems* (New York, NY, USA, 2005), ACM Press, pp. 47–54.