DYNAMIC SERVICE COMPOSITION: A PETRI-NET BASED APPROACH

Michael Köhler, Daniel Moldt, Jan Ortmann
University of Hamburg, Computer Science Department
Vogt-Kölln-Str. 30, D-22527 Hamburg

Keywords: Web Services, High-level Petri Nets, Semantic Service Composition, Ontologies.

Abstract: Dynamic service composition requires a formal description of the services such that an agent can process these descriptions and reason about them. The amount of detail needed for an agent to grasp the meaning of a service may lead to clumsy specification.
Petri nets offer a visual modeling technique for processes, that offers a refinement mechanism. Through this, a specification can be inspected on the level of detail needed for a given objective.
In this paper we introduce a Petri net based approach to capture the semantics of services by combining Petri nets ideas from the description logic area focusing on ontologies. The resulting framework can than be used by agents to plan about activities involving services.

1 INTRODUCTION

The dynamic composition of services requires a detailed understanding of the effects the invocation of a particular service has. This makes it necessary to have a formal semantics that can be interpreted by machines.

There is always a tradeoff between formal and precise semantics of a model and ease of modeling. Assuming that a semantic description always has to be formal, we chose Petri nets as a conceptual background, which is rather intuitive because of its visual notation and the possibility to refine models. We consider three aspects of semantic service modeling of major importance:

- A common vocabulary shared by the participants provided by a common ontology,
- a description of the functional behavior of single Web services or parts thereof, and
- a description of the coordination of multiple services.

An agent has some knowledge about the state of the world it is located in. To change this state the (software) agent can communicate to the world via message passing (i.e. through the invocation of other services). These messages have to be understood by both sides which is guaranteed by a common ontology. (For further information see (Staab and Studer, 2004).) The functional behavior will be described by the change of the interpretation of an algebraic specification. This is related to ideas of abstract state machines (Börger and Stärk, 2003). Finally the coordination and sequence in which the services are executed are modeled by a Petri net.

The combination of services has different aspects. The services need to be invoked in the right order and the right services showing the right behavior need to be invoked. In this paper we will focus on the operational aspects of behavior and will refer to e.g. (van der Aalst, 2003) or (Kindler et al., 2000) for an discussion of when services and workflows fit together.

A Web service interprets the messages according to a given ontology. Here, we make use of *Description Logics* (Baader et al., 2002) as the formal basis for the ontology. To specify the functionality of a service, we use *algebraic* constructs (Ehrig and Mahr, 1985). By the execution of the different possible processes of a net, an agent can then pick a sequence of services best suited for a given objective. Planning can thus be considered as reaching a particular marking from a given one and is hence closely related to the planning as model checking approach as e.g. in (Giunchiglia and Traverso, 1999).

The paper starts with an example we will use to illustrate the main idea in Section 2. Hereafter follows a brief introduction to Description Logics in Section 3 before we introduce Service Description Nets in Section 4. We will then illustrate the idea with the exam-

Köhler M., Moldt D. and Ortmann J. (2006).
DYNAMIC SERVICE COMPOSITION: A PETRI-NET BASED APPROACH.
In *Proceedings of the Eighth International Conference on Enterprise Information Systems - SAIC*, pages 159-165
DOI: 10.5220/0002494901590165
Copyright © SciTePress

ple and give a short overview of our implementation in Section 5 and will finally draw a conclusion and give an outlook in Section 6.

2 MOTIVATIONAL EXAMPLE

Dynamic Web service orchestration is useful in a variety of different application areas. Here, we will focus on the management of ship-supplies as an illustrating example. Imagine a ship entering different ports all over the world. The only time when it can purchase supplies is while mooring in a port, which is normally only a short period of time. Hence the delivery of all supplies needs to be coordinated in advance such that there is no delay. There might be different providers offering different prices for the same goods. Additionally we might have unexpected needs. For example, due to heavy weather nobody attended the restaurant, so that there is less need of food, but the weather caused a fuel shortage.

If we access the different providers of goods by Web services, we would certainly like to know, what happens if we chose a particular provider. This is commonly given by a textual description which cannot be understood by a machine. Thus we need a more formal description of what happens. In the case of ship supplies we have hard constraints such as the delivery time and we have things we would like to be maximized or minimized such as quality and price. What we need to know of a service is therefore what effects it has on the particular part of the world that we are interested in. Having a representation of the world, we would like to know, how this representation changes. To allow a communication in general, all parties involved need to comply with a common vocabulary given by an ontology.

Different services can be classified based on there behavior. If two providers offer the same item at different prices we will normally put them in the same category of services. This can be reflected by a classification of services in a process ontology. The calling agent accesses a discovery service with a process ontology to retrieve a list of services matching a certain class. Through their semantic description the agent can now determine, which of the services suits its needs best.

3 ONTOLOGIES AND DESCRIPTION LOGICS

Ontologies specify the concepts that are used in a given domain. A concept is determined by its attributes and by the roles existing between itself and the other concepts. Concept descriptions are built inductively from a set of primitive concepts, roles and attributes by using concept and role constructors. The most widespread approach is to model the concepts of an ontology by a *Description Logic* (DL). Different Description Logics vary in their expressiveness, i.e. in the number and kind of constructors they provide. We distinguish between attributes as functional roles and roles (non-functional). An example of a concept description would be the definition of a concept
LibraryBook $\doteq$ Book $\sqcap$ $\forall$signature.Signature $\sqcap$ $\forall$subject.String

Description logics knowledge bases are divided into TBoxes and ABoxes. (For an in-depth discussion on DL see (Baader et al., 2002).)

TBoxes (terminology boxes) hold intensional knowledge in form of a terminology. They are built through declarations of general properties of concepts and thus restrict the set of possible worlds. More formally, given a DL $\mathcal{L}$, a $\mathcal{L}$-TBox $\mathcal{T}$ is a finite set of concept definition axioms of the form $A \doteq C$, with A being a concept symbol and C being a $\mathcal{L}$-concept description (the defining concept).

ABoxes (assertion boxes) on the other hand allow to describe a specific state of the world. They hold extensional knowledge that is specific to the individuals of the domain of discourse. A *concept assertion* is denoted $C(i)$ where C is a concept description and i is an individual. A role assertion is denoted $R(i_1, i_2)$ where R is a role symbol and i_1, i_2 are individuals. An ABox $\mathcal{A}$ is a finite set of concept and role assertions. ABoxes are defined with respect to some TBox. The LibraryBook example above would be part of the TBox $\mathcal{T}$ whereas the instance *LibraryBook(book43)* would be part of an ABox wrt. $\mathcal{T}$.

An *interpretation* $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consisting of a domain $\Delta^{\mathcal{I}} \neq \emptyset$ and an interpretation function $\cdot^{\mathcal{I}}$, that assigns a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ to every concept symbol A, a partial function $a^{\mathcal{I}} : \Delta^{\mathcal{I}} \rightarrow \Delta^{\mathcal{I}}$ to every attribute symbol a, and a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to every role symbol R.

Let C and D be concept descriptions. D *subsumes* C, denoted as $C \sqsubseteq D$, iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for all interpretations $\mathcal{I}$. $C \equiv D \Leftrightarrow C \sqsubseteq D \wedge C \sqsupseteq D$ denotes the *equivalence* of two concept descriptions.

4 MODELING OF SERVICE BEHAVIOR USING NETS

To model scenarios like the one given in Section 2 we introduce Service Description nets. They have been discussed more formally in (Köhler and Ortmann, 2005). Service Description nets (SD nets) can be mapped to algebraic Petri nets. This allows us to adopt the analysis techniques introduced for them

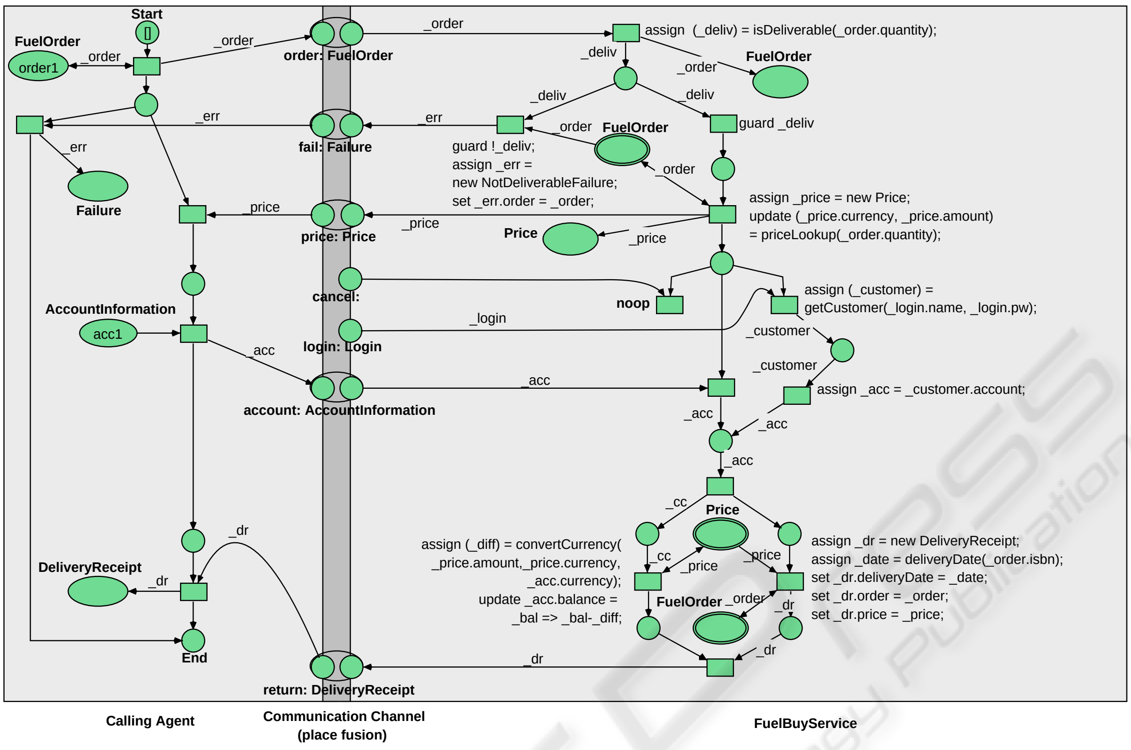

Figure 4.1: An Example for a Service Description Net.

(Reisig, 1991).

A Petri net is a bipartite graph consisting of places and transitions. More formally a Petri net $\mathcal{N}$ is a triple $\mathcal{N} = (P, T, F)$, where P is a finite set of places, T is a finite set of transitions with $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a flow relation (a set of arcs).

The basic idea behind Service Description nets is that each service affects the current state of the agents world by changing the value of data items. A service can either create new data items or it can modify the attributes of existing data items. The data items are represented by an ontology, which is based on Description Logics. Description Logics allows us to define concepts on a formal basis and is on the other hand well supported by tools. As described in Section 5 we integrated (PROTÉGÉ, 2005,) in our Petri net simulation engine (RENEW, 2005,) to have an executable model.

SD nets allow us to model the workflow of a Web service together with the data exchanged and manipulated. Hence we make use of workflow nets (van der Aalst, 2003) or more specifically of open workflow nets (Reisig et al., 2005). Workflow nets reflect the specific structure of workflows with a single starting point and a single end point. Open Workflow nets additionally introduce border places to model the incoming and outgoing messages of workflows. We further extend the expressiveness by allowing typed messages and by allowing the functional description of activities. The types are taken from an ontology. The concepts modeled in the ontology can be used in the net. The roles and attributes modeled in the ontology are hidden in the inscriptions. E.g. $C.x$ refers to the attribute x of a concept C.

To represent the concepts and the operations defined on them, we use order-sorted algebra (Goguen and Meseguer, 1992). Order-sorted algebra extends many-sorted algebras by a partial order relation, which the sorts need to obey. Here, it is used to define an order-sorted logical specification as follows:

Definition 1 (*Logical Specification, Structure*). A *logical specification* is a tuple $\mathbf{LS} = (S, \leq, \Omega, E, \Pi)$ where S is a set of sorts, $\leq$ is a partial order on the set of sorts, Ω is a set of operation symbols and E is a set of equations such that $(S, \leq, \Omega, E)$ is an order-sorted specification and $\Pi = (\Pi_w)_{w \in S^*}$ is a family of predicate symbols, which is disjoint from S and Ω.

A *structure* (or interpretation) of a logical specification $\mathbf{LS}$ is a triple $\mathcal{A}_{\mathbf{LS}} = (S_A, \Omega_A, \Pi_A)$ consisting of a family of sets $S_A = (A_s)_{s \in S}$ respecting $\leq$ and of operations $\Omega_A = (\omega_A)_{\omega \in \Omega}$ with $\omega_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ for all $\omega \in \Omega_{s_1 \ldots s_n, s}$ as well as a family of relations $\Pi_A = (\pi_A)_{\pi \in \Pi}$ with $\pi_A \subseteq A_{s_1} \times \ldots \times A_{s_n}$ for $\pi \in \Pi_{s_1 \ldots s_n}$. $\Diamond$

Using a logical specification we can now define the current state of an agent's world as formula in first order logic. This represents the ABox of our ontology. To state, that ship *vessel01* has 2000 gallons of fuel we might have the following fact in our knowledge base:

stored(vessel01,item04)$\wedge$*ItemQuantity(item04)*$\wedge$
quantity(item04,2000)$\wedge$*unit(item04,Gallon)*$\wedge$
type(item04,Diesel)

Naturally the knowledge base entry depends on the ontology. Here, the predicates represent the roles (and attributes) defined by an ontology. The operations defined on the data are specified by an algebra. Figure 4.1 shows a potential description of a ship supplies workflow as SD-net. Its underlying structure consists of two open workflows that yield a workflow net when composed by place fusion. The net on the left side represents the calling client whereas the net on the right side represents the Web service invoked. Some information is optional to invoke *FuelBuyService*. The invoking service can either cancel the operation, provide a *Login* or some new *AccountInformation*. In Figure 4.1 the latter alternative is chosen.

We say a logical specification is *compatible* with an ontology if each named concept has a corresponding sort, that respects the hierarchy defined by the ontology and each role has a least sort that is its domain among the set of sorts. Furthermore we require the predicate symbols to have the sorts of the corresponding roles. Through this, we can now define a net class changing the current concepts and attributes of an ontology by changing the interpretation of its logical specification, which we will call Service Description Nets. These nets have special inscriptions to change the interpretation of the predicates. Additionally we have variable assignments that assign the result of a term evaluation to a variable, so that we can create new items or use local variables.

We can now specify the syntax of the transition inscriptions. In the following we will let $\mathbb{T}_{\Sigma,s}(X)$ denote the set of terms of sort s over the set of variables X wrt. the signature Σ.

Definition 2 (*Assignment and Update, Consistency***).** Let $\mathbf{LS} = (S, \leq, \Omega, E, \Pi)$ be a logical specification and X a set of variables. Given $v \in X_s$ and $s, s' \in S$ a *variable assignment* is of the form

1. assign v = new s,
2. assign $v = \underline{t}'$ with $\underline{t}' \in \mathbb{T}_{\Sigma,s}(X)$, or
3. assign $v = \underline{t}'.\pi$ with $\underline{t}' \in \mathbb{T}_{\Sigma,s'}(X)$, $\pi \in \Pi_{s',s}$.

A *predicate update* for some $\pi \in \Pi_{s',s}$, $\underline{t} \in \mathbb{T}_{\Sigma,s'}(X)$ is of the form

4. update $\underline{t}.\pi = \underline{t}'$ with $\underline{t}' \in \mathbb{T}_{\Sigma,s}(X)$.

A set of variable assignments Ass (*Assignment set*) is *consistent* if no left hand side occurs more than once in Ass, i.e. (assign $v = rexp$) $\in Ass$ and (assign $v = rexp'$) $\in Ass$ implies $rexp = rexp'$. A set of predicate updates is *consistent* similarly if no left hand side occurs more than once in it. A set of assignments together with a set of updates is called a set of *action statements* Act or $Act(X)$ respectively if it ranges over a specific set of variables X. $\Diamond$

If a variable occurs on the left hand side of a variable assignment it is called bound, otherwise it is called free. The set of all bound variables of an assignment set Ass is denoted by $Bound(Ass)$.

The assignment statements define a natural extension α_{ext} of a variable assignment α as long as there are no circular dependencies in the assignment statements. Each free variable gets assigned the value of the term on the right hand side of the equation evaluated under $\bar{\alpha}$ or under the extended evaluation $\bar{\alpha}_{ext}$. The firing of a set of action statements is done by first evaluating the extended assignment, i.e. by evaluating all *assign* statements, and then executing the *update* statements.

Definition 3 (*Firing of Action Statements***).** Given an interpretation $\mathcal{A}_{\mathbf{LS}} = (S_A, \Omega_A, \Pi_A)$ of $\mathbf{LS}$ and an assignment $\alpha : X \to A$, where X is the set of all free variables. The firing of a set of action statements Act first evaluates all assignments yielding an extended assignment α_{ext}. Then the updates of the form $u =$ (update $\underline{t}.\pi = \underline{t}'$) are evaluated in $\mathcal{A}_{\mathbf{LS}}$. This yields a successor interpretation $\mathcal{A}'_{\mathbf{LS}} = (S_A, \Omega_A, \Pi'_A)$ such that we have $\Pi'_A = (\Pi_A \setminus \{\pi_A\}) \cup \{\pi'_A\}$ where $\pi'_A = (\pi_A \setminus \{(\bar{\alpha}_{ext}(\underline{t}), x)\}) \cup \{(\bar{\alpha}_{ext}(\underline{t}), \bar{\alpha}_{ext}(\underline{t}'))\}$ for some value x. $\Diamond$

In a consistent update set, each predicate update affects either a different predicate or a different value of the same predicate. Therefore we have the following lemma.

Lemma 4: For a consistent set of predicate updates the firing can be done in arbitrary order.

This allows us to speak of *the* firing of a consistent update set in an interpretation $\mathcal{A}_{\mathbf{LS}}$ which will lead to a successor interpretation. Now that we have modeled the change of attributes and roles by logical specification and the creation of objects by assignments, we can use this in a net, where we can define the order in which these changes occur.

Service Description Nets (SD Net) consist of typed places, arcs inscribed by variables and transitions with guards. Additionally transitions specify the change of the predicates. The control flow is modeled by a dedicated sort called $s_{\mathbf{token}}$ which has the constant $\bullet$ as its only operation symbol. If an arc or place has no inscription it is implied that it is of sort $s_{\mathbf{token}}$. Figure 4.1 shows an SD Net.

Definition 5 (*Service Description Net, Service Description Net system***).** Let $\mathbf{LS} = (S, \leq, \Omega, E, \Pi)$ be a logical specification compatible to an ontology $\mathcal{O}$ such that $\mathbf{LS}$ contains the concrete domain $\mathbb{B}$ of booleans as well as a sort $\bot_{undef}$, with $\bot \sqsubseteq \bot_{undef} \sqsubseteq C$ for all $C \neq \bot$.

A *Service Description Net* is a tuple $\mathcal{SDN} =_{def} (\mathbf{LS}, X, \mathcal{N}, d, W, G, act)$ where

- $\mathbf{LS} = (S, \leq, \Omega, E, \Pi)$ is a logical specification as defined in Def. 4,
- X is a set of $\mathbf{LS}$-variables,
- $\mathcal{N} = (P, T, F)$ is a Petri net,
- $d : P \to S$ is a place typing,
- $W : F \to X$ is an arc inscription such that for $p \in P$ and $f \in F$ we have $W(f) \in (X_s \setminus Bound(act(t)))$ with $d(p) \leq s$ if $f = (p, t)$ and $W(f) \in X_s$ with $s \leq d(p)$ if $f = (t, p)$,
- $G : T \to \mathbb{T}_{\Sigma,\mathrm{Bool}}(X)$ assigns an enabling condition to each transition, and
- $act : T \to Act(X)$ assigns a regular and consistent set of assignments and updates to each transition as defined in Def. 4.

A Service Description Net together with an initial state $Q_0 = (\mathbf{m}_0, \mathcal{I}^0_{\mathbf{LS}})$ consisting of an initial marking $\mathbf{m}_0 : P \to \mathbb{T}_\Sigma$ such that for $\mathbf{m}_0(p) \in \mathbb{T}_{\Sigma,s}$ we have $s \leq d(p)$ for all $p \in P$ and an initial interpretation $\mathcal{I}^0_{\mathbf{LS}} = (S_{I^0}, \Omega_{I^0}, \Pi_{I^0})$ of $\mathbf{LS} = (S, \leq, \Omega, E, \Pi)$ is called a *Service Description Net system*. $\diamondsuit$

To guarantee the executability of Service Description Nets, we require that a variable used on an output arc of a transition is either also used on an input arc or defined through an assignment, thus reducing the search space for possible bindings.

Definition 6 (*State, Marking, Enabling and Firing***).** A *marking* $\mathbf{m}$ of a Service Description Net $\mathcal{SDN}$ is a mapping $\mathbf{m} : P \to \mathbb{T}_\Sigma$ such that $\mathbf{m}(p) \in \mathbb{T}_{\Sigma,s}$ with $s \leq d(p)$ for $p \in P$. A *state* $Q = (\mathbf{m}, \mathcal{I}_{\mathbf{LS}})$ is a marking together with an interpretation $\mathcal{I}_{\mathbf{LS}} = (S_I, \Omega_I, \Pi_I)$ of $\mathbf{LS}$.

A transition t is *enabled* under a variable assignment $\alpha : X \to A$ iff the evaluation of the activation condition $\bar{\alpha}(G(t)) = true$ and for all places $p \in P$ we have $\mathbf{m}(p) \geq \bar{\alpha}(W(p, t))$ and α assigns a value to each free variable in $act(t)$.

The *firing* of a transition under a variable binding α is denoted as state transition $Q \xrightarrow{t,\alpha} Q'$ which is again split into $\mathbf{m} \xrightarrow{t,\alpha} \mathbf{m}'$ and $\mathcal{I}_{\mathbf{LS}} \xrightarrow{t,\alpha} \mathcal{I}'_{\mathbf{LS}}$. $\mathbf{m}'$ is defined for each place $p \in P$ as $\mathbf{m}'(p) = \mathbf{m}(p) - \alpha(W(p, t)) + \alpha(W(t, p))$. The interpretation $\mathcal{I}'_{\mathbf{LS}} = (S_I, \Omega_I, \Pi'_I)$ is defined as in Definition 4 by successively applying the predicate updates (update $\underline{t}.\pi = \underline{t}'$) $\in act(t)$ resulting in a sequence $\Pi_I = \Pi^0_I, \ldots, \Pi^n_I = \Pi'_I$ with $\Pi^{i+1}_I = (\Pi^i_I \setminus \{\pi^i_I\}) \cup \{(\pi^i_I \setminus \{(\bar{\alpha}(\underline{t}), x)\}) \cup \{(\bar{\alpha}(\underline{t}), \bar{\alpha}(\underline{t}'))\}\}$ for some value x. $\diamondsuit$

For convenience reasons we introduce two abbreviating notions for Service Description Net inscriptions:

5. update $\underline{t}.\pi = v' \Rightarrow \underline{t}'$, which is equivalent to the assignment assign $v' = \underline{t}'.\pi$ together with the predicate update update $\underline{t}.\pi = \underline{t}'$, and
6. $\underline{t}.\pi$ within a term expression, which is equivalent to assign $v' = \underline{t}'.\pi$ and the usage of v' instead.

The second abbreviation allows us to write e.g.

assign v = _book.author.name instead of
assign v1 = _book.author and assign v = v1.name

Additionally we require a repository containing a subset of A where assignments of the form (assign v = new s) can retrieve an $a \in A_s$ that has not been previously used in the net. Practically this is done by a pool, from which an element is taken. For most practical scenarios we have a finite set of items, so that we can determine an upper bound as the size of the pool.

Furthermore it is often convenient in practice to have not only binary relations but relations of arbitrary arity. Here, we will only consider functional relations and call them predicates as well. For an n-ary predicate π we additionally introduce

assign $(v_1, \ldots, v_j) = \pi(t_1, \ldots, t_i)$, $(i + j = n)$

which replaces the value of $v_1, \ldots, v_i$ by appropriate values such that $(t_1, \ldots, t_i, v_1, v_j) \in \pi_{\mathbf{LS}}$.

Through this, we can express database queries or external queries that assign values to more than one variable. In Figure 4.1 such an assignment operation would be `priceLookup`. For these assignments the definition of the logical specification has to be modified, to allow n-ary predicates.

There are some specialties about the net definition above. We allow only one concept instance on each place. Having more than one concept instance on a single place results in picking one instance nondeterministicly, if more than one instance satisfies the guards of a transition. Nondeterminism, however, is rarely intended and only complicates planning. Additionally by restricting the number of tokens on a place to one, the underlying net becomes safe and will be easier to analyze. This restricts the expressiveness of the net on the one hand, but makes planning and analyzing easier on the other hand. Although the change of roles (i.e. binary relations) could also be modeled by Transform nets as defined above, we omitted them, since we did not introduce flexible assignments. Therefore, in Figure 4.1 only attributes have been modeled.

5 EXAMPLE REVISITED

Let us return to the SD Net in Figure 4.1 again. We will now explain the first steps the net might take. We have the following ontology and the following initial facts in our knowledge base:

Ontology (TBox)	
FuelOrder $\doteq$	Order $\sqcap$ quantity.ItemQuantity $\sqcap$ type.FuelType
AccountInformation $\doteq$	currency.Currency $\sqcap$ balance.Decimal $\sqcap$ owner.Person $\sqcap$ acc_number.integer $\sqcap$ bank_code.integer
NotDeliverableFailure $\doteq$	Failure $\sqcap$ order.Order
Price $\doteq$	amount.Decimal $\sqcap$ currency.Currency
DeliveryReceipt $\doteq$	deliveryDate.Date $\sqcap$ order.Order $\sqcap$ price.Price
...	

Facts (Abox) (initial)
FuelOrder(order1)
order1.quantity = quant1
order1.type = Diesel
ItemQuantity(quant1)
quant1.quantity = 200
quant1.unit = Gallon
AccountInformation(acc1)
acc1.currency = Euro
acc1.balance = 40000
...

After the message with the order arrives at the service, the service looks up, whether the order quantity can be delivered. If it cannot be delivered, the calling agent receives a *NotDeliverableFailure*. If it can be delivered, the agent receives a *DeliveryReceipt* and the account corresponding to the *AccountInformation* the agent has sent will have a reduced balance. After the execution of the net, the knowlegde base might contain the following statements:

Facts (Abox) (final)
FuelOrder(order1)
order1.quantity = quant1
order1.type = Diesel
ItemQuantity(quant1)
quant1.quantity = 200
quant1.unit = Gallon
AccountInformation(acc1)
acc1.currency = Euro
acc1.balance = 39200
DeliveryReceipt(dr1)
dr1.deliveryDate = date1
dr1.order = order1
dr1.price = price1
date1.day=05
date1.month=12
price1.amount=800
price1.currency=Euro
...

The agent can now calculate the difference and reason about what happens when it executes the service. In this simple scenario we only have one service. It is of course possible to have a much more complex scenario with many different services. Another major advantage of Petri nets as underlying formalism is their support for *refinement*. This allows us to build classes of net fragments that correspond to classes of services. Saying that a *BuyService* always involves some kind of payment and some kind of delivery, we can now refine this basic structure in different ways resulting in an ontology of services – an approach very similiar to the one in (OWL Services Coalition, 2005).

The approach described in this paper has been prototypically implemented using (RENEW, 2005,) and (PROTÉGÉ, 2005,). The SAND (Service Agent Net Description) Plug-in is a plug-in of the RENEW Petri net simulation engine which allows to model and execute Service Description Nets. It makes use of the PROTÉGÉ API to import ontologies which can then be used within an SD net. These ontologies can be either PROTÉGÉ files or OWL ontologies. Since the latter is a standardized exchange format, this format is strongly encouraged. The current attribute values of the net can be inspected by clicking on the corresponding tokens in the simulation. This allows a user to visually follow the steps and their impact on the data. SAND can automatically evaluate different nets and can return the one best suited for a given task to the user. Through this, an agent can select or preselect workflows with the desired qualities. As for now this planning is simply carried out by a depth-first search, which, however, may not terminate if the underlying workflow net has an infinite reachability graph.

6 CONCLUSION AND OUTLOOK

In this paper we have presented a formal framework to model processes with data based on description logics, algebra and Petri nets. We see this as a step toward a semantic description of processes as it is needed in the Semantic Web services and agent context.

Predicate/Transition nets (Genrich, 1986) have as well used petri nets to model the modification of logical terms. Service Description Nets, however, focus more on the manipulation of information represented by the terms defined in an ontology.

Since Service Description Nets are closely related to algebraic nets, analyzing techniques for algebraic nets as e.g. discussed in (Reisig, 1991) can be applied to Service Description Nets as well. Additionally, this relationship guarantees the executability of these nets. Further investigation has to show if the special structure of Service Description Nets results in other interesting properties and analysis techniques. Furthermore it should be discussed to what extend it is sensible to forget about the algebraic structure and reduce Service Description Nets to S/T-Nets or workflow nets to study their behavioral properties.

As further research we will classify services according to their behavior and their interface. This will give us a semantic classification of Web services which will make planning much faster by selecting only services from particular classes. Another topic is the improvement of the planning and the conversion of the representation to some XML dialect such that we can import the process description of other formats (e.g. OWL-S).

REFERENCES

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., editors (2002). *Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, Cambridge (Mass.).

Börger, E. and Stärk, R. (2003). *Abstract State Machines – A Method for High-Level System Design and Analysis*. Springer, Berlin et.al.

Ehrig, H. and Mahr, B. (1985). *Equations and initial semantics; Fundamentals of algebraic specification*. EATCS Monographs on Theoretical Computer Science. Berlin.

Genrich, H. J. (1986). Predicate/transition nets. In Brauer, W., Reisig, W., and Rozenberg, G., editors, *Advances in Petri Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247.

Giunchiglia, F. and Traverso, P. (1999). Planning as model checking. In Biundo, S. and Fox, M., editors, *ECP*, volume 1809 of *Lecture Notes in Computer Science*, pages 1–20.

Goguen, J. A. and Meseguer, J. (1992). Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273.

Kindler, E., Martens, A., and Reisig, W. (2000). Interoperability of workflow applications: Local criteria for global soundness. In van der Aalst, W. M. P., Desel, J., and Oberweis, A., editors, *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 235–253.

Köhler, M. and Ortmann, J. (2005). Formal aspects for semantic service modeling based on high-level petri nets. In Proceedings of the International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC) 2005.

OWL Services Coalition (2005). OWL-S 1.1 Release. Technical report, ˙ www.daml.org/services/owl-s/1.1/.

PROTÉGÉ (2005). The protégé ontology editor and knowledge acquisition system. http://protege.stanford.edu/. Provides references to programs, source code and documentation of the Protégé tool set.

Reisig, W. (1991). Petri nets and algebraic specifications. In Jensen, K. and Rozenberg, G., editors, *High-level Petri Nets – Theory and Application*, pages 137–170, Berlin, Heidelberg. Springer.

Reisig, W., Schmidt, K., and Stahl, C. (2005). Kommunizierende workflow-services modellieren und analysieren. *Informatik - Forschung und Entwicklung*.

RENEW (2005). RENEW – the reference net workshop. http://www.renew.de/. Provides references to programs, source code and documentation of the Renew tool set.

Staab, S. and Studer, R., editors (2004). *Handbook on Ontologies*. International Handbooks on Information Systems.

van der Aalst, W. M. P. (2003). Inheritance of interorganizational workflows: How to agree to disagree without loosing control? *Inf. Tech. and Management*, 4(4):345–389.