Using Goals for Flexible Service Orchestration
A First Step*

M. Birna van Riemsdijk Martin Wirsing

Ludwig-Maximilians-Universität München, Germany
{riemsdijk, wirsing}@pst.ifi.lmu.de

Abstract. This paper contributes to a line of research that aims to apply agent-oriented techniques in the field of service-oriented computing. In particular, we propose to use goal-oriented techniques from the field of cognitive agent programming for service orchestration. The advantage of using an explicit representation of goals in programming languages is the flexibility in handling failure that goals provide. Moreover, goals have a close correspondence with declarative descriptions as used in the context of semantic web services. This paper now presents first steps towards the definition of a goal-based orchestration language that makes use of semantic matchmaking. The orchestration language we propose and its semantics are formally defined and analyzed, using operational semantics.

1 Introduction

This paper contributes to a line of research that aims to apply agent-oriented techniques in the field of service-oriented computing. Services are generally defined as autonomous, platform-independent computational entities that can be described, published, and discovered. An important concern in service-oriented computing is how services can be composed in order to solve more complex tasks. One way to go about this, is to use a so-called *orchestration language* such as WS-BPEL [10] or Orc [7], by means of which one can specify an executable pattern of service invocations. Another important issue in the context of services is dealing with *failure* [14, 6]. Especially when services are discovered at run-time, one needs to take into account that a service might not do exactly what one had asked for, or that a particular orchestration does not yield the desired result.

We argue that the agent community has something to offer to the services community, as agents are meant to be capable of flexible action in dynamic environments. Being capable of flexible action means that the agent should be able to cope with failure, and should respond adequately to changed circumstances. The idea is now that using agent-oriented techniques in orchestration languages could potentially yield more adaptive and flexible orchestrations.

In this paper, we focus in particular on the usage of *(declarative) goals* as is common in agent programming (see, e.g., [22, 5, 20]), for flexible service orchestration. Goals as used in agent programming describe situations that the agent

* This work has been sponsored by the project SENSORIA, IST-2005-016004.

wants to reach. The use of an explicit representation of goals in a programming language provides for added flexibility when it comes to failure handling, as the fact that a goal represents a desired state, can be used to check whether some plan for achieving a goal has failed. This is then combined with a mechanism for specifying which plan may be used for achieving a certain goal in certain circumstances, which also allows for the specification of multiple plans for achieving a certain goal.

When considering the application of goal-oriented techniques to services, especially *semantic web services* seem to have a natural relation with goals. The idea of semantic web services is to endow web services with a declarative description of what the service has to offer, i.e., a declarative description of the semantics of the service. This then allows *semantic matchmaking* [16] between the declarative description of a service being sought, which in our case would correspond with a goal of an agent, and a description of the service being offered. In fact, the WSMO framework for web service discovery refers explicitly to semantic matching between goals and web services [19].

This paper now describes first steps towards the design of a *goal-based orchestration language that makes use of semantic matchmaking*, building on research on goal-oriented agent programming, orchestration, and semantic web services. This orchestration language and its semantics are formally defined and analyzed. The orchestration language and the description of services are based on propositional logic. Being a first step towards combining cognitive agent programming languages and orchestration languages, the relative simplicity of propositional logic allows us to focus on the essential aspects of an agent-based orchestration language.

Proposals for combining agent-oriented and service-oriented approaches are appearing increasingly often in the literature. In the CooWS platform [4], for example, an agent-based approach to procedural learning is used in the context of web services. Other approaches focus on communication protocols [13, 2]. The relation between goals as used in agent programming and semantic web services has also been pointed out in [9], in which an architecture is described that focuses on the translation of high-level user goals to lower-level goals that can be related to semantic service descriptions.

To the best of our knowledge, however, this is the first proposal for a formally defined language for goal-based orchestration. Flexibility in handling failure is an issue that is well-recognized in service composition, and it is exactly this flexibility that goal-based techniques can provide to orchestration languages. The main technical contribution of this paper is that we provide a clean and formally defined way of combining service-oriented and agent-oriented models of computation.

2 Syntax

In this section, we describe the syntax and informal semantics of our goal-oriented orchestration language. The way we go about combining goal-oriented

techniques and orchestration languages, is that we take (a family of) cognitive agent programming languages as a basis, and incorporate into these constructs for service orchestration. The family of agent programming languages on which we build are variants of the language 3APL as also reported in [20], and we moreover draw inspiration from the language AgentSpeak(L) [18]. The orchestration language that we take as a basis is the language Orc [7].

In (goal-oriented) agent programming languages, agents generally have a belief base, representing what the agent believes to be the case in the world, and a goal base, representing the top-level goals of the agent. Agents execute plans in order to achieve their goals. These plans, broadly speaking, consist of actions that the agent can execute, and so-called subgoals. In order to achieve these subgoals, an agent needs to select plans, just like it selects plans to achieve its top-level goals. Selecting plans for (sub)goals is done by means of so-called *plan selection rules* that tell the agent which plan they may use for achieving which goal, given a certain state of the world.

The general idea of our goal-oriented orchestration language now is that an agent may use not only actions that it can execute itself for achieving its goals, but it can also call services. We thus extend the language of plans with a construct for calling services. These services may be called directly using the service name, or may be discovered, based on the goal that the agent wants to achieve. One of the main technical issues that arises when modifying an agent programming language in this way, is that it has to be determined how to handle the results that are returned by services in a way that is in line with the (goal-oriented) model of computation of agent programming languages. This will be explained in more detail in the sequel.

In order to illustrate our approach, we use a very simple example scenario that is adapted from [23]. In the car repair scenario, the car's diagnostic system reports a failure so that the car is no longer drivable. Depending on the problem, the car may be repaired on the spot, or it might have to be towed to a garage. The car's discovery system then calls a repair service in order to try to repair the car on the spot. Alternatively, it may identify and contact garages and towing truck services in the car's vicinity in order to get the car towed to a garage.

In Section 2.1, we define how services are described, and in Section 2.2 we define the syntax of the orchestration language.

2.1 Service Descriptions

The way we describe services is based on the Ontology Web Language for Services (OWL-S) [12] which seeks to provide the building blocks for encoding rich semantic service descriptions. In particular, we focus on the description of functional properties of services. According to OWL-S, these can be described in terms of *inputs*, *outputs*, *preconditions*, and *effects* (so-called IOPEs). The idea is that a service is "a process which requires inputs, and some precondition to be valid, and it results in outputs and some effects to become true" [12].

The inputs description specifies what kind of inputs the service accepts from a caller, and the outputs description specifies what the service may return to

the caller. Preconditions are conditions on the state of the world, which need to hold in order for the service to be able to execute successfully. Effects describe what the service can bring about in the world, i.e., the effects description is like a description of post-conditions.

In this paper, the inputs description is a set of atoms. The set of atoms represents what kind of formulas the service is able to handle (similar to a type specification). The outputs description is a set of propositional formulas or the special atom *failure*. Informally, the idea is that the formulas in the outputs description represent alternative possible outputs, where the actual output may also be a combination of these alternatives (see Definition 9 for a precise specification). The effects are described in a similar way.

Below, we formally define service descriptions. A service description has a name, and inputs, outputs, preconditions, and effects. The name of the service may not be the reserved name d. The name d is used to express in plans that a service should be discovered.

Definition 1 *(service description)* Throughout this paper we assume a language of propositional logic $\mathcal{L}$ with typical element ϕ that is based on a set of atoms Atom, where $\top, \bot \in \mathsf{Atom}$ and $\mathit{failure} \notin \mathsf{Atom}$. Moreover, we define a language $\mathcal{L}_o = \mathcal{L} \cup \{\mathit{failure}\}$ for describing the output of services. We use ϕ not only to denote elements of $\mathcal{L}$, but also of $\mathcal{L}_o$, but if the latter is meant, this will be indicated explicitly. Let N_{sn} with typical element sn be a set of service names such that $d \notin N_{sn}$.

The set of *service descriptions* $\mathcal{S}$ with typical element sd is then defined as follows:

$$\{\langle sn, \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff}\rangle \mid \mathsf{in} \subseteq \mathsf{Atom}, \mathsf{out} \subseteq \mathcal{L}_o, \mathit{failure} \in \mathsf{out}, \mathsf{prec} \in \mathcal{L} \text{ and } \mathsf{eff} \subseteq \mathcal{L}\}.$$

An example from the car repair scenario of a service description is the service that gives information on when it would be possible to make a garage appointment (with some particular garage). In this simple example, the input of this *garageAppInfo* service is $\{\text{possAppGarageMonday}, \ldots, \text{possAppGarageFriday}\}$, representing that it can accept information requests regarding whether it would be possible to have an appointment on Monday (*possAppGarageMonday*), Tuesday, etc. The output is the same as the input, with the addition of *failure* and with the addition of the negations of the atoms in the input, representing that it can provide information on possibilities for making an appointment. The idea is that on an input of, e.g., *possAppGarageMonday*, returns either this formula itself or the negation, depending on whether it is possible to make a garage appointment on Monday. Both the preconditions and effects of the service are empty (i.e., $\top$), expressing that the service does not make any changes in the world.

In order to provide some more guidance and intuition to the use of service descriptions, we refine the above definition such that two kinds of services can be described: *information providing* services and *world altering* services [15]. Intuitively, information providing services such as flight information providers

give information on something that is the case in the world, and world altering services such as flight booking services can make changes in the world.

In the context of this paper, we define information providing services as services that have no effect, and for each formula appearing in the output description, the negation of this formula should also be in the output description. The idea here is that the service should be able to provide information on each formula in the output description, meaning that it should be able to tell for each formula ϕ in the output description whether this formula holds in the world. The service should thus be able to return on an input ϕ, either ϕ or $\neg\phi$, depending on whether ϕ holds or does not hold, respectively, and therefore both ϕ and $\neg\phi$ should appear in the output description.

World altering services are here defined as services for which the output description is equal to the effects description. The idea is that a world altering service should be able to return the effect of its execution to its caller. As will be explained in more detail in Section 3.1, in order to make full use of goal-based orchestration, it is important that a service returns what it has done. The two kinds of services are defined formally below.

Definition 2 *(information providing and world altering services)* Let $\langle sn, \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff}\rangle$ be a service description. This service description is an information providing service iff $\mathsf{eff} \equiv \top$ and for each $\phi \neq \mathit{failure} \in \mathsf{out}$, there is a $\phi' \in \mathsf{out}$ such that $\phi' \equiv \neg\phi$. The service description is a world altering service iff $\mathsf{out} \setminus \{\mathit{failure}\} \equiv \mathsf{eff}$, i.e., if $\phi \in \mathsf{out} \setminus \{\mathit{failure}\}$, then $\exists \phi' \in \mathsf{eff} : \phi' \equiv \phi$, and vice versa.

A typical example of an information providing service is the *garageAppInfo* service that was mentioned above. The corresponding world altering service for actually making garage appointments, i.e., the *garageAppMaker* service, takes as input $\{\text{appGarageMonday}, \ldots, \text{appGarageFriday}\}$, representing that it accepts request for making appointments on Monday, Tuesday, etc. The output description is the same as the input description, with the addition of *failure*, and the effect description is the output description without *failure* (which is in this case equal to the input description), in accordance with the definition of a world altering service. The idea is that the service receives as input, e.g., *appGarageMonday*, expressing a request for making a garage appointment on Monday. Assuming that this is possible, the world is changed in such a way that the appointment is actually made, and the appointment itself is returned, to let the requester know that it has made the appointment.

In this paper, we assume that a service description describes either an information providing service, or a world altering service. In principle, one could imagine that there are services that are both information providing and world altering. However, we think that distinguishing these two kinds of services provides a conceptually clear guidance for how service descriptions can be used. Moreover, the two kinds of services fit well with the two kinds of goals that we consider here, i.e., test goals and achievement goals (see Section 2.2).

2.2 Orchestration Language

In this section, we describe the agent-based orchestration language. An agent has a *belief base* and a *goal base*. In [20], the belief base is generally a set of propositional formulas. Here, the belief base, typically denoted by σ, is a pair (σ_a, σ_b) of sets of propositional formulas that are mutually consistent. The idea is that σ_b forms the *b*ackground knowledge of the agent, which does not change during execution. The set σ_a forms the *a*ctual beliefs, which *are* modified during execution. The sets σ_a and σ_b correspond loosely with the A-Box and T-Box of description logics, respectively. The reason that we make this distinction is explained in Section 3.2.

The goals of the agent can be of two kinds, i.e., a goal is either an *achievement goal* or a *test goal*. An achievement goal $!\phi$, where ϕ is a propositional formula, represents that the agent wants to achieve a situation in which ϕ holds. A test goal $?\phi$ represents that the agent wants to know whether ϕ holds.[1] The idea is that test goals are to be fulfilled by information providing services, and achievement goals may be fulfilled by world altering services. Belief bases and goal bases are defined formally below.

Definition 3 *(belief base and goal base)* The set of belief bases Σ with typical element σ is defined as $\{(\sigma_a, \sigma_b) \mid \sigma_a, \sigma_b \subseteq \mathcal{L},\ \sigma_a \cup \sigma_b \not\models \bot\}$. The set of goals $\mathcal{L}_\mathsf{G}$ with typical element κ is defined as $\{?\phi, !\phi \mid \phi \in \mathcal{L}\}$. A goal base γ is a subset of $\mathcal{L}_\mathsf{G}$, i.e., $\gamma \subseteq \mathcal{L}_\mathsf{G}$.

In the car repair scenario, the initial goal base contains *!carRepaired*, representing that the agent has the goal of getting the car repaired. The part of the initial belief base with actual beliefs contains *repOnSpot*, representing that initially it is assumed that the car is repairable on the spot, and the background knowledge consists of $\{\text{appGarageMonday} \rightarrow \text{appGarage}, \ldots\}$, expressing that if the agent has a garage appointment on Monday etc., it has a garage appointment.

Following agent terminology, an expression in our orchestration language is called a *plan*. Broadly speaking, a plan consists of actions which change the belief base of the agent, of subgoals that are to be achieved by selecting a more concrete plan, and of service calls, the results of which may be passed along and which may be stored in the belief base.

A service call has the form $sn^r(\phi, \kappa)$, where sn is the name of the service that is to be called, κ represents the goal that is to be achieved through calling the service, and ϕ represents additional information that forms input to the service. The parameter r is called the revision parameter. This revision parameter can be either np or p, where np stands for non-persistent (meaning that the result of the service call is not stored), and p stands for persistent (meaning that the result is stored in the belief base). We thus provide the programmer with the possibility to specify what to do with the result of service calls. Typically, the results of world altering service calls will be stored in the belief base, together

[1] The syntax of representing achievement goals and test goals is taken from [18].

with results of information providing services that are likely to be needed at a later stage.

The service name sn of an annotated service call may be either from the set of service names N_{sn}, or it may be the reserved name d. If a name $sn \in N_{sn}$ is used in a service call, we say that this is a *service call by name*[2], i.e., the service with name sn should be called. Usage of the name d in a service call represents a *service call by discovery*, i.e., a service should be discovered that matches the goal with which the service is called.

Sequential composition of actions and service calls is done by means of the construct $b >x> \pi$, where b is an action, a subgoal, or a service call. The result returned from b is bound to the variable x, which may be used in the remaining plan π. This construct for sequential composition is inspired by a similar construct in the orchestration language Orc [7]. Note that the result of a service call can thus be used in the remaining plan, even though it is not stored in the belief base.

Definition 4 *(plan)* Assume that a set BasicAction with typical element a, and a set of variable names Var with typical element x are given. Let N_{sn}^{+} be defined as $N_{sn} \cup \{d\}$.[3] Let $sn \in N_{sn}^{+}$, $r \in \{np, p\}$, $\phi \in \mathcal{L}$ and $\kappa \in \mathcal{L}_{\mathsf{G}}$. Then the set of plans Plan with typical element π is defined as follows, where b stands for basic plan element.

$$b ::= a \mid \kappa \mid sn^{r}(\phi, \kappa)$$
$$\pi ::= b \mid b >x> \pi$$

As in Orc, a plan of the form $b \gg \pi$ is used to abbreviate a plan $b >x> \pi$ where x does not occur in π. This may be used in particular in case b is a basic action, as the execution of an action only modifies the belief base and does not return a result. Extending the syntax of plans with more involved constructs, such as constructs for programming parallelism, is left for future research.

An example of a plan in the car repair scenario is (where we abbreviate "Monday" with "M", etc., and if the information parameter of a service call is not shown, this should be interpreted as being $\top$):

$$d^{np}(?(\text{possAppGarageM} \vee \ldots \vee \text{possAppGarageF})) >poss>$$
$$\text{chooseApp}^{np}(poss, ?(\text{appGarageM} \vee \ldots \vee \text{appGarageF})) >app> d^{p}(!app). \quad (1)$$

This plan intuitively represents that a service should be discovered that provides information on when a garage appointment would be possible, e.g., on Monday and on Tuesday. In our example, the service to be discovered would be the *garageAppInfo* service, as described in Section 2.1. The idea is that one does not know beforehand where a car breakdown will occur, and therefore the orchestration expresses that a service for making garage appointments should

[2] Note that usage of the term *call by name* here is not related to the distinction between call by name and call by value in programming language research.

[3] We use sn as typical element of N_{sn} and of N_{sn}^{+}. It will generally be clear from the context which is meant, and otherwise it will be indicated explicitly.

be discovered. The result of the service is passed to a service that chooses an appointment from possible appointments.[4] The result of the *chooseApp* service, e.g., *appGarageMonday*, is passed to a service for making garage appointments, which needs to be discovered.[5] The intermediate results of the first two service calls are passed along and not stored anywhere, and the result of the service that actually makes the appointment *is* stored.

Plans are executed in order to achieve the agent's goals. The specification of which plan may be executed in order to achieve a certain goal is done by means of *plan selection rules* [20]. A plan selection rule $\kappa \mid \beta \Rightarrow \pi$ intuitively represents that if the agent has the goal κ and believes β to be the case, it may execute the plan π.

Definition 5 *(plan selection rules)* The set of plan selection rules $\mathcal{R}_{\mathsf{PS}}$ is defined as $\{\kappa \mid \beta \Rightarrow \pi \ : \ \kappa \in \mathcal{L}_{\mathsf{G}},\ \beta \in \mathcal{L},\ \pi \in \mathsf{Plan}\}$[6].

Plan selection rules can be applied to an agent's top-level goals in the goal base, but also to (the goals of) service calls in a plan that is currently executing (if the service call has not yielded a satisfactory result). Our example agent has two plan selection rules that specify how the goal of getting the car repaired can be reached. The first rule, which we omit here, specifies that a road assistance company can be called if the car is believed to be repairable on the spot. As we assume initially that the agent beliefs the car to be repairable on the spot, this rule is applied first. If it turns our that the car is after all not repairable on the spot, then the second rule can be applied:

$$!\text{carRepaired} \mid \neg\text{repOnSpot} \Rightarrow \\ !\text{appGarage} \gg d^p(!\text{appTowTruck}) \gg \text{monitor}^p(?\text{carRepaired}). \quad (2)$$

This rule says that if the agent has the goal of getting his car repaired and he believes it is not possible to repair the car on the spot, it should make a garage appointment and a tow truck appointment, and then it should check whether the car is actually repaired. In order to achieve the goal of having a garage appointment, the agent can apply the plan selection rule $!\text{appGarage} \mid \top \Rightarrow \pi$, where π is the plan from (1).

The mechanism of applying plan selection rules is formalized using the notion of a stack. This stack can be compared with the stack resulting from procedure calls in procedural programming, or method calls in object-oriented programming, and a similar mechanism was also used in [21]. During execution of the

[4] We assume that the *chooseApp* service returns just one possible appointment from the possible appointments.

[5] Presumably, this should be a service of the same garage as the discovered service for providing information on possible appointments. Extending the orchestration language with a linguistic mechanism for expressing this (using service variables), is left for future research.

[6] We use the notation $\{\ldots \ : \ \ldots\}$ instead of $\{\ldots \mid \ldots\}$ to define sets, to prevent confusing usage of the symbol $\mid$ in this definition.

agent, a single stack is built. Each element of the stack represents, broadly speaking, the application of plan selection rules to a particular (sub)goal. To be more specific, each element of the stack is of the form $(\pi, \kappa, \mathsf{PS})$, where κ is the (sub)goal to which the plan selection rule has been applied, π is the plan currently being executed in order to achieve κ, and PS is the set of plan selection rules that have not yet been tried in order to achieve κ.

Definition 6 *(stack)* The set of stacks Stack with typical element St to denote arbitrary stacks, and st to denote single elements of a stack, is defined as follows, where $\pi \in \mathsf{Plan}$, $\kappa \in \mathcal{L}_{\mathsf{G}}$, and $\mathsf{PS} \subseteq \mathcal{R}_{\mathsf{PS}}$.

$$\begin{aligned} \mathrm{st} &::= (\pi, \kappa, \mathsf{PS}) \\ \mathrm{St} &::= \mathrm{st} \mid \mathrm{st}.\mathrm{St} \end{aligned}$$

E is used to denote the empty stack (or the empty stack element), and E.St is identified with St.

We are now in a position to give a definition of an agent. An agent has a belief base, a goal base, a stack, a set of plan selection rules, and a belief update function. The belief update function is introduced as usual [20] for technical convenience, and is used to define the semantics of action execution. We introduce a constraint on agents that expresses that any goal that is used should be consistent with the background knowledge of the belief base. Note that according to this definition, in particular goals $?\top$, $?\bot$, and $!\bot$ are not allowed.

Definition 7 *(agent)* An agent $\mathcal{A}$ is a tuple $\langle \sigma, \gamma, \mathrm{St}, \mathsf{PS}, \mathcal{T} \rangle$ where $\sigma \in \Sigma$ is the belief base, $\gamma \subseteq \mathcal{L}_{\mathsf{G}}$ is the goal base, $\mathrm{St} \in \mathsf{Stack}$ is the current plan stack of the agent, $\mathsf{PS} \subseteq \mathcal{R}_{\mathsf{PS}}$ is a finite set of plan selection rules, and $\mathcal{T}$ is a partial function of type $(\mathsf{BasicAction} \times \Sigma) \to \Sigma$ and specifies the belief update resulting from the execution of basic actions.

Further, agents should satisfy the following constraint. Let $\sigma = (\sigma_a, \sigma_b)$. It should then be the case that for any goal $\cdot\phi$, where "$\cdot$" stands for ? or !, occurring in a goal in γ or in a service call in one of the plans of PS, $\sigma_b \not\models \neg\phi$. For any goal $?\phi$ it should also be the case that $\sigma_b \not\models \phi$. Initially, the plan stack of the agent is empty.

3 Semantics

In this section, we define the semantics of the orchestration language. The definition is split into two parts. First, we define the semantics of service calls (Section 3.1), and then we define the semantics of the orchestration language as a whole, making use of the semantics of service calls (Section 3.2).

3.1 Service Calls

In defining the semantics of service calls, we have to define two things. First, we need a definition of when a service matches a service call. Second, we need to specify what a service may return, if it is called.

Although it is not the purpose of this paper to define advanced matching algorithms, we do provide one possible definition of matching. The reason for this is that in this goal-oriented context, the definition of a match depends on the (kind of) goal with which a service is called. We think it is important to identify how the use of goals influences the definition of a match, in order to be able to identify at a later stage which existing matching algorithms can be used in a goal-oriented setting, or how they might have to be adapted.

When matching a service to a goal, the idea is that a test goal is matched to an information providing service, and an achievement goal is matched to a world altering service. That is, for a test goal it is important to match the goal against the *output* description, and for an achievement goal the goal is matched to the *effect* description. The matching definition below corresponds loosely with what is called plug-in matching in [11].

This approach to matching is based on the idea that a service should provide *at least* the information that is asked for, or do *at least* what is desired. Formally, a service description sd matches a service call $sn(\phi, ?\phi')$ if ϕ and ϕ' do not contain atoms that are not in the inputs description of sd. Moreover, what the agent believes to be the case should not contradict with the preconditions description of sd. The idea here is that the agent may not always be able to check whether the precondition of a service holds, but it should at least not have explicit information that the precondition does *not* hold. Finally, there should be a consistent subset of the outputs description (for test goals) or effects description (for achievement goals) from which the goal of the service call follows. Intuitively, this represents that the service is able to provide at least the information that is asked for, or is able to do at least what is desired, respectively.

Definition 8 *(matching a service to a goal)* In the following, we define $\sigma \models \phi$ where $\sigma = (\sigma_a, \sigma_b)$ as $\sigma_a \cup \sigma_b \models \phi$, where $\models$ is the standard entailment relation of propositional logic. Assume a function $atoms : \mathcal{L} \rightarrow \wp(\mathsf{Atom})$ that takes a formula from $\mathcal{L}$ and yields the set of atoms that occur in the formula. Let $sd = \langle sn', \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff} \rangle$ be a service description. Then the matching predicate $\text{match}(sn(\phi, \kappa), \sigma, sd)$, which takes a service call $sn(\phi, \kappa)$, a belief base σ, and a service description sd, is defined as follows if $sn \neq d$.

$$
\begin{aligned}
\text{match}(sn(\phi, ?\phi'), \sigma, sd) \Leftrightarrow\ & sd \text{ is information providing and } sn = sn' \text{ and} \\
& \text{atoms}(\phi), \text{atoms}(\phi') \subseteq \mathsf{in} \text{ and } \sigma \not\models \neg\mathsf{prec} \text{ and} \\
& \exists \mathsf{out}' \subseteq \mathsf{out} : \mathsf{out}' \not\models \bot \text{ and } \mathsf{out}' \models \phi' \\
\text{match}(sn(\phi, !\phi'), \sigma, sd) \Leftrightarrow\ & sd \text{ is world altering and } sn = sn' \text{ and} \\
& \text{atoms}(\phi), \text{atoms}(\phi') \subseteq \mathsf{in} \text{ and } \sigma \not\models \neg\mathsf{prec} \text{ and} \\
& \exists \mathsf{eff}' \subseteq \mathsf{eff} : \mathsf{eff}' \not\models \bot \text{ and } \mathsf{eff}' \models \phi'
\end{aligned}
$$

If $sn = d$, then the same definition applies, but the requirement that $sn = sn'$ is dropped.

Note that one needs to define a match by specifying that a goal is a logical consequence of a *consistent subset*, rather than as a logical consequence of the outputs

or effects description as a whole, as these descriptions may be inconsistent. This definition is inspired by the so-called consistent subset semantics as proposed in [20, Chapter 4] for defining semantics of goals in case these goals may be inconsistent. Also, note that a service call by name has the additional restriction that the name of the service call should match the name of the service description, meaning that a service call by name is more restrictive than a service call by discovery. Further, if a service is able to provide information on ϕ, it can also provide information on $\neg\phi$, as expressed by the following proposition.

Proposition 1

$$\exists \mathsf{out}' \subseteq \mathsf{out} : \mathsf{out}' \not\models \bot \text{ and } \mathsf{out}' \models \phi' \Rightarrow \exists \mathsf{out}' \subseteq \mathsf{out} : \mathsf{out}' \not\models \bot \text{ and } \mathsf{out}' \models \neg\phi'$$

The semantics of service execution is defined using a predicate $\mathrm{ret}(sd, \phi)$, which specifies that ϕ may be returned by the service corresponding with service description sd. The idea is that what is returned by a service should be compatible with its output description. If the predicate $\mathrm{ret}(sd, \phi)$ is true, this represents that ϕ may be returned by a service that has service description sd. It is important to have a specification of what may be returned by the service, as we will need it in the semantics of the orchestration language to determine whether the goal of a service call is reached.

Definition 9 *(semantics of service execution)* Let $sd = \langle sn, \mathsf{in}, \mathsf{out}, \mathsf{prec}, \mathsf{eff} \rangle$ be a service description. The predicate *ret* is then defined as follows.

$$\begin{aligned} \mathrm{ret}(sd, \phi) \Leftrightarrow\ & \phi \equiv \mathit{failure} \text{ or} \\ & \exists \mathsf{out}' \subseteq \mathsf{out} \setminus \{\mathit{failure}\} : (\mathsf{out}' \not\models \bot \text{ and } \textstyle\bigwedge_{\phi_o \in \mathsf{out}'} \phi_o \equiv \phi) \end{aligned}$$

3.2 Orchestration Language

The semantics of the orchestration language is defined by means of a transition system [17]. A transition system for a programming language consists of a set of axioms and transition rules for deriving transitions for this language. A transition is a transformation of one configuration (or agent in this case) into another and it corresponds to a single computation step.

For reasons of presentation, we will in the following omit the set of plan selection rules PS and the function $\mathcal{T}$ from the specification of an agent, as these do not change during computation. In the transition rules below, we will refer to the set of plan selection rules of the agent with $\mathsf{PS}_{\mathcal{A}}$. Further, we assume the agent has access to a finite set of service descriptions $S_{\mathcal{A}}$. Finally, we sometimes omit the revision parameter r from service calls, if this parameter is not relevant there.

The first transition rule specifies how a transition for a composed stack can be derived, given a transition for a single stack element. It specifies that only the top element of a stack can be transformed or executed.[7]

[7] We omit a rule that specifies that the two topmost stack elements may be modified at the same time.

Definition 10 *(stack execution)* Let $st \neq E$.

$$\frac{\langle \sigma, \gamma, \mathrm{st} \rangle \rightarrow \langle \sigma', \gamma', \mathrm{st}' \rangle}{\langle \sigma, \gamma, \mathrm{st}.\mathrm{St} \rangle \rightarrow \langle \sigma', \gamma', \mathrm{st}'.\mathrm{St} \rangle}$$

Before we continue with the definition of transition rules, we need to define when a goal is achieved, and when a plan selection rule can be applied to a goal. The definition of when a goal is achieved differs for test goals and achievement goals. A test goal is evaluated against the result of a service call, i.e., the belief base is not taken into account. The idea is that a test goal is used if the agent wants to obtain or to check a piece of information, regardless of whether it already believes something about this piece of information. The achievement of an achievement goal, on the other hand, is determined on the basis of the belief base, together with the result of a service call.

We specify what it means to take a belief base together with a result of service execution, using a belief revision function. For examples on how such a function is defined, see, e.g., [1] which shows how a belief revision mechanism can be incorporated into the agent programming language AgentSpeak(L). Below, we only specify the constraints that such a belief revision function should satisfy. That is, if a belief base (σ_a, σ_b) is updated with a result x, only σ_a should be updated. The function is not defined if x is inconsistent with the background knowledge.

Definition 11 *(belief revision function)* In the following, we assume a partial belief revision function $brev : (\wp(\mathcal{L}) \times \wp(\mathcal{L})) \rightarrow (\mathcal{L} \rightarrow \wp(\mathcal{L}))$. The function $brev((\sigma_a, \sigma_b), x)$ is defined iff $\sigma_b \not\models \neg x$, and if it is defined, it should satisfy the following constraints on behavior: $brev((\sigma_a, \sigma_b), x) = (\sigma'_a, \sigma_b)$ where $\sigma'_a \models x$ and $\sigma'_a \cup \sigma_b \not\models \bot$; if $\sigma \models x$, then $brev(\sigma, x) = \sigma$.

In agent programming frameworks, the achievement of achievement goals is determined on the belief base, as this is the only component representing the current situation. In this context where we have service calls that return results, not all results are stored in the belief base. Therefore, we also take into account the result of the relevant service call when evaluating whether a goal is achieved. In fact, the evaluation of a test goal should be performed *only* on the result of a service call, as we disregard whether the agent already believes something about the test goal.

This is reflected in the semantics of goal achievement as defined formally below, in which we specify when a predicate $\mathrm{ach}(\kappa, \sigma, x)$ holds, representing that the goal κ is achieved with respect to the result of a service call x and belief base σ. To be more specific, a test goal $?\phi$ holds if the result of the service call expresses that either ϕ or $\neg\phi$ hold. An achievement goal $!\phi$ holds if ϕ follows from the belief base that would result from updating the old belief base with the service result.

Definition 12 *(semantics of goal achievement)* The semantics of goal achievement is defined as a predicate $\mathrm{ach}(\kappa, \sigma, x)$ that takes a goal κ, a belief base σ,

and a propositional formula $x \in \mathcal{L}$ that represents the result against which κ should be checked.

$$\text{ach}(?\phi, \sigma, x) \Leftrightarrow \mathit{brev}(\sigma, x) = \sigma' \text{ and } (x \models \phi \text{ or } x \models \neg\phi) \text{ and } x \neq \mathit{failure}$$
$$\text{ach}(!\phi, \sigma, x) \Leftrightarrow \mathit{brev}(\sigma, x) = \sigma' \text{ and } \sigma' \models \phi \text{ and } x \neq \mathit{failure}$$

Although test goal achievement is defined in principle only on the result of the service call x, we use the belief base to check that x is consistent with the background knowledge. We thus have by definition that a goal cannot be achieved with respect to a result of a service call, if this result is inconsistent with the background knowledge (in that case *brev* would be undefined for this result).

A plan selection rule ρ of the form $\kappa' \mid \beta \Rightarrow \pi$ is applicable to a goal κ given a belief base σ, if κ' matches κ, β holds according to σ, and κ' is not achieved. The kind of matching we use is one that could be called *partial matching*. Here, a rule with head κ' is applicable to a goal κ if κ' "follows from" κ. That is, a rule with, e.g., head $!p$, could match a goal $!(p \wedge q)$. This kind of semantics is generally used for these rules in agent programming [20], as the goal decomposition for which it allows has practical advantages.

Definition 13 *(applicability of plan selection rule)* We define a predicate $\text{applicable}(\rho, \kappa, \sigma)$ that takes a plan selection rule ρ, a goal κ, and a belief base σ as follows, where "$\cdot$" stands for ? or !.

$$\text{applicable}(\cdot\, \phi' \mid \beta \Rightarrow \pi, \cdot\, \phi, \sigma) \Leftrightarrow \phi \models \phi' \text{ and } \sigma \models \beta \text{ and } \neg\text{ach}(\cdot\, \phi', \sigma, \top)$$

In order to start execution and create a first stack element, the agent applies a plan selection rule to a goal in the goal base. However, we leave out the corresponding transition rule for reasons of space.

When an agent encounters a service call construct during execution of a plan, it tries to call matching services until there are no more matching services, or the goal of the service call is reached. In order to keep track of which services have been called, we annotate the service call construct (initially) with the set of services that are available. For an achievement goal, the agent only tries to call services if the goal is not already reached. For a test goal, the agent always tries to call a service, no matter whether it already believes the test goal to hold. The service call is then used to check whether the information of the agent is correct. We omit the rule for test goals for reasons of space. From this set of services, a matching service is selected non-deterministically. The result of the execution of the selected service is also stored in the annotation with the service call construct, as this is used in other transition rules (Definitions 16 and 22) to check whether the goal of the service call is reached. For reasons of presentation, we omit here and in the sequel rules for dealing with service calls and actions that form the last element of a plan.

Definition 14 *(calling services)*

$$\frac{\neg\text{ach}(!\phi', \sigma, \top)}{\langle \sigma, \gamma, (\mathit{sn}(\phi, !\phi') >x> \pi, \kappa, \mathsf{PS})\rangle \rightarrow \langle \sigma, \gamma, (\mathit{sn}(\phi, !\phi')[S_{\mathcal{A}}, \top] >x> \pi, \kappa, \mathsf{PS})\rangle}$$

$$\frac{\neg\text{ach}(\kappa,\sigma,x_o) \quad sd \in S \quad \text{match}(sn(\phi,\kappa),\sigma,sd) \quad \text{ret}(sd,x_n)}{\langle\sigma,\gamma,(sn(\phi,\kappa)[S,x_o] >x> \pi,\kappa',\mathsf{PS})\rangle \rightarrow \langle\sigma,\gamma,(sn(\phi,\kappa)[S\setminus\{sd\},x_n] >x> \pi,\kappa',\mathsf{PS})\rangle}$$

Note that the second transition rule selects a matching service if the goal of the service call is not reached. This provides a way of dealing with failure of services, as another service is tried if a former service call did not have the desired result. This is easily specified in our semantics, as we use an explicit representation of goals.

In the following, we use a revision function that takes a revision parameter r, a belief base σ and a result of a service call x, and updates σ with x, depending on r.

Definition 15 *(revision function)* The revision function rev is defined as follows: $rev(np,\sigma,x) = \sigma$ and $rev(p,\sigma,x) = brev(\sigma,x)$.

The next transition rule specifies what happens if the goal of a service call is reached after calling a service, and the goal of the stack element is not yet reached. If this is the case, the belief base is updated according to the revision parameter, and all occurrences of the parameter x of the sequential composition in the rest of the plan π are replaced by the result of the service call x', i.e., the result of the service call is passed along. Moreover, all goals in the goal base that are believed to be reached after the revision resulting from the service call are removed.

Definition 16 *(goal of service call achieved after service execution)*

$$\frac{\neg\text{ach}(\kappa',\sigma,x') \quad \text{ach}(\kappa,\sigma,x') \quad rev(r,\sigma,x') = \sigma' \quad \gamma' = \gamma\setminus\{\kappa \mid \text{ach}(\kappa,\sigma,x')\}}{\langle\sigma,\gamma,(sn^r(\phi,\kappa)[S,x'] >x> \pi,\kappa',\mathsf{PS})\rangle \rightarrow \langle\sigma',\gamma',([x'/x]\pi,\kappa',\mathsf{PS})\rangle}$$

It might also be possible that a subgoal or the goal of a service call is already reached before a service is called (only in case of an achievement goal). The question is, what should be passed along in this case. One possibility would be to pass along the goal itself. However, this yields unintuitive results in case the background knowledge is used to derive the goal. In the car repair scenario, the subgoal !appGarage can be achieved by applying a plan selection rule !appGarage $\mid \top \Rightarrow \pi$, where π is the plan from (1). The goal !appGarage can be achieved through the service call $d^p(!app)$, which is matched to the *garageAppMaker* service. This service might return, e.g., *appGarageMonday*. Taking the background knowledge of the agent, we can then derive !appGarage, making this goal achieved.

The idea now is, that we want to pass along *appGarageMonday*, rather than *appGarage*, as the first is the concrete realization of the second higher level goal. This is achieved by passing along only the σ_a part of the belief base that "contributes" to the goal being reached. That is, background knowledge is not passed along. The part of σ_a that should be passed along, is what we call the *base*, and this is defined formally below.

Definition 17 *(base)* The predicate base(σ, ϕ, x) takes a belief base σ, a formula ϕ where $\sigma \models \phi$, and a formula x representing the base of ϕ in σ. Let $\sigma = (\sigma_a, \sigma_b)$, and let $\sigma'_a \subseteq \sigma_a$ such that $(\sigma'_a, \sigma_b) \models \phi$ and for any σ''_a such that $\sigma''_a \subset \sigma'_a$, we have $(\sigma''_a, \sigma_b) \not\models \phi$. The predicate is then defined as follows: base$((\sigma_a, \sigma_b), \phi, x) \Leftrightarrow x = \bigwedge_{\phi' \in \sigma'_a} \phi'$.

The idea of the base is thus to extract the concrete realization of a goal, rather than the higher level abstract goal. In description logic, a concrete realization might correspond with an instance, where the higher level goal could be represented using a concept. The transition rule below specifies the case where the goal is reached before a service is called. Is similar rule is used for subgoals, but we omit it for reasons of space.

Definition 18 *(goal of service call achieved before services are called)*

$$\frac{\neg\text{ach}(\kappa', \sigma, \top) \quad \text{ach}(!\phi', \sigma, \top) \quad \text{base}(\sigma, \phi', x')}{\langle \sigma, \gamma, (sn(\phi, !\phi') >x> \pi, \kappa', \mathsf{PS})\rangle \rightarrow \langle \sigma, \gamma, ([x'/x]\pi, \kappa', \mathsf{PS})\rangle}$$

If a subgoal is not achieved, a plan selection rule may be applied to the subgoal. The application of a plan selection rule to a (sub)goal is the only way in which a new stack element can be created.

Definition 19 *(apply rule to create stack element)* Below, $\mathsf{PS}' = \mathsf{PS}_{\mathcal{A}} \setminus \{\kappa' \mid \beta \Rightarrow \pi\}$.

$$\frac{\neg\text{ach}(\kappa'', \sigma, \top) \quad \kappa' \mid \beta \Rightarrow \pi \in \mathsf{PS}_{\mathcal{A}} \quad \text{applicable}(\kappa' \mid \beta \Rightarrow \pi, \kappa, \sigma)}{\langle \sigma, \gamma, (\kappa >x> \pi', \kappa'', \mathsf{PS})\rangle \rightarrow \langle \sigma, \gamma, (\pi, \kappa, \mathsf{PS}').(\kappa >x> \pi', \kappa'', \mathsf{PS})\rangle}$$

If the plan of a stack element is empty, a plan selection rule may be applied in order to select another plan to try to reach the goal of the stack element. Note that if a plan selection rule is applied to the goal of a stack element, this does not lead to the creation of a new stack element.

Definition 20 *(apply rule within stack element)* Below, $\mathsf{PS}' = \mathsf{PS} \setminus \{\kappa' \mid \beta \Rightarrow \pi\}$.

$$\frac{\kappa' \mid \beta \Rightarrow \pi \in \mathsf{PS} \quad \text{applicable}(\kappa' \mid \beta \Rightarrow \pi, \kappa, \sigma)}{\langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS})\rangle \rightarrow \langle \sigma, \gamma, (\pi, \kappa, \mathsf{PS}')\rangle}$$

Popping a stack element is done in two cases: the goal of the stack element is reached, or the goal is not reached and there are no more applicable rules. The goal of a stack element may be reached after a service call, or after action execution. In the first case, the result of the relevant service call is passed to the stack element just below the top element. In the second case, a result to be passed is obtained from the belief base using the *base* predicate (Definition 17).

Definition 21 *(popping a stack element: goal of stack element reached or unreachable)*

$$\frac{\text{ach}(\kappa_1, \sigma, x) \quad rev(r, \sigma, x) = \sigma' \quad \gamma' = \gamma \setminus \{\kappa \mid \text{ach}(\kappa, \sigma', \top)\}}{\langle \sigma, \gamma, (sn^r_1(\phi_1, \kappa_3)[S, x] >x_1> \pi_1, \kappa_1, \mathsf{PS}_1).(\kappa_1 >x_2> \pi_2, \kappa_2, \mathsf{PS}_2)\rangle \rightarrow \langle \sigma', \gamma', ([x/x_2]\pi_2, \kappa_2, \mathsf{PS}_2)\rangle}$$

$$\frac{\mathcal{T}(\sigma, a) = \sigma' \quad \text{ach}(!\phi', \sigma', \top) \quad \text{base}(\sigma', !\phi', x') \quad \gamma' = \gamma \setminus \{\kappa \mid \text{ach}(\kappa, \sigma', \top)\}}{\langle \sigma, \gamma, (a \gg \pi', !\phi', \mathsf{PS}').(!\phi' >x> \pi, \kappa, \mathsf{PS})\rangle \rightarrow \quad \langle \sigma', \gamma', ([x'/x]\pi, \kappa, \mathsf{PS})\rangle}$$

$$\frac{\neg\text{ach}(\kappa, \sigma, \top) \quad \neg\exists \rho \in \mathsf{PS} : \text{applicable}(\rho, \kappa, \sigma)}{\langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS}).(\kappa >x> \pi, \kappa', \mathsf{PS}')\rangle \rightarrow \langle \sigma, \gamma, (\epsilon, \kappa', \mathsf{PS}')\rangle}$$

Note that if the plan of a stack element is empty, the goal of the stack element is not reached. The reason is that the stack element would have been popped before the plan got empty, if the goal of stack element would have been reached after a service call or an action execution.

If an action of a plan cannot be executed, or there is no applicable rule for a subgoal of a plan, or the goal of a service call has not been reached and there are no more services that match, then the plan fails. If this happens, the plan is dropped. Consecutively, another plan for achieving the goal of the stack element may be tried, providing for flexibility in handling failure (Definition 20). Below, we only show the transition rule for the case where an action is not executable.

Definition 22 *(plan failure)*

$$\frac{\mathcal{T}(\sigma, a) \text{ is undefined}}{\langle \sigma, \gamma, (a \gg \pi, \kappa, \mathsf{PS})\rangle \rightarrow \langle \sigma, \gamma, (\epsilon, \kappa, \mathsf{PS})\rangle}$$

Popping a stack element if the goal is not reached and there are no more applicable rules, prevents the agent from getting "stuck" or from looping inside a stack element, while not reaching the (sub)goal of the stack element. If stack elements are popped if the goal cannot be reached, the agent can try another plan in the stack element that then becomes the new top element of the stack. This mechanism functions recursively, meaning that if the agent has tried everything without success, it will have an empty stack element again. However, the top-level goal that the agent tried to reach is still in the belief base, at it was probably not reached. The agent can then try another goal, or wait for the circumstances to change for the better, and give it another try later on. If the agent terminates, then either the agent has an empty stack and the goal base is empty, or the agent has an empty stack and there are no applicable rules to the goals in the goal base. These considerations of progress and terminations are formalized in the proposition below.

Proposition 2 *(progress and termination)* Let $\mathcal{A} = \langle \sigma, \gamma, St\rangle$ be an agent. Then, if $St \neq E$, there is an $\mathcal{A}'$ such that $\mathcal{A} \rightarrow \mathcal{A}'$. Further, on any computation $\mathcal{A}, \mathcal{A}_1, \ldots$ there is an $\mathcal{A}'$ of the form $\langle \sigma', \gamma', E\rangle$. Finally, if there is no $\mathcal{A}'$ such that $\mathcal{A} \rightarrow \mathcal{A}'$, then either $\mathcal{A}$ is of the form $\langle \sigma, \emptyset, E\rangle$, or $\mathcal{A}$ is of the form $\langle \sigma, \gamma, E\rangle$ and there is no ρ and $\kappa \in \gamma$ such that applicable$(\rho, \kappa, \sigma, \top)$.

4 Conclusion

In this paper, we have proposed to use goal-oriented techniques from the field of cognitive agent programming for service orchestration. The advantage of using an explicit representation of goals is the flexibility in handling failure that

goals provide. To be more specific, goals provide for flexibility in at least four ways. First, goals can be used to do semantic matchmaking, yielding flexibility in selection of services, as one does not necessarily have to define at design time which particular service should be called. Second, the explicit use of goals makes it easy to check whether a service call was successful, making it easy to build into the semantics a mechanism for trying other matching services if one service fails. Third, the plan selection rules can be used in a natural way to specify an alternative plan if calling a service directly fails. Finally, the possibility to specify *multiple* plans for achieving a goal in combination with the mechanism for detecting whether a goal is achieved, can make the orchestration more flexible in handling failure, and also more responsive to the actual situation as plan selection rules are conditionalized on beliefs. We have made these ideas concrete by formally defining a goal-based orchestration language that makes use of semantic matchmaking.

We see several directions for future research. One of the main issues that needs to be dealt with is the fact that the language is based on propositional logic. Important candidates to replace propositional logic are description logics, given the fact that we incorporate semantic matchmaking into our framework, or first-order logics. We plan to study in particular whether the WSML [8] language can be integrated into our work, as this is a language that has both first-order logic and description logic variants. Further, WSML seems to be particularly suited for our work, as it provides a formal syntax and semantics for WSMO, which is based on goals. Moreover, we plan to extend the language of plans to include more sophisticated constructs such as a construct for parallel composition. Further, we consider to investigate more expressive kinds of service communication in which interaction protocols are used for communication. Furthermore, we want to investigate how work on soft constraints [3] can be used to obtain a more expressive language for representing goals and for defining goal achievement. Moreover, we want to analyze how exactly our semantics for the result passing sequential composition construct differs from the Orc semantics. Finally, we aim to analyze formally how our goal-based mechanism for handling failure is related to more conventional approaches to failure handling, such as used in WS-BPEL.

Concluding, we believe that the framework as laid out in this paper can provide a foundation for several interesting directions of future research, and we hope it contributes to the further investigation of combining agent-oriented and service-oriented approaches.

References

1. N. Alechina, R. Bordini, J. Hübner, M. Jago, and B. Logan. Automating belief revision for agentspeak. In *Proc. of DALT'06*, 2006.
2. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Interaction protocols and capabilities: A preliminary report. In *Proc. of PPSWR'06*, pages 63–77, 2006.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of ACM*, 44:201–236, 1997.

4. L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. CooWS: Adaptive BDI agents meet service-oriented programming. In *Proc. of WWW/Internet'05*, volume 2, pages 205–209. IADIS Press, 2005.
5. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal representation for BDI agent systems. In *Proc. of ProMAS'04*, volume 3346 of *LNAI*, pages 44–65. Springer, Berlin, 2005.
6. R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *Proc. of POPL'05*, pages 209–220, 2005.
7. W. R. Cook and J. Misra. Computation orchestration: A basis for wide-area computing, 2007. To appear in the Journal on Software and System Modeling.
8. J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, and D. Fensel. The web service modeling language WSML. WSML deliverable d16.1v0.2, 2005. `http://www.wsmo.org/TR/d16/d16.1/v0.2/`.
9. I. Dickinson and M. Wooldridge. Agents are not (just) web services: considering BDI agents and web services. In *Proc. of SOCABE'05*, 2005.
10. M. Juric, P. Sarang, and B. Mathew. *Business Process Execution Language for Web Services 2nd Edition*. Packt Publishing, 2006.
11. L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proc. of WWW'03*, pages 331–339. ACM Press, 2003.
12. D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing semantics to web services: The OWL-S approach. In *Proc. of SWSWPC 2004*, volume 3387 of *LNCS*, pages 26–42, 2005. Springer, Berlin.
13. V. Mascardi and G. Casella. Intelligent agents that reason about web services: a logic programming approach. In *Proc. of ALPSWS'06*, pages 55–70, 2006.
14. M. Mazzara and R. Lucchi. A framework for generic error handling in business processes. *Electr. Notes Theor. Comput. Sci.*, 105:133–145, 2004.
15. S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
16. M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic matching of web services capabilities. In *Proc. of ISWC'02*, volume 2342 of *LNCS*, pages 333–347. Springer-Verlag, 2002.
17. G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
18. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag, 1996.
19. D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1:77–106, 2005.
20. M. B. van Riemsdijk. *Cognitive Agent Programming: A Semantic Approach*. PhD thesis, 2006.
21. M. B. van Riemsdijk, M. Dastani, J.-J. Ch. Meyer, and F. S. de Boer. Goal-oriented modularity in agent programming. In *Proc. of AAMAS'06*, pages 1271–1278, Hakodate, 2006.
22. M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems. In *Proc. of KR'02*, 2002.
23. M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, A. Knapp, N. Koch, and A. Schroeder. Semantic-based development of service-oriented systems. In *Proc. of FORTE'06*, volume 4229 of *LNCS*, pages 24–45. Springer-Verlag, 2006.