Making Web Services Tradable

A Policy-based Approach for Specifying Preferences on Web Service Properties

Sudhir Agarwal\textsuperscript{a}, Steffen Lamparter\textsuperscript{a,\ast}, Rudi Studer\textsuperscript{a}

\textsuperscript{a}Institute of Applied Informatics and Formal Description Methods (AIFB), University of Karlsruhe (TH), Germany

Abstract

In service-oriented architectures, applications are developed by integrated services which are often provided by different organizations. Since a service might be offered under different conditions by various different organizations, sophisticated service selection and negotiation algorithms are required. Policies capture the conditions under which services are offered or requested and thereby constrain the negotiation space. However, current policy languages are ill-suited to realize beneficial trade-offs within a negotiation, since they support only boolean decisions and cannot capture all relevant service information.

Therefore, we present a novel policy language in this work that provides two main contributions: (i) We enable the specification of constraints on functional as well as non-functional properties of Web services. The functional properties include data objects and the behavior, whereas the non-functional properties include QoS attributes. (ii) Given such constraints, we show how the concept of utility function policies can be used to define cardinal preferences over functional as well as non-functional properties. This is required to rank Web service offers, define their prices, and consequently to realize automated negotiations between service providers and requesters.

Key words: Web Service Policies, Preferences for Functional/Non-Functional Properties, Automated Negotiations

\ast Corresponding author. Tel: +49 721 608 7946

Email addresses: agarwal@aifb.uni-karlsruhe.de (Sudhir Agarwal), lamparter@aifb.uni-karlsruhe.de (Steffen Lamparter), studer@aifb.uni-karlsruhe.de (Rudi Studer).

Preprint submitted to Elsevier 23 November 2007
1 Introduction

Web services are self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces. They allow flexible and dynamic software integration that is often referred to as the "Find-Bind-Execute"-paradigm. Moreover, by using standard Internet technology, Web services facilitate cross-organizational transactions and thus outsourcing of software functionality to external service providers. When moving from distributed systems operating within one company to systems that involve different, independent companies, the "Find-Bind-Execute"-schema describes nothing else than a B2B procurement process, where services such as information delivery or execution of calculations are purchased. Thus, service-oriented computing requires an infrastructure that provides a mechanism for coordination between service requesters and providers. This coordination mechanism has to provide a platform where potential business partners can be discovered, prices can be ascertained, and contracts can be closed. A market, where prices are determined by the interplay between supply and demand, can be regarded as a coordination mechanism that efficiently provides these functionalities [19].

In each phase of the market different kind of information is required. Functional properties are those attributes that are mandatory to be able to invoke a service and to integrate the results, e.g. the input/output or behaviour of a service. All discovered services fulfill the desired goal, but may differ in their non-functional properties. These are properties that are not required to invoke the service nor to integrate the results, but they are the decisive factors for service selection and price determination. For example, price, payment method, security as well as trust properties, and most notably quality of service properties. Since for each property several different alternatives can be adopted, sophisticated service selection and negotiation algorithms are required that depend on the preferences of service providers and requesters. In this entire process one has to make sure that a companies' policies including business objectives, regulative norms, such as Sarbanes-Oxley\(^1\), or IT-Governance standards (e.g. ISO 20000\(^2\)) are met, while other service properties are chosen in an optimal way (e.g. such that costs or the execution time are minimal).

In order to support scenarios, in which Web services consumers search and bind Web services dynamically without much human intervention from millions of Web services offered in the market, it is necessary that tasks like service matching, selection and negotiation are performed automatically. However, today's

---


Web service description and policy languages do not support the automation of these tasks since they either lack expressivity or formal semantics. For example, often policy languages are restricted to constraints on static properties and cannot express constraints on the behavior of Web services. In addition, to support negotiations of service properties a policy language has to support expressing detailed preference information that captures trade-offs between the different possible Web service configurations. Therefore, information beyond hard constraints is required. In order to enable interoperability and support automation of tasks like service matchmaking, negotiation and selection, it is not enough to have an expressive policy language, but the language should also have formal semantics, such that automatic reasoning procedures for policy evaluation can be developed and one can make sure that different agents interpret language constructs in the same way.

In this paper we present a formal policy language that enables the specification of cardinal preferences on arbitrary functional as well as non-functional properties of a Web service. We realize this by developing a policy ontology that supports the concept of utility function policies [23] and thereby captures detailed preferences. By augmenting the ontology with a temporal logic, preferences on the behaviour of a Web service can be specified, yielding an expressive policy language that allows specification of soft constraints on functional properties including behavior as well non-functional properties. By calculating the utility of a given Web service description according to the requester’s policies that include detailed preference information, automatic support for the matchmaking and negotiation can be enabled.

The paper is organized as follows. In Section 2, we first elaborate on the requirements a policy language for Web services should meet and then discuss in Section 3 whether these requirements are fulfilled by existing approaches. In Section 4, a general policy framework is introduced that features the concept of utility function policies. By means of customizable matching rules the framework can be adapted to arbitrary functional and non-functional properties. The specification of preferences on functional properties requires to define constraints on the behaviour of Web services, which is discussed in Section 5. In Section 6, a prototypical implementation of the policy evaluation algorithm is presented that builds upon a standard OWL-DL reasoner. Finally, we conclude in Section 7 with a brief summary and a short outlook.

2 Requirements

In this section, we motivate with examples various requirements that a Web service policy language should fulfill in order to be suitable for dynamic Web service markets. We distinguish between functional and non-functional proper-
Fig. 1. Model of a Web service concerning functional properties illustrated with three actors. The dashed lines denote the control flow from left to right and the dotted lines denote the data flow from the activity adorned with '-' to the activity adorned with '+'.

2.1 Requirements Concerning Functional Properties

A Web service operates on resources that can be real world objects or information objects. A Web service expects resources from the client and delivers resources to the client. A Web service is a process that in general may involve many actors with multiple interactions among them. An Interaction between two actors takes place in form of a message exchange. Figure 1 shows the functional model of a Web service that involves three actors A, B and C. We denote the set of actors involved in a Web service with \( \mathcal{A} \). Each actor \( A \in \mathcal{A} \) is associated with a resource schema \( S_A \), a finite set of resources \( R_A \) and a behavior \( B_A \). The resources can be real world objects or information objects (refer to Figure 1).

2.1.1 Constraints on Resources

Consider for example a library that maintains a wish list of books where the library members can enter the ISBN of books they would like to have. On the basis of some ranking (e.g. number of members that wish the book) of the desired books the library system needs to place book orders at regular intervals. At the time of designing and implementing the library system the software engineers need to find book selling Web services. Suitable book selling services are those that accept an ISBN and deliver a book. Considering
the library scenario a bit deeper, we identify that the software engineers are actually interested in Web services that do not sell just any book, but exactly the book that is ordered by the library software. In this example a suitable Web service is one that delivers a book with the same ISBN as the one it obtains as input. In realistic scenarios we must consider that there are many book selling Web services available and there is no global vocabulary for the domain of books that every Web service provider can or wants to use for describing the resources of his book selling Web service. As a result, we have to assume that in general Web service providers describe their Web services with their respective vocabularies independently of other Web service providers. Similarly, a requester needs a vocabulary that he understands to be sure that the constraints that he specifies capture the intended meaning.

**Requirement 1 (Constraints on Resources)** *It must be possible to define constraints on the values and types of input and output parameters of a Web service. The formalism for specifying constraints on resources must be able to consider mappings among the vocabularies while checking the satisfiability of constraints.*

2.1.2 *Constraints on Behavior*

A suitable Web service for the library system must have the matching (opposite pole) communication pattern so that the interaction between the Web service and the client can take place. If the Web services had only one input activity and only one output activity, one may assume that a Web service always perform the output activity after the input activity. As a consequence matching the communication patterns of the client and a Web service would be trivial. However, in general this is not the case. Even if Web services abstract from implementation details, the process triggered by invoking a Web service may be very complex involving multiple interactions with the client or other Web services. Consider a book selling Web service that after receiving a book order, sends a confirmation to the client and expects a back confirmation. Only after receiving the back confirmation from the client, it sends the ordered book to the client. Even if the Web service receives an ISBN in the first input activity and output the book with the same ISBN in the last activity, it may not be suitable for the library system, if the library system does not foresee to receive an order confirmation and sending a back confirmation. In other cases, an opposite situation may occur. That is, a client system is ready to receive a book only after it has received an order confirmation and sent a back confirmation, but the Web service neither sends any order confirmation nor it waits for a back confirmation before sending the book to the client.

While the above requirement concerns the public communication protocol of the Web services (choreography), there are use cases, in which a requester
may be interested in specifying constraints on the internal communication structure of a Web service (orchestration). Consider, for example a company internal Web service exposing a complex order process. It may be desired that the ordered good must be approved by at least two managers before it is bought. In other words, this means that in order to check whether the Web service is compliant to company’s policies, one must be able to reason about the information flow and the order of activities that transport information.

Requirement 2 (Constraints on Communication Patterns)  
In order to find Web services that can be incorporated in the client system it must be possible for the requester to specify constraints on the public communication pattern (choreography) as well as internal communication pattern (orchestration) of a Web service.

A Web service execution may cause changes in the knowledge state of the participants. For example, charging requester’s credit card as a consequence of the order he has placed is performed by means of an update (setting the available credit amount to a lower value) in the database of the corresponding bank. While searching for desired Web services, a requester may be interested in specifying which effects he wishes to take place and which not.

Requirement 3 (Constraint on Changes of Resources)  
The formalism for specifying constraints must support specification of desired and undesired changes in the resources.

Requirements 2: Constraints on Communication Patterns and Requirement 3: Constraints on Changes of Resources can be seen as constraints that specify desired or undesired situations during the execution of a Web service. Informally, such a situation describes the knowledge states of the involved actors and their possible behavior at a particular point in time. Formally, such a system can be described as a labeled transition system.

Definition 1 (Labeled Transition System)  
A labeled transition system is a tuple \((S, A, S)\) where \(S\) is a set of states, \(A\) is a set of actions and \(\rightarrow \subseteq S \times A \times S\) is a ternary relation between states and actions called transition. If \(s, s' \in S\) and \(a \in A\), then \((s, a, s') \in \rightarrow\) represents a transition from state \(s\) to \(s'\) triggered by action \(a\) and is written \(s \xrightarrow{a} s'\).

A state \(s \in S\) characterizes a system or system component and is usually described by a set of attributes (directly or indirectly) measured by a sensor. In a current state \(s\) a certain set of actions \(A\) can be taken which results in a transition to a new state \(s'\).

Figure 2 shows on the left hand side various situations or states that may occur during the execution of the example Web service depicted in Figure 1. The right hand side of the Figure 2 shows the labeled transition system of the ex-
ample Web service. For $A = \{A_1, \ldots, A_n\}$, the sets of resources $R_{A_1}, \ldots, R_{A_n}$ of the agents $A_1, \ldots, A_n$, together with their behaviors $B_{A_1}, \ldots, B_{A_n}$ at some given point of time, describe the state of the system at that time.

2.2 Requirements Concerning Non-Functional Properties

In practical scenarios, a user often selects a service from the set of services with same functionality on the basis of its non-functional properties. For example, from many book selling Web services, the library wishes only Web services that offer a customer hotline. A very important issue concerning non-functional properties is that of interoperability, since in distributed and open environments, it is not realistic to assume a global vocabulary of the names of non-functional properties. Rather, Web services providers should be able to chose the property names independent of other Web service providers. For example, if a Web service property calls a property "hotline", another Web service might want to call it "helpdesk". We denote with $P$ the set of all non-functional properties.

**Requirement 4 (Constraints on Non-functional Properties)** The formalism should allow a requester to specify constraints on the non-functional properties in an interoperable way to restrict the set of involved actors.

2.3 Requirements Concerning Combination of Constraints

Above we have identified that a user should be able to specify a constraint on functional and non-functional properties of Web services. In practice, often a user wishes to specify more than one constraint and combine them logically. For example, a user may wish to search for book selling services that offer customer hotline and charge the credit card after the delivery of the ordered book.

**Requirement 5 (Combination of Constraints)** The formalism should allow the specification of complex constraints from simpler constraints of possibly...
Table 1
Assessment of related work.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWL-S [42]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>(✓)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>WSMO [39] / WSML [7]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Berardi et al. [4]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>KAoS[45]/REI[20]/Kolovski et al. [27]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Contract Languages [15,40,38,36]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bienvenu et al. [5]</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>(✓)</td>
</tr>
</tbody>
</table>

different types.

2.4 Requirements Concerning Soft-Constraints

Until now we considered constraints as knock-out criteria. That is, if a Web service does not fulfil a constraint, it should not be detected as a match. However, in practice, often a user’s constraints are not always so hard. Rather a user should be able to specify his preferences on different alternatives. For example, instead of completely ruling out a Web service that does not offer a customer hotline, a user may wish to specify that he prefers the one with customer hotline to one without customer hotline. In another example, a user may wish to specify that he prefers Web service that charge the credit card after the delivery to those that charge the credit card before the delivery. In order to decide now whether a Web service that provides a customer hotline and charges the credit card before delivery should be preferred to a Web service that provides no customer hotline but charges the credit card after delivery, cardinal preferences that explicitly specify the trade-off between alternatives are required.

Requirement 6 (Specification of Preferences) *The formalism should allow to specify preferences on different types of constraints and constraint combinations.*
3 Related Work

In this section, we review literature with respect to the requirements for Web service policies identified above. Table 1 provides an overview and shows whether the approaches fulfill our requirements (indicated by a check mark). A first set of approaches suggest to use ontologies for describing Web services.

3.1 Web Service Description

One of the early approaches that present an ontology for describing Web services is OWL-S [42]. The OWL-S Matchmaker uses OWL-S Profile for describing Web services. Even if OWL-S Profile is designed for modeling pre and post conditions in addition to the types of input and output parameters of the Web services, there is still no concrete formalism fixed for describing the conditions. As a result, the constraint specification reduces to the types of input and output parameters. So, it neither allows the specification of relationships between inputs and outputs nor constraints on the temporal structure of a Web service or preferences over functional and non-functional service properties.

Another approach to semantic Web service descriptions is WSMO (Web Service Modelling Ontology) [39] which is expressed via WSML (Web Service Modelling Language) [7]. A WSML goal specification is identified by the `goal` keyword and consists of capability and interfaces. A capability description consists of shared variables, pre- and post conditions as well as assumptions and effects. Due to the availability of shared variables WSML goals are more expressive than OWL-S Profiles. The interface description is used to specify the desired choreography and orchestration of a Web service. In other words, WSMO interface used in a goal specifies constraints on the dynamic behavior of a Web service. However, currently there is no concrete proposal for describing the choreography and orchestration (Requirement 2: Constraints on Communication Patterns) and updates of resources are not supported (Requirement 3: Constraints on Changes of Resources). Furthermore, due the separation of capabilities and dynamic behavior already at the conceptual level, one would not be able to specify temporal constraints on effects, which contradicts Requirement 5: Combination of Constraints. That is, constraints like “the action ‘charge credit card’ (effect) should take place after delivery (choreography)”. Furthermore, WSMO goal specification does not capture preferences which are required to rank and select services (Requirement 6: Specification of Preferences).

Perhaps, the work that is closest to our work as far as the description of Web services is concerned is [4], in which an approach is presented to characterize
Web services with their transition behavior and their impacts on the real world (modeled as relational databases). In our work, local knowledge bases of the participating actors are represented with a decidable description logic, which can be helpful in proving decidability of discovery and composition algorithms. In addition, using description logic is also more suitable for the Web since the Web Ontology Language OWL standardized by W3C is based on description logics. Another major difference is the choice of the logic for specifying temporal constraints on Web services. [4] uses propositional dynamics logic (PDL) whereas we have used a more expressive logic μ-calculus. In contrast to our work, [4] does restrict their approach to goal policies which are not sufficient for ranking Web services and negotiating about Web service properties.

3.2 Policy Specification

As discussed above, the state of the art regarding Semantic Web Services currently provides only limited support for expressing constraints on non-functional service properties. Therefore, policy languages such as KAoS [45], REI [20] and the work presented by Kolovski et al. [27] can be used to extend the service ontologies (e.g. [21]). While KAoS and the approach by Kolovski et al. are based mainly on OWL-DL, REI uses OWL-Lite only as syntax for exchanging policies and performs reasoning based on a logic programming approach supporting rights, prohibitions, obligations and dispensations. How KAoS policies are used for managing semantic Web services is outlined in [45]. Since pure OWL-DL is not fully sufficient to cope with the situation where one value depends on other parts of the ontology, they extend the logic by so called role-value maps (see also [2]). Recently this issue is addressed by combining description logics and logic programming [35,44,34]. In our work, we adhere to such a combination by using OWL-DL with DL-safe rules (see Section 4.2.1).

The approach to formal policy languages by Kolovski et al. [27] defines the semantics of the WS-Policy specification also by means of an OWL-DL ontology. This allows them to use a standard OWL reasoner for policy management and enforcement. However, due to the open world assumption of OWL their results sometimes are counterintuitive. For our work the open world assumption is currently no problem, since only subsumption-based matching of attribute values is used in the policy evaluation process. In case other DL-based reasoning techniques such as satisfiability checking are introduced, the open world assumption of OWL can become problematic. This issue has been addressed by the use of non-monotonic logics in [26,14,13] and by manually closing the ontology. The latter can be realized by changing axioms in the ontology [10] or by introducing additional restrictions to the knowledge base [12]. The major disadvantage of the mentioned policy formalisms in our context is their limitation in terms of expressing fine-grained preferences (Requirement 6: Specification of Preferences). Using these approaches only hard constraints
can be expressed and policy evaluation thus always leads either to true or false, which is insufficient for ranking of Web service offers.

Several approaches for representing preference information have been proposed in literature, e.g. based on utility functions [46,9,22], fuzzy logics [6,25] or other forms of preference representation like [24,3]. However, they typically do not consider a Web environment where preference information is exchanged between heterogeneous agents and matching of attribute values has to be done by considering rich semantic structures. Since service offers and requests are typically defined on different levels of abstraction (e.g. while a service requester might specify that asymmetric encryption methods are preferred, a service provider might specify that only the RSA algorithm is supported), this is not sufficient for specifying policies for open and heterogeneous environments such as the Web.

More expressivity in this context is provided by rule languages, such as SweetDeal [15], DR-NEGOTIATE [40,11], RBSLA [38] and the approach presented by Oldham et al. [36]. While the former use defeasible reasoning (i.e. Courteous Logic Programs or defeasible logic) to specify contract templates (including preferences), the latter formalizes WS-Agreement and expresses preferences using a proprietary rule language. Similar to our approach they all feature automatic reasoning based on a formal logic. However, there are some issues regarding the use of a (pure) logic programming paradigm. Often such languages do not provide full-fledged declarative semantics and thus combining rules from different sources becomes highly problematic. In fact, manual integration of the different logic programs might be required. Since in our setting, policies have to be integrated from different sources, this is a major drawback. There are few approaches that combine preference representation formalisms with techniques for handling information from heterogeneous sources. For example, Klein et al. [25] combine fuzzy concepts with service hierarchies, Lamparter [29] combines utility theory with OWL-based service descriptions, and Felfernig et al. [8] as well as McGuinness et al. [33] combine constraint satisfaction problems with description logic based product descriptions. However, all these approaches do not consider temporal behaviour as required by Requirement 2: Constraints on Communication Patterns.

Temporal aspects are addressed by the approach presented in [5], which allows the specification of temporal constraints and shows how qualitative preferences can be attached to these constraints. While qualitative preferences are often sufficient to determine rankings of Web service offers, they are not suitable to define the trade-offs between different alternatives, which is often required during negotiations. Furthermore, since the best alternative might not be good enough, an absolute measure of suitability is required. In our work, we address these problems by introducing a cardinal utility measure. In addition, Bienvenu et al. [5] do not elaborate on the specification of constraints
on non-functional properties of Web services (Requirement 2.2: Constraints on Non-functional Properties).

4 Specification of Utility Function Policies

In this section, we show how utility function policies that are expressed formally via an ontology can provide the required expressivity to meet Requirements 1 – 6. We first introduce in Section 4.1 an abstract policy model that takes up traditional concepts from decision theory and shows how preferences can be captured by means of utility functions. This is important to express trade-offs between different alternatives and thereby to meet Requirement 6: Specification of Preferences. After introducing the abstract policy model, we show how this model can be implemented using existing ontology languages in Section 4.2.

4.1 Policy Model

As postulated by Requirement 6: Specification of Preferences, in order to support the optimal allocation of Web services in a system, the trade-offs between different Web service configurations have to be explicitly specified. Policy that provide the necessary expressivity to capture this information are referred to as utility function policies [23], which represent the functional relation between alternatives and their value for the decision maker. Over the last decades, there has been a broad stream of work about specifying utility functions [46,9,22]. The goal is to provide sufficient expressivity for modeling complex decisions, while keeping the elicitation and computation effort at an admissible level. In a Web services scenario, a utility function can be defined as a function $U : C \rightarrow \mathbb{R}$ mapping the set of possible Web service configurations $C$ to a real-valued measure reflecting the value a decision maker attaches to a certain alternative. The set of possible configurations $C$ is defined as the cartesian product $C = P_1 \times \cdots \times P_n$, where the $P_1, \ldots, P_n \in \mathcal{P}$ refers to functional and non-functional attributes that describe a Web service. The utility is measured on a cardinal scale, which allows making statements about the relative as well as absolute suitability of a configuration. They thus generalize the concept of goal policies by allowing not only two levels 'admissible' and 'not admissible', but make all configurations comparable by introducing a preference structure over the configurations.

Definition 2 (Preference Structure) A preference structure is defined by the complete, transitive, and reflexive relation $\succeq$. For example, the configuration $c_1 \in C$ is preferred to $c_2 \in C$ if $c_1 \succeq c_2$. The preference structure can be
derived from the utility function $U(c)$ by means of the following condition:

$$\forall c_a, c_b \in C : c_a \succeq c_b \iff U(c_a) \geq U(c_b)$$

(1)

Since policy-based decision making approaches are usually applied in large-scale applications, typically more than one policy is specified in order to regulate a certain decision. Therefore, traditional policy languages support conjunctions and disjunctions of policies. Since in this work we focus on the specification and evaluation of a single policy, introducing primitives for modeling conjunctions and disjunctions goes beyond the scope of this paper and is discussed in [29, Chp. 6.2].

In the context of a service-oriented architecture, utility functions policies can be used on requester-side to specify preferences, assess the suitability of trading objects and derive a ranking of trading objects based on these preferences and on provider-side to specify the reservation price of a provider. The latter case is illustrated by the following example.

**Example 1** We take up our initial example and assume a Web service request where preferences are specified via a utility function policy. The policy captures an additive utility function, where valuations are given for each attribute separately. This typically requires mutual preferential independency between the attributes [22] and can be expressed via the following function:

$$U(c) = \sum_{i \in [1,n]} \lambda_i u_i(p_i)$$

(2)

That means the willingness to pay of the requester depends on the configuration and can be defined via functions for each of the independent attributes $u_j(p_j)$ as follows:

- **Functional property ‘Behaviour’**: $u_1(p_1)$ is defined by the points $P(“payment before delivery”, 0)$ and $P(“payment after delivery”, 1)$
- **Non-functional property ‘Response Time’**: $u_2(p_2) = 1 - \frac{1}{10}p_1$ with $p_1$ measured in seconds.
- **Non-functional property ‘Hotline’**: $u_3(p_3)$ is defined by the points $P(“yes”, 1)$ and $P(“no”, 0.5)$.

Furthermore, we assume weights of $\lambda_1 = 0.5$, $\lambda_2 = 0.2$, and $\lambda_3 = 0.3$. For example, given this request a Web service that charges the credit card before delivery, guarantees a response time of 2 seconds and provides no hotline would consequently lead to a utility of .65.

Since policies can be specified by many different parties (e.g. different departments in a company), methods for aggregating policies to one consistent
decision rule are required. We therefore discuss policy aggregation in the next section.

4.2 Policy Ontology

Given the abstract policy model above, we now focus on implementing this model using existing standards and tools for the open and heterogenous Web environment. We use the Web Ontology Language OWL [47] together with its rule extension SWRL [18] to implement our policy model, which allows us to perform sophisticated matchmaking and ranking of services by means of logical inferencing.

4.2.1 Ontology Formalism

OWL is an ontology language standardized by the World Wide Web Consortium (W3C) [47] and is based on the description logic (DL) formalism [2]. Due to its close connection to DL it facilitates logical inferencing and allows to derive conclusions from ontologies that have not been stated explicitly. We briefly review some of the modeling constructs of OWL.

The main elements of OWL are individuals, properties that relate individuals to each other and classes that group together individuals which share some common characteristics. Classes as well as properties can be put into subsumption hierarchies. Furthermore, OWL allows for describing classes in terms of complex class constructors that pose restrictions on the properties of a class. Based on names for concepts (as \( C, D, \ldots \)), roles \( (R, S, \ldots) \), and individuals \( (a, b, \ldots) \), OWL provides constructors like negation, conjunction, disjunction, existential quantifier, universal quantifier and qualified number restrictions to build complex concepts from simpler ones. Further, it supports concrete datatypes and there exist corresponding axioms for quantifiers and cardinality constraints for roles with a datatype range. Therefore, an OWL knowledge base consists of a set of axioms, which can be distinguished into terminological axioms (building the so-called TBox \( T \)) and assertional axioms or assertions (constituting the ABox \( A \)). A TBox consists of a finite set of concept inclusion axioms \( C \sqsubseteq D \), where \( C \) and \( D \) are either both concepts or relations. The A-Box consists of a finite set of concept assertions \( C(a) \), role assertions \( R(a, b) \), individual equalities \( a = b \), and individual inequalities \( a \neq b \). Those assertional axioms or assertions introduce individuals, i.e. instances of a class, into the knowledge base and relate individuals with each other. For details about the semantics of OWL-DL constructors, T-Box axioms and A-Box axioms, we refer to [47,17,2].

For the declarative formulation of policy evaluation process in form of rules,
we require additional modeling primitives not provided by OWL. We use the Semantic Web Rule Language (SWRL) [18] which allows us to combine rule approaches with OWL. We restrict ourselves to a fragment of SWRL called DL-safe rules\(^3\) [35], which is more relevant for practical applications due to its tractability and support by inference engines such as KAON2\(^4\). For the notation of rules we rely on a standard first-order implication syntax.

4.2.2 Representation and Evaluation of Policies

In this section, we show now how the policy model introduced in Section 4.1 can be implemented using the standard ontology formalisms introduced above. Figure 3 sketches the policy ontology that enables the representation of Web service policies based on the concept of utility function policies. In our ontology this is reflected by the concepts Web Service Policy that, on the one hand, describes the technical aspects of a Web service such as the physical address captured by the concept Service and, on the other hand, enables the specification of functions that assign certain valuations to service properties. This requires to express functions in a declarative way using the ontology formalism at hand. We realize this by specializing the concept Utility Function. Approaches on how functions are expressed via an ontology are discussed in [31,32]. There are mainly three techniques for modeling functions: Point Based Functions, Piecewise Linear Functions and Pattern-based Functions.

---

\(^3\) DL-safety restricts the application of rules to individuals that are explicitly mentioned in the ontology. However, this restriction does not affect the suitability of DL-safe rules in our scenario.

\(^4\) http://kaon2.semanticweb.org/
While the former approach is restricted to discrete functions, the latter two approaches are applicable to arbitrary continuous functions. To illustrate the basic idea, the concept Point Based Function is introduced in more detail.

A Point Based Function can be used for discrete properties. In this context the utility function \( u_\ell(p_\ell) \) is modeled by specifying sets of Points that explicitly map a Property Value referred to in a Property Value Pair to a utility measure. Based on this model the following rule can be used to evaluate the policy \( p \) with respect to a given configuration \( c \). A Configuration is modeled as a concept that has exactly one (datatype or object) property for each service property.

\[
\text{evaluatePolicy}(p, c, u) \leftarrow \text{Policy}(p), \text{Configuration}(c), \\
\bigwedge_{i=1,\ldots,n} \text{defines}(p, f_i), \\
\text{PointBased}(f_i), \text{constitutedBy}(f_i, pt_i), \\
\text{propertyValue}(pt_i, pv_i), \text{utility}(pt_i, u_i), \\
\text{hasProperty}_i(c, pr_i), \text{match}(pv_i, pr_i)), \\
\text{SWRLB:add}(u, u_1, \ldots, u_n)
\] (3)

Rule 3 basically captures the additive utility function given in Equation 2 by comparing each property of the configuration with the appropriate Points in the definition of the Point-based Function and by adding up the utility values belonging to matching property values. In order to determine whether property values match each other, the match-predicate is applied. In the following, we discuss how this predicate is defined.

The way property values are modelled in the ontology depends on the type of the property. For instance, in case of the non-functional properties ‘Response Time’ and ‘Hotline’ the Property Value can be directly represented as literals in the ontology. In the evaluation process such property values can be compared by operators such as ‘=’, ‘<’, etc. We thus define matching rules to specify how a certain property should be matched. For example, the property ‘Hotline’ should be matched by a ‘=’-operator:

\[
\text{match}(x, y) \leftarrow \text{hasProperty}(x, \text{‘Hotline’}), \text{hasValue}(x, v1), \\
\text{SWRLB:equals}(v1, y)
\] (4)

Since there are settings in which all possible property values cannot be enumerated or are not known, this approach is not sufficient. For example, assume the property ‘Encryption’ that defines possible encryption algorithm which are supported by a Web service. Since enumerating all possible encryption algorithms might not be feasible, an alternative approach is required. Therefore, we
use a meta-modeling approach in which instances of the concept Property Value in our policy ontology are considered as concepts in a domain ontology. For example, for the service property ‘Encryption’ a concrete Property Value could be ‘Asymmetric’ identifying all possible asymmetric encryption algorithms as defined in the encryption ontology (see Figure 3). However, the matching of such concept-based property values has to be done differently compared to the instance-based property values specified above. To realize the concept-based matching, we build on well-known notions of matching for Semantic Web services, such as subsumption-based ‘plugin’ or ‘exact’ matches [37]. For the property ‘Encryption’ this means that we could use the following matching rule to indicate that a certain utility should be associated with all asymmetric encryption algorithms. The built-in subsumes implements the algorithm that checks for subsumption between two OWL concepts.

\[
\text{match}(x, y) \leftarrow \text{hasProperty}(x, \text{‘Encryption’}), \text{hasValue}(x, v1), \\
\text{subsumes}(v1, y)
\] (5)

While policies on non-functional properties can be specified purely based on the ontology languages OWL-DL and DL-safe rules, policies on functional properties require statements about the temporal structure of actions, which cannot be expressed using description logics. Therefore, we discuss in the next section how we can incorporate a temporal logic into our framework and show how the appropriate matching rule is defined.

5 Specification of Constraints on Functional Properties of Web Services

In this section, we present a formalism to specify constraints on behaviour of Web services. The main aim of a goal specification language is to provide a syntax and semantics to restrict the set of states in such a labeled transition system (refer to Definition 1). That is, by specifying constraints, a user should be able to fix the set of states that may/should occur in the execution of a Web service or a Web service composition. Note, that our view of Web services is not restricted to ”black-box” like Web services with interfaces for one input and one output. As illustrated in Figure 1 and Figure 2, a Web service may involve multiple actors and multiple interactions (not only with the client). So, the only difference between a Web service and a Web service composition is that the composition is more like a business process, the description of which is at first available only locally to the client that has composed the business process. If the client wishes to offer the business process as a Web service, he or she needs to compile the description of the Web service suitable for publishing
Table 2
\(\mu\)-calculus Syntax and Semantics

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>true</td>
<td>(S)</td>
</tr>
<tr>
<td>False</td>
<td>false</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>Atomic Proposition</td>
<td>(P)</td>
<td>(V_{Prop}(P))</td>
</tr>
<tr>
<td>Conjunction</td>
<td>(\phi_1 \land \phi_2)</td>
<td>(\llbracket \phi_1 \rrbracket \cap \llbracket \phi_2 \rrbracket)</td>
</tr>
<tr>
<td>Universal Quantifier</td>
<td>([a]\phi)</td>
<td>({s</td>
</tr>
<tr>
<td>Negation</td>
<td>(\neg \phi)</td>
<td>(S \setminus \llbracket \phi \rrbracket)</td>
</tr>
<tr>
<td>Minimal Fixpoint</td>
<td>(\mu Z.\phi)</td>
<td>(\cap {S \subseteq S</td>
</tr>
<tr>
<td>Disjunction</td>
<td>(\phi_1 \lor \phi_2)</td>
<td>(\llbracket \phi_1 \rrbracket \cup \llbracket \phi_2 \rrbracket)</td>
</tr>
<tr>
<td>Existential Quantifier</td>
<td>(\langle a \rangle \phi)</td>
<td>({s</td>
</tr>
<tr>
<td>Maximal Fixpoint</td>
<td>(\nu Z.\phi)</td>
<td>(\cup {S \subseteq S</td>
</tr>
</tbody>
</table>

by hiding internal information while still not losing the semantics of the Web service.

We will first introduce \(\mu\)-calculus and show how it can be used to specify constraints on a labeled transition system. \(\mu\)-calculus is one of the most expressive temporal logics while still being decidable[28]. Then we will add a facility to specify constraints on the involved resources by giving structure to \(\mu\)-calculus propositions and actions.

5.1 Specifying Constraints on Behavior with \(\mu\)-calculus

\(\mu\)-calculus, as it is mostly used today was first introduced in [28]. Let \(Var\) be an (infinite) set of variables names, typically indicated by \(X,Y,Z\ldots\); let \(Prop\) be a set of atomic propositions, typically indicated by \(P,Q\ldots\); and let \(A\) be a set of actions typically indicated by \(a,b\ldots\). The syntax of formulae (with respect to \((Var, Prop, A)\) is presented in column two of Table 2.

**Definition 3 (Structure)** A structure \(T\) over \(Prop, A\) is a labeled transition systems, namely a set \(S\) of states and a transition relation \(\xrightarrow{} \subseteq S \times A \times S\), together with an interpretation \(V_{Prop} : Prop \rightarrow P(S)\) for the atomic propositions. We often write \(s \xrightarrow{a} t\) for \((s,a,t) \in \rightarrow\).

Given a structure \(T\) and an interpretation \(V : Var \rightarrow P(S)\) of the variables, the set \(\llbracket \phi \rrbracket_V\) of states satisfying a formula \(\phi\) is defined in column three of Table 2.
With fixed point operators, one can define all known constraints on the temporal behavior. In the following, we give examples of some most widely known temporal operators. The formula $\mu X. (\phi_2 \lor (\phi_1 \land \langle - \rangle \text{true} \land [-]X))$ describes $\phi_1$ until $\phi_2$, as it can be read as: either $\phi_2$ holds in the current state or sooner or later the process reaches a state in which $\phi_2$ holds and until then $\phi_1$ holds. The modality eventually $\phi$ can be easily defined as true until $\phi$.

5.2 Adding Structure to $\mu$-calculus Propositions and Actions with OWL-DL

$\mu$-calculus in its pure form abstracts from the meaning and structure of the propositions. In our formal model, the set of propositions of an actor correspond to the facts in the knowledge base of the actor. The facts (explicit or derived) in the knowledge base of an actor at some point of time represent the set of propositions that are true at that point of time.

We use a OWL-DL description $D$ in place of $\mu$-calculus atomic propositions (denoted by $P$ in Table 2), which is true if the OWL-DL concept $D$ has some instances in the knowledge base of the Web service.

For actions (see $\mu$-calculus formulas $\langle a \rangle \phi$ and $[a]\phi$), we make similar structural extensions. We differentiate between input and output actions by using the sign ‘+’ or ‘−’ for input and output actions respectively. We give an input action $a$ the structure $+((P,A,v_1:T_1,\ldots,v_m:T_m)$, which means a Web service performs an input action that can receive $m$ values of types $T_1,\ldots,T_m$ respectively over a channel of protocol $P$ at the address $A$. Similarly, we use $-(P,A,Q_1,\ldots,Q_m)$ for an output action, which means a Web service performs an output action that sends $m$ values which are answers of the queries $Q_1,\ldots,Q_m$ respectively over a channel with protocol $P$ and address $A$.

Having introduced our goal specification language, we now look back to the requirements identified in Section 2 and discuss how our language covers the requirements. In an input formula $+((P,A,v_1:T_1,\ldots,v_m:T_m)$, $T_1,\ldots,T_n$ are desired input types and in an output formula $-(P,A,Q_1,\ldots,Q_m)$, $Q_1,\ldots,Q_m$ are description logic queries. The queries $Q_1,\ldots,Q_m$ can use the process variables, e.g. those in an input formula. This way it becomes possible to establish relationships between inputs and outputs. Interoperability is covered by the fact that we use description logic OWL-DL for specifying constraints on terminologies. Simple mappings between terminologies can be expressed via subsumption relationship, whereas complex mappings with DL-safe rules [16]. So, the Requirement 1: Constraints on Resources is fulfilled. Requirement 2: Constraint on Communication Patterns is covered by formula types $[a]\phi$ and $\langle a \rangle \phi$. By using OWL-DL descriptions as propositions we fulfill Requirement 3: Constraints on Changes of Resources. Requirement 5: Combination of Constraints
is fulfilled by the logical connectors available in the language.

5.3 Evaluating Constraint Satisfaction

Having a language for specifying constraints on the behaviour of Web services, we now turn our attention to the problem of checking whether a Web service behaviour fulfils a given constraint. Recall that formally we view a Web service as a labeled transition system. There are approaches like OWL-S, in which a Web service process is described. Such a process model can be converted to a labeled transition system with an appropriate execution semantics (Figure 2). So, in any case we have a labeled transition system of a Web service and the states of the LTS are annotated with the formulas of the language we presented above. [41] first introduced a set of tableau rules for model checking $\mu$-calculus formulas. We use these tableaux rules as a basis and modify them as required by our structural extensions for atomic propositions and actions. An approach for calculating the LTS for a Web service online, which is not beforehand but at the time of checking the constraint satisfaction, is presented in [1]. It also shows that the time complexity of checking, whether a Web service with $n$ states fulfills a constraint represented by a formula of length $m$ is $O(n \cdot m)$. In practical scenarios, $m$, the length of the formula, is typically not very large. If the Web service process is very complex, containing many concurrently acting parties, then $n$, the number of states in the Web service process can become relatively large. However, note that automatic verification of such processes makes even more sense in case of complex processes, since the manual verification will be slower and error-prone.

5.4 Integrating Constraints on Web Services

In the previous section, we have seen how utility function policies can be specified by defining utilities for different Web service configurations. Thereby, the configurations are described by values of Web service properties. In this section, we have developed a formalism for specifying constraints on functional properties of Web services. We obtain the complete policy framework by viewing a formula of the constraint specification language developed in this section as a property value for which a utility can be specified. It is up to the user whether he specifies one formula that defines values of all desired properties or he specifies more than one formulae, assigns a utility to each formula and determines the function for aggregating the various utilities.

**Example 2** In Example 1, there is a functional property ‘Behaviour’ with values “payment before delivery” and “payment after delivery”. With the formalism developed in this section, “payment before delivery” can be specified
as
\[ \neg (\text{delivery}) \text{true} \text{until payment} \wedge (\text{delivery}) \text{true}. \]
The above formula says that there should not be any delivery before payment and there should be a delivery, which is equivalent to saying that the delivery should take place after the payment. The constraint “payment after delivery” can be specified as
\[ \neg \text{payment} \text{until (delivery)} \text{true}. \]

Note, that the formula actually specifies that the client does not wish to pay before delivery and leaves open whether he wishes to pay after delivery or not. So, the services that are free (client does not have to pay at all) as well the services where the client has to pay after the delivery will satisfy the formula.

Though the above example already shows the added value of our formalism to the existing work, it is still rather simple as compared to those needed in real practical scenarios. In this context particularly constraints involving service behaviour (as described in Example 2) and resources are highly relevant. An example for such a constraint is given in the following.

**Example 3** Considering the book selling web service scenario a bit more deeply, we observe that a requester, in addition to the constraint from the above example, also wishes to makes sure that (1) the service delivers not just any book but the book that he orders (2) the amount the credit card is charged for should be equal to the price of the book. These constraints can be formulated as follows:

\[
[+\text{order}(\text{ob}, \text{cc})] \text{isbn}(\text{ob}, \text{obISBN}) \wedge \text{price}(\text{ob}, \text{obPrice}) \wedge \text{eventually}
\langle -\text{delivery}(\text{db}) \rangle \text{isbn}(\text{ob}, \text{dbISBN}) \wedge \text{equals}(\text{obISBN}, \text{dbISBN}) \wedge
\text{charged}(\text{cc}, \text{amount}) \wedge \text{equals}(\text{obPrice}, \text{amount})
\]

where ob and db and cc represent the ordered book, the delivered book and the credit card respectively. The names of the rest of the parameters of the actions and predicates are self explaining. Roughly the above formula says, that for all orders for a book ob that should be paid with the credit card cc, there should eventually be a delivery of a book db, the ISBN number of which is equal to that of the ordered book and the amount charged from the credit card (the same credit card that was entered while ordering and not any credit card), should be equal to the price of the ordered book.

In order to specify how the values of the property ‘Behaviour’ have to be matched, the following rule is added to the ontology. The predicate satisfied-by implements a model checking algorithm for verifying the desired behavioral properties [1, Chp. 7] and enables the integration of temporal reasoning into our overall policy framework presented in Section 4.
match(x, y) ← hasProperty(x, 'Behaviour'), hasValue(x, v1), satisfied-by(v1, y)  

(7)

6 Implementation

We have implemented a prototype for the policy framework presented in the previous sections. In the following, we describe the functionalities of the main components of the prototype.

First, we provide a repository that maintains a large number of web services offers. This repository is maintained by the KAON2 ontology management framework. The ontologies needed for modeling service offers and utility function policies as presented in Section 4 are known to the repository. The individual Web service offers including provider policies are saved in the repository as instances of the appropriate concepts of the ontologies.

Second, a client sided tool also allows users to specify their requests and send them to the server. In this context, policies can be specified by choosing one of the predefined utility function patterns and adjusting the corresponding parameters. New patterns can be added as long as their semantics can be captured by the provided SWRL built-ins [18] or by already existing patterns. In this case the new rule is transferred to the server along with the query. The server performs the matchmaking and evaluates the offers with respect to the request by comparing the corresponding policies. The ranked list of matches is sent back to the client tool, which shows the matches to the user.

First evaluations have shown that the use of utility function polices allows us to increase efficiency of representing Web service offers and requests, while keeping matching and ranking of offers and requests computationally tractable. Given an additive preference structure (as specified in Equation 2) the worst case space complexity for storing $n$ service offers with $m$ possible configurations each decreases from $O(nm)$ in case of traditional enumeration based approaches to $O(n + \log(m))$. For example, in a scenario with randomly generated 1000 offers with 100 configurations each, we can reduce the size of the knowledge base from 77 MB to 24 MB. A detailed discussion of this results can be found in [32]. In terms of computational tractability, two problems can be distinguished: the complexity of policy evaluation itself and the complexity of evaluating the matching rules for each attribute. As discussed in [30], evaluation of utility function policies can be reduced to a traditional mixed integer programming formulation, which can be efficiently solved using the well-known simplex algorithm. Our experimental results for example show

---

5 http://kaon2.semanticweb.org
that the degree of satisfiability can be calculated within 16 ms in case of 100 possible configurations and within 161 ms in case of 500 possible configurations. The complexity of matching attribute values depends on the predicate that is assigned to a certain attribute. While executing predicates such as \texttt{SWRLB:equals} is very fast, more complex predicates like \texttt{subsumes} or \texttt{satisfied-by} can be computationally more demanding. As an example, consider the \texttt{subsumes}-predicate used in Rule 5. The predicate tests for subsumption between two concepts in OWL-DL, which is shown to be NExpTime-hard [43, Corollary 4.13]. Evaluating temporal constraints using the \texttt{satisfied-by}-predicate has the time complexity $O(n \cdot m)$ where $n$ denotes the number of states in the LTS of the Web service process and $m$ the length of the formula [1, Chp. 7].

7 Conclusion and Outlook

In this paper, we have studied the problem of specifying Web service policies with the aim of facilitating automated discovery and selection of services. We have argued that on one side it is necessary to specify policies on functional properties (beyond the types of input and output parameters) as well as on non-functional properties, and on the other side that these policies should enable the specification of hard as well as soft constraints on properties of Web services.

We have first presented a general policy framework that allows the assignment of utilities to various types of Web service constraints in a flexible way. In case of Web service offers, such a utility measure can be used to represent prices of a concrete Web service configuration, whereas in case of a Web service request, it represents how much a particular Web service configuration is worth for the user. The general policy framework, that can actually be used for products other than Web services as well.

We then turned our attention to the Web service specific properties and presented a formalism that allows the specification constraints on the types and values of the objects involved during the execution of a Web service as well as on the behavior of a Web service. Then we showed how the constraints on the functional properties can be integrated in the overall policy framework, such that they can be considered while calculating the utility of a Web service. Finally, we have presented the main components of the prototypical implementation of our policy framework as well as some evaluation results.

In this paper, we have presented an expressive formalism for specifying policies on Web service properties. We have not dealt with the problem of policy elicitation. That is, the problem of how Web service providers and requesters know which policies they want to specify and how they can be supported in
identifying their policies automatically or semi-automatically. Another interesting problem for future research is service execution monitoring. After a requester has selected a Web service on the basis his own as well as the Web service provider’s policies, he executes the Web service a few times. Execution monitoring deals with recording the values of various properties during an execution and supporting a requester automatically or semi-automatically while identifying any non-compliance to the values promised by the Web service provider in his policy.

References


