The Design and Enforcement of A Rule-based Constraint Policy Language for Service Composition

Wei Wei, Ting Yu
Department of Computer Science, North Carolina State University
Raleigh, North Carolina, United States
wwei5@ncsu.edu, yu@csc.ncsu.edu

Abstract—Service composition is a new paradigm for efficient and cost-effective IT service provisioning over the network. To safely and effectively deploy composed services within an organization or among multiple domains, one must be able to specify and enforce a variety of constraints such as those derived from legal regulations, Quality of Service (QoS) requirements and privacy and security policies. In this paper, we show through concrete examples that constraints involving topologies are common for composed services but are rarely supported by existing service composition policy languages. We then present a rule-based constraint policy language built on top of a general graph-based service composition model. The language provides a unified treatment to support both attribute-based and topology-based constraints. We further design an ontology-based policy framework for effective policy enforcement and analysis. Finally, we implement a prototype policy management system for service composition, and conduct extensive experiments to evaluate the effectiveness of the proposed techniques.

I. INTRODUCTION

With the prevalence of concepts such as Software as a Service (SaaS) [1] and Service Oriented Architecture (SOA) [2], software services are increasingly built by composing services offered by different service providers over the network. Service composition has become an efficient and cost effective way to create value-added services. To safely and effectively deploy composed services within an organization or among multiple domains, one must be able to specify and enforce a variety of constraints such as those derived from legal regulations, Quality of Service (QoS) requirements and privacy and security policies [3]–[7].

Several policy languages have been proposed to specify policy constraints or assertions for supporting various service requirements, for example WS-PolicyConstraints [8], WS-SecurityPolicy [9] and REI [5]. The policies defined in those languages are usually represented as constraints on attributes of services. For instance, online shopping checkout services must be provided by trustworthy service providers such as PayPal and Google Checkout. Meanwhile, we observe that, besides constraints based on attributes of services, it is also common for real world systems to have constraints that rely on topologies of service compositions. For example, in a military information system, information should not be passed from a service of high secrecy level to one of low secrecy level according to multi-level security policies. However, to the best of our knowledge, no existing service composition policy languages support such topology-based constraints.

In this paper, we present a unified rule-based constraint policy language for service composition, which seamlessly supports both attribute-based and topology-based constraints. The policy language is based on a general graph-based service composition model, in which the topology of service composition is explicitly defined. In our proposed language, the basic construct is a Rule that defines a constraint in terms of pre-conditions and post requirements. For example, to protect privacy, a business such as Google may require that if information is sensitive, it should not be shared with or passed to other businesses [10]. Rules are further organized into rule sets to support complex combinations of constraints.

Once policies are specified for service composition, it is important to further support policy enforcement and analysis. The goal of policy enforcement is to check whether a service composition violates any requirement defined in policies, while policy analysis is to study the properties of policies, in particular whether it contains certain design errors. In this paper, we propose an ontology-based policy framework for effective policy enforcement and analysis. Instead of developing a new policy analysis tool, we show that the policy language we designed can be naturally mapped to an ontology reasoning system. The basic idea is to transform policies and service compositions to a service composition ontology defined using Web Ontology Language (OWL) [11] and a semantic web rule language (SWRL) [12], and utilize the reasoning capabilities provided by off-the-shelf OWL reasoners like Pellet [13] to implement our policy framework. Our major contributions are summarized as follows:

- We introduce topology-based constraints for service composition, and present a unified rule-based constraint policy language, which seamlessly supports both attribute-based and topology-based constraints.
- We propose an ontology-based policy framework for effective policy enforcement and analysis such as the detection of invalid service composition, policy conflict and redundancy.
- We implement a prototype policy management system for service composition based on Eclipse [14], and conduct
extensive experiments to evaluate the effectiveness of the proposed techniques.

The rest of the paper is organized as follows. We introduce constraints in service composition in Section II. In Section III, we define the service composition model, and present the language syntax and semantics. Section IV presents the detailed design of our policy framework. Section V provides a brief discussion of policy analysis services that our policy framework supports. The prototype implementation and evaluation are discussed in Section VI. Section VII compares our work with related work. The paper concludes in Section VIII.

II. CONSTRAINTS IN SERVICE COMPOSITION

In this section, we use a typical scenario in business, shown in Figure 1, to motivate the need for a new constraint policy language for service composition.

Figure 1 shows how a business incorporates service compositions and policies to effectively fulfill its business tasks, which involves three types of players: service composition designer, service provider and policy designer. The lifetime of service composition can be divided into three phases: design phase, service provider discovery phase and execution phase. During the design phase, a service composition designer creates service compositions according to the specifications of business tasks. At this phase, a service composition does not fulfill any function since it has not been assigned service providers to implement it. Assigning service providers to service compositions is exactly what happens during the discovery phase. Once service providers are assigned to a service composition to carry out specific services, the service composition enters into the execution phase, and becomes an executable service composition, which can provide the functionality required by the original business task.

Service composition is subject to a variety of constraints in terms of functionality, QoS requirements, security and privacy policies [3]–[5]. For example, the discovery process usually requires that the functionality offered by a service provider be matched with that required by the service. In addition to functionality matching, domain-specific constraints, security and privacy constraint policies may also be applied to service compositions. For example, medical information cannot be stored or processed by a service outside of the United States. To create these constraint policies is where policy designers come into play.

Constraints in the above examples can be expressed based on attributes of services or service providers. For example, if \( s.inputCategory = "medical" \), then \( p.location = "US" \), where \( s \) and \( p \) denote respectively a service and a service provider that offers \( s \). In addition to constraints based on attributes, it is also common that policies have constraints that rely on the topologies of service compositions. For example:

1) Information should not be passed from a service with high security level to a service with low security level in order to be compliant with multi-level security policies.

2) If the output of a service is sensitive, for example containing credit card number, bank account number and security code, all services where its output information passes should provide a secure channel for data transmission, which encrypts data before sending them out.

It is not intuitive to specify the above policies using attributes of services or service providers since they add restrictions to the topology relations among services, not simply their attributes. Consider the first constraint. Intuitively the way that information flows relies on how services are connected and organized in a service composition. We need to explicitly refer to the topology of a service composition to specify such policies. Thus, a high level policy language directly supporting topology-based constraints is needed.

III. CONSTRAINT POLICY LANGUAGE

A. Service Composition Model

In the following, we will discuss service composition according to the three phases, design phase, discovery phase and execution phase.

In the design phase, we refer to service composition and service as abstract service composition and service function respectively, since they are not able to be executed. A service function \( f \) is described by a set of attributes \( \{a_1, \ldots, a_n\} \), which may comprise its type, input and output, authentication method, and encryption algorithm, etc. An abstract service composition is composed of a set of service functions, defined as follows.

Definition III.1. (Abstract Service Composition) An abstract service composition is modeled as a directed graph \( SCA = \{F, T\} \), where \( F \) is a set of vertices representing service functions \( \{f_1, \ldots, f_n\} \) and \( T \) is a set of directed edges representing topology relations \( \{t_1, \ldots, t_m\} \) between service functions.

Each \( t_k = (f_i, f_j) \) in \( SCA \) represents direct dependency including both control flow and data flow from \( f_i \) to \( f_j \). Usually, service providers advertise the services they offer, each of which is described as a set of attributes \( \{b_1, \ldots, b_n\} \) such as provider name and response time. During the discovery phase, each service function \( f_i \) will be assigned a service provider \( p_i \) by matching their corresponding attributes such
as input or output. Such a service assignment is denoted by a pair \((f_i, p_i)\). After each \(f_i\) of an abstract service composition is assigned a service provider \(p_i\), it becomes an executable service composition, and enters into the execution phase.

To provide a unified way to deal with both abstract and executable service composition, we have the following general definition of service compositions:

**Definition III.2. (Service Composition)** A service composition is modeled as a directed graph \(SC = \{S, T\}\), where \(S\) is a set of vertices representing services \(\{s_1, \ldots, s_n\}\) composed together to form a composite service and \(T\) is a set of directed edges representing topology relations \(\{t_1, \ldots, t_m\}\) between services. Different from service functions, each service is associated with attributes from both a service function and a service provider, which is denoted by \(s = \{a_1, \ldots, a_n, b_1, \ldots, b_0\}\).

The discussion in the rest of the paper is based on the above unified service composition model.

**B. Language Syntax and Semantics**

**Syntax.** The syntax of the proposed language is described in Table I. Everything in bold is a language keyword. Definitions are identified with angle brackets \(<>\), choices are enclosed in braces \(\{}\) and separated by \(\mid\). A *policy* is a combination of one or more rules that use the logical operators *And* and *Or*. A *rule* is the basic and core construct in the language. The *rule* construct consists of a *when* construct and a *satisfy* construct, both of which allow an arbitrary combination of a set of constraints using the logic operators *And* and *Or*. Each *constraint* is represented as a predicate with a list of arguments. An argument could be a service variable \(x_i\), an attribute of a service variable \(x_i, a_j\), or a constant. For practical reasons, we assume predicates can be evaluated efficiently.

Table II shows a policy example for an online shopping business using the above policy language. The policy contains only two rules with an *And* operator, which represent two requirements for service compositions running in the business. First, any “Checkout” service must be provided by a reputable service provider, for example “PayPal” or “Google”. Second, any two services that provide the same functionality should be offered by the same service provider.

**Semantics.** Policies defined in our policy language represent requirements for service compositions. When we bind a service composition with policies, variables in policies are mapped to services or attributes of services in the service composition. Variables are assigned values from the service composition. Constraints are evaluated to be *true* or *false* according to those values. Since both *when* and *satisfy* constructs in a rule are logic combinations of constraints, they can be represented as general logic formulas with a set of service variables. Given a rule: When \(F_1(x_1, \ldots, x_n)\) *Satisfy* \(F_2(x_1, \ldots, x_n)\), where \(F_1\) and \(F_2\) are the logical formulas in the when and the satisfy constructs respectively, the semantics of the rule is defined as the following first-order logic formula:

\[
\forall x_1 \forall x_2 \ldots \forall x_n F_1(x_1, \ldots, x_n) \rightarrow F_2(x_1, \ldots, x_n)
\]

The scope of each variable \(x_i\) is restricted within a rule. Each variable \(x_i\) can be bound with any service in a service composition so that the expression can be evaluated to be *true* or *false*. Let \(R: When F_1(\ldots) Satisfy F_2(\ldots)\) be a rule, and \(SC\) be a service composition. \(SC\) satisfies \(R\) if and only if \(F_1(x_1, \ldots, x_n) \rightarrow F_2(x_1, \ldots, x_n)\) is true for arbitrary binding of services in \(SC\) to variables \(x_1, \ldots, x_n\). Since a policy is a logic combination of rules, its evaluation is straightforward once rules are evaluated. \(SC\) satisfies a policy \(P\) if and only if \(P\) is evaluated to be *true*.

**C. Attribute-based Constraints**

Attributes of a service are composition-independent, which means that they will not change as the service joins different service compositions. In the context of service composition, constraints based on those attributes can be further divided into two categories: *local* and *global*. The *local* constraints only contains attributes from one service, and the *global* constraints involve attributes from multiple services for its evaluation.

The policy example shown in Table II contains both local and global attribute-based constraints. The first rule contains only local attribute-based constraints, which use only one service variable \(x_1\) and two attributes: “provider” and “type”, while the second rule contains two global constraints: \(x_1\) and “type” \(\equiv x_2\) and \(x_1\) provider \(\equiv x_2\) provider, each of which uses two service variables \(x_1\) and \(x_2\).
D. Topology-based Constraints

We add topology-based constraint support in our policy language. They are naturally defined using pre-defined topology-related predicates with services as arguments. To support topology-based constraint, we define a few basic topology-related predicates to represent the basic topology relations among services. The two major topology predicates are described as follows:

- **directLinkTo**(x₁, x₂) represents that a service x₁ directly connects to a service x₂. In a directed graph, it means there is a directed edge from x₁ to x₂.
- **hasPathTo**(x₁, x₂) represents that there is a path from a service x₁ to a service x₂, but x₁ may not directly connect to x₂.

Besides, we define **directLinkFrom** and **hasPathFrom** as the inverse of **directLinkTo** and **hasPathTo**. If **directLinkTo**(x₁, x₂) holds, then **directLinkFrom**(x₂, x₁) must also hold. The same can be applied to **hasPathFrom**. In addition, we define **isConnected** to represent the connectivity of two services. We also define the negation of each predicate to support the negation of constraints and further enhance the expressiveness of our policy language. Table III describes the details of the policy examples in Section II. For the first example, we use **directLinkTo** to indicate the information flow from one service to another service. For the second example, **hasPathTo** is used to identify the connectivity from service x₁ to service x₂. Besides security and privacy policies like these two examples, the topology-based constraints can be applied to other topology-related policies.

Note that other topology relations such as connectivity, loop, split (or branch), and join can be easily expressed using these two basic topology predicates. Table IV shows how to use the **directLinkTo** and **hasPathTo** predicates to represent other more complicated topology predicates.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Predicate Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>isConnected(x₁, x₂)</td>
<td>hasPathTo(x₁, x₂) \∧ \hasPathTo(x₂, x₁)</td>
</tr>
<tr>
<td>isLoop(x₁, x₂, x₃)</td>
<td>directLinkTo(x₁, x₂) \∧ \directLinkTo(x₂, x₃) \∧ \directLinkTo(x₃, x₁)</td>
</tr>
<tr>
<td>isSplit(x₁, {x₂, x₃})</td>
<td>directLinkTo(x₁, x₂) \∧ \directLinkTo(x₁, x₃)</td>
</tr>
<tr>
<td>isJoin(x₁, x₂, x₃)</td>
<td>directLinkTo(x₁, x₂) \∧ \directLinkTo(x₁, x₃)</td>
</tr>
</tbody>
</table>

TABLE IV

TOPOLOGY PREDICATE EXTENSION.

IV. ONTOLOGY-BASED POLICY FRAMEWORK

In this section, we present the detailed design of ontology-based policy framework for policy enforcement and analysis, which makes use of the Web Ontology Language (OWL) [11] together with its rule extension SWRL [12].

Figure 2 shows an overview of how the policy framework enforces policies for service compositions. In the figure, composition designers design service compositions to implement the business logics, and policy designers design policies according to regulations, security and privacy concerns of businesses. To enforce those policies, we first convert service compositions into facts of a service composition ontology defined in Section IV-A. Second, we map policies to SWRL rules. Finally, we combine the service composition ontology with SWRL rules and utilize off-the-shelf ontology reasoning engines to enforce policies. To ensure decidability for practical reasons, we restrict ourselves to DL-safe rules [15].

![Service Composition Ontology](image)

**Fig. 3.** Service Composition Ontology.

A. Service Composition Ontology

We create a service composition ontology using OWL in Figure 3, which serves as a common platform for policies and service compositions. The owl:Class **ServiceComposition** and **Service** define the concepts of service composition and service. The owl:ObjectProperty **hasService** describes the inclusion relation between **ServiceComposition** and **Service**. With these three constructs of OWL, we can define a service composition composed by a set of services, but we do not know the attributes of each service and the topology relations among services.

**Attribute Representation.** We consider each attribute of a service as a simple attribute with a basic data type such as Integer, Boolean, String and so on. Each attribute is defined as an OWL data property. Its domain is the owl:Class **Service**,
and rdfs:range specifies its data type. For example, in Figure 3, the owl:DataProperty type represents the service type. It belongs to owl:Class Service, and its data type is xsd:string.

**Topological Representation.** We define the owl:ObjectProperty directLinkTo to directly support directLinkTo topology predicate defined in our policy language. Both its rdfs:domain and rdfs:range are owl:Class Service. “A directLinkTo B” indicates a directed edge from service A to service B.

To further support other topology predicates such as directLinkFrom, hasPathTo, hasPathFrom and isConnected, we exploit functionalities provided by OWL to directly represent those topology predicates as OWL object properties and establish the logic implications among them. For example, hasPathTo is transitive, hasPathFrom and hasPathTo are inverse of each other, and hasPathTo is inherited from isConnected. The definition details of hasPathTo are shown in Figure 3.

**B. Mapping of Service Compositions**

Here, we discuss how to map a service composition to facts of our service composition ontology. The mapping process is straightforward. Figure 4 shows an example of mapping a simple BPEL process to service composition ontology.

![Fig. 4. Mapping Service Composition.](image)

In this example, we create a simple business process named “SampleProcess” using Eclipse BPEL plug-in. Each Invoke activity in the BPEL process is represented as a Service individual, for example A, B and C. directLinkTo is used to represent the execution flow from A to B and from B to C. The attributes of each Invoke activity are mapped to the corresponding OWL data properties, which are not shown in Figure 4 for simplicity.

Note that we do not need to generate other topology-related OWL object properties, such as directLinkFrom and hasPathTo, for service composition. They can be derived using existing knowledge in our service composition ontology. Thanks to this advantage, linear and non-linear (e.g., branches) topologies of service compositions make no big difference in terms of mapping service compositions and policy reasoning.

**C. Mapping of Rules and Policies**

With SWRL support, a policy can be mapped into service composition ontology as a set of SWRL rules. The mapping process is divided into three steps: policy normalization, rule mapping and policy mapping.

**Policy Normalization.** The normalization of a policy is to convert a policy into a normal form, which is defined in two levels: rule and policy. To convert a rule into a normal form, first we convert the set of constraints in when into a disjunctive normal form (DNF), and the set of constraints in satisfy into a conjunctive normal form (CNF). After this conversion, a rule is changed as follows:

When \((q_1) \lor \cdots \lor (q_w)\) Satisfy \((d_1) \land \cdots \land (d_s)\)

Note that \(q_i\) and \(d_j\) represent a conjunction and disjunction of constraints respectively. Second, without changing the semantics of the rule, we further decompose it into a conjunction of a set of rules that are in the normal form. They are described in the following:

When \((q_1)\) Satisfy \((d_1), \ldots, \) When \((q_1)\) Satisfy \((d_s)\)

When \((q_w)\) Satisfy \((d_1), \ldots, \) When \((q_w)\) Satisfy \((d_s)\)

A similar transformation process can be applied to a policy. A normalized policy is a disjunction of a set of rule sets \(\{r_{s1}, r_{s2}, \ldots\}\), and a rule set \(r_{si}\) is a conjunction of a set of normalized rules \(\{r_1, r_2, \ldots\}\). The rest of the discussion in the paper assumes that policies and rules are in the normal form.

**Mapping a rule.** When translating a rule to a SWRL rule, we observe that there is one difference between them. According to SWRL specification, the consequent of a SWRL rule is a conjunction of a set of atoms (constraints). However, the Satisfy construct as its corresponding part is a disjunction of a set of constraints. To translate a rule \(r\) when \((q_i)\) Satisfy \((d_j)\), where \(q_i = c_1 \land \cdots \land c_p\), \(d_j = c_{q+1} \lor \cdots \lor c_{q+d}\), and \(c_i\) represents a constraint, we design the following steps:

1. We define an OWL class for the rule, \(Invalid_{sc-r}\). \(Invalid_{sc-r}\) represents all service compositions that violate the rule \(r\).
2. We define the negation of \(r\), denoted by \(r_{neg}\) as follows:
   \[
   r_{neg} = \neg r = q_i \land \neg d_j = c_1 \land \cdots \land c_q \land \neg c_{q+1} \lor \cdots \lor \neg c_{q+d}
   \]
3. We generate a SWRL rule for the \(r_{neg}\) in the following form:
   \[
   Invalid_{sc-r} \leftarrow r_{neg}
   \]

Each rule is translated to a SWRL rule saying that any service composition that satisfies its antecedent \(r_{neg}\) violates the original rule.

**Mapping a policy.** To map a policy to SWRL rules, we introduce similar OWL classes for each rule set, each rule and the policy. The process of mapping a policy is as follows:
For service composition are enforced. InvalidServiceComposition of owl:Class is straightforward by executing a query to retrieve all instances. Any violation against any policy $p$ creates a SWRL rule for each policy relation between a policy and invalid service composition, we aims at detecting all invalid service compositions. A service InvalidServiceComposition fine an owl:Class rules, which are described in Manchester syntax.

In Table V, we provide the full translation of a policy, the The last rule indicates that any service composition that violates the policy. Note that for brevity we use a short form to represent SWRL rules in the above discussion, which omits variables and constraints in rules. In Table V, we provide the full translation of a policy, the conjunction of all rules introduced in Table II and III, to SWRL rules, which are described in Manchester syntax.

D. Policy Enforcement

The policy enforcement process in essence is a logic inference process based on the service composition ontology. It aims at detecting all invalid service compositions. A service composition that violates any single policy is invalid. We define an owl:Class InvalidServiceComposition, which represents the set of invalid service compositions. To infer the logic relation between a policy and invalid service composition, we create a SWRL rule for each policy $p_i$ as follows:

$$\text{InvalidServiceComposition} \leftarrow \text{Invalid}_i$$

Any violation against any policy $p_i$ will render a service composition invalid. To detect all invalid service compositions is straightforward by executing a query to retrieve all instances of owl:Class InvalidServiceComposition. In this way, policies for service composition are enforced.

Table V

<table>
<thead>
<tr>
<th>Full translation of the policy to SWRL rules in Manchester syntax.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Manchester Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Invalid}_{ac}\neg r_1(\text{sc}) \leftarrow \text{ServiceComposition}(\text{sc}), \text{hasService}(\text{sc}, \text{sc}_1), \text{provider}(\text{sc}_1, \text{p}_1), \text{type}(\text{sc}_1, \text{t}_1), \text{equal}(\text{t}_1, \text{“Checkout”), notEqual}(\text{p}_1, \text{“Google”), notEqual}(\text{p}_1, \text{“Paypal”)</td>
<td></td>
</tr>
<tr>
<td>$\text{Invalid}_{ac}\neg r_2(\text{sc}) \leftarrow \text{ServiceComposition}(\text{sc}), \text{hasService}(\text{sc}, \text{sc}_2), \text{provider}(\text{sc}_2, \text{p}_2), \text{type}(\text{sc}_2, \text{t}_2), \text{equal}(\text{t}_1, \text{t}_2), \text{notEqual}(\text{p}_1, \text{p}_2)</td>
<td></td>
</tr>
<tr>
<td>$\text{Invalid}_{ac}\neg r_3(\text{sc}) \leftarrow \text{ServiceComposition}(\text{sc}), \text{directLinkTo}(\text{sc}_1, \text{sc}_2), \text{hasService}(\text{sc}, \text{sc}_1), \text{hasService}(\text{sc}, \text{sc}_2), \text{securityLevel}(\text{sc}_1, \text{sl}_1), \text{securityLevel}(\text{sc}_2, \text{sl}_2), \text{greaterThan}(\text{sl}_1, \text{sl}_2)</td>
<td></td>
</tr>
<tr>
<td>$\text{Invalid}_{ac}\neg r_4(\text{sc}) \leftarrow \text{ServiceComposition}(\text{sc}), \text{hasPathTo}(\text{sc}_1, \text{sc}_2), \text{hasService}(\text{sc}, \text{sc}_1), \text{hasService}(\text{sc}, \text{sc}_2), \text{outputLevel}(\text{sc}_1, \text{tl}_1), \text{type}(\text{sc}_2, \text{t}_1), \text{equal}(\text{t}_1, \text{“Sensitive”), notEqual}(\text{t}_1, \text{“Secure”)</td>
<td></td>
</tr>
<tr>
<td>$\text{Invalid}<em>{ac}\neg r_1(\text{sc}) \leftarrow \text{Invalid}</em>{ac}\neg r_1(\text{sc})</td>
<td></td>
</tr>
<tr>
<td>$\text{Invalid}<em>{ac}\neg r_3(\text{sc}) \leftarrow \text{Invalid}</em>{ac}\neg r_3(\text{sc})</td>
<td></td>
</tr>
<tr>
<td>$\text{Invalid}<em>{ac}\neg r_4(\text{sc}) \leftarrow \text{Invalid}</em>{ac}\neg r_4(\text{sc})</td>
<td></td>
</tr>
</tbody>
</table>

The policy analysis services are aimed at helping policy designers identify potential issues and flaws in the policies. To this end, our ontology-based policy framework provides two major policy analysis services: policy conflict and redundancy. Conflict and Redundancy in Policies. Intuitively, any meaningful and effective policy divides the set of all possible service compositions into two sets of service compositions. One contains all service compositions that satisfy the policy, denoted by valid $\text{ac}$ - valid service composition space, and the other contains all service compositions that violate the policy, denoted by invalid $\text{ac}$ - invalid service composition space. It indicates a potential issue if either of them is empty. Due to the space limit, we focus on the discussion of conflicts and redundancy in a rule. A similar analysis can be applied to rule set or policies. Conflicts occur when the valid $\text{ac}$ of a rule is empty, which means that no service composition can satisfy the requirements defined in the rule. Redundancy may occur in two cases. First, when the invalid $\text{ac}$ of a rule is empty, which means that the rule does not rule out any service composition. Second, if a rule is removed and the service composition space represented by policies does not change, then the rule is redundant. Reduction to OWL Expressions. To achieve the detection of conflicts and redundancy in policies, we resort to the reasoning capabilities provided by OWL reasoning system. To this end, we need to translate our rules and policies to OWL constructs such as classes. Note that in Section IV-C, we already discussed how to convert our rules and policies to SWRL rules. The translation between SWRL rules to OWL constructs has been addressed in [16]. However, limitations on such translations do exist due to the expressive restrictions of SWRL and OWL, which make OWL reasoning decidable. The restrictions include but not limited to the tree model property of OWL constructs [17], only unary and binary predicates, and no existential appearing within the head of rules. We restrict the policy analysis services within the subset of SWRL rules and OWL ontology language, which is defined in [16].
Once we convert rules and policies to SWRL rules according to the mechanisms described in Section IV-C, each SWRL rule $r_i$ is translated to an OWL class $R_i$, where $R_i$ represents an invalid $r_i$. The logic combinations of rules in a policy are translated to the intersection or union of OWL classes that represent rules. The example translation of rules in Table V is shown as follows:

$$R_1, R_2, R_3, R_4$$

$$RS_1 \equiv R_1 \sqcup R_2 \sqcup R_3 \sqcup R_4$$

$$P \equiv RS_1$$

Each of them represents its own invalid service composition space, which will be used in the following.

Detection of Conflict and Redundancy. On the one hand, to check if any conflict exists in a rule $r_i$, instead of checking the satisfiability of the OWL rule class $R_i$, we need to see if $ServiceComposition \subseteq R_i$ can be derived. If so, it means that no service composition can satisfy the requirements defined in the rule $r_i$. The same method can be applied to a policy to check if any conflict exists in a policy.

On the other hand, redundancy may occur in two cases as mentioned above. In the first case, we check the satisfiability of each rule class $R_i$. If $R_i$ cannot be satisfied, then it does not rule out any invalid service composition. Thus, it is redundant. In the second case, to check if a rule is redundant, we simply remove the rule and generate a new policy class $P_{new}$, and then check if $P$ is subsumed by $P_{new}$ and $P_{new}$ is subsumed by $P$. If they both can be derived, it means that $P_{new}$ is equivalent to $P$. Therefore, the removed rule is redundant. If multiple policies exist, the same conflict and redundancy detection algorithms can be applied to policies.

VI. EXPERIMENTAL EVALUATION

Prototype Implementation. We have implemented a prototype policy management system based on Eclipse [14]. The architecture design of our prototype mainly consists of two parts: user interfaces and the ontology-based policy framework. In the prototype, we design two user interfaces, which are Eclipse BPEL plug-in with policy extension and Eclipse Policy plug-in, for composition designers and policy designers. Our policy framework employs an open source OWL reasoner Pellet [13] as its ontology reasoning engine.

Prototype Evaluation. We conducted extensive experiments to evaluate the performance of our approach, compared the performance of the evaluations of attribute-based and topology-based constraints, and investigated how the number of rules and services in a service composition influences the performance of OWL reasoning engine. As mentioned in Section IV-B, linear and non-linear topologies of service compositions make no big difference in terms of mapping service compositions and policy reasoning. Thus, we only generate service compositions with a linear topology for our experiments. We measure the performance according to response time, which is the time to load service composition ontology, add policies and a service composition to it, and check if the service composition is valid.

Figure 5 shows the response time versus the number of rules with different numbers of services in a service composition. The result shows that the response time goes up linearly as the number of service compositions increases, and the time cost of local attribute-based constraints is small. The time of evaluating global attribute-based constraints is shown in Figure 6, where we use different numbers of services in a service composition for experiments. It is no doubt that it is much slower to evaluate global constraints than local constraints. The result also shows that the cost increases dramatically when the number of services in a service composition is doubled, which is because the number of evaluations that need to be done for a global constraint increases dramatically with the increase of the number of services in a service composition.

To evaluate topology-based constraints, we did a performance comparison between the evaluation of attribute- and topology-based constraints, which is shown in Figure 7, given the number of services in a service composition is 10. The result shows that the performance difference between local attribute constraint and directLinkTo topology constraints is small, which is mainly because the directLinkTo is explicitly expressed in ontology and no new knowledge needs to be inferred for it. However, the hasPathTo is much slower than the directLinkTo because the hasPathTo has a transitive property, which may result in dramatic increase in terms of the number of pairs that have hasPathTo topology relation. It is also the reason why the number of services in a service composition greatly influences the response time of evaluating hasPathTo topology relation, which is shown in Figure 8.

VII. RELATED WORK

Service composition has become an efficient and cost-effective way to compose existing services to fulfill complicated tasks. It recently has received a great amount of attention [3]–[5]. Carminati et al. [5] focused on security constraints of web service composition. Chun et al. [3] used not only the syntactic and semantic compatibilities of service, but also uses the policies and rules for discovering and finding compatibility to generate sensible service compositions. Medjahed et al. [4] defined context-based policies using WS-Policy [18] for web service composition. To the best of our knowledge, none of the existing work takes the internal topology of service composition into consideration when constructing service composition related policies. In this paper, the proposed policy language explicitly supports the topology relations among services. With the support of topology relations, it makes the policy language more expressive for users to define their policies for service compositions.

The techniques such as ontology representation and rule-based reasoning adopted in this paper have been widely used in recent work [19]–[21]. Lamparter et al. [19] used the Web Ontology Language OWL [11] together with its rule extension SWRL [12] to implement their service selection framework. With ontology and rules support, it is able to perform sophisticated matchmaking and ranking of services by means of logical inferencing. With OWL, Kolovski et al. [20] used the
framework to extend access control policies with ontology-based descriptions and utilize the ontology logic reasoning to provide different kinds of policy analysis services such policy comparison, policy verification and policy redundancy. Different from previous work, we use OWL and SWRL as the basis of our policy framework for effective policy enforcement and analysis for service compositions.

VIII. CONCLUSION

In this paper, we have introduced topology-based constraints for service composition, and presented a rule-based constraint policy language, which provides a unified way to support both attribute- and topology-based constraints. We have implemented an ontology-based policy framework for effective policy enforcement and analysis. To the best of our knowledge, our work makes the first attempt to take the topology relations among services into consideration when establishing and enforcing policies for service compositions. We have implemented a prototype based on Eclipse and an open source OWL reasoner Pellet [13]. Finally, we demonstrated through experimental evaluation that our policy framework is practical in enforcing policies for service composition.

In future, we intend to add topology semantics support for topology-based constraints, extend exiting policy languages like WS-Policy [18] to support our rules and topology-based constraints to specify policies for composite services, and further investigate policy redundancy analysis based on rule-rewriting techniques.

ACKNOWLEDGMENT

The authors would like to thank Zach Jorgensen, Juan Du and the anonymous reviewers for their helpful suggestions. This work is supported by the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and by the NSF under grant IIS-0430166. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government.

REFERENCES