Compatibility Checking of Heterogeneous Web Service Policies Using VDM++

Quan Z. Sheng[1], Jian Yu[1], Zakaria Maamar[2], Wei Jiang[1], and Xitong Li[3]

[1]The University of Adelaide
Adelaide, Australia
qsheng@cs.adelaide.edu.au

[2]Zayed University
Dubai, U.A.E
zakaria.maamar@zu.ac.ae

[3]Tsinghua University
Beijing, China
lxt04@mails.tsinghua.edu.cn

Abstract

Web service policies capture the capabilities and requirements of Web services from both functional and non-functional perspectives. Policies of a Web service govern and ensure the runtime consistency of the service, i.e., people or services interacting with this service are only allowed to perform legitimate actions. When composing Web services, policies of the participated Web services have to be compatible in order to make sensible compositions. Unfortunately, due to heterogeneity of policy specification languages, it is difficult to compare policies of different Web services directly. In this paper, we propose an approach for compatibility checking of Web service policies specified in difference languages. In particular, our approach applies the model-oriented specification from the Vienna Development Method (VDM++). An executable formal model of policy languages is represented in VDM++ and different policies are then translated to this VDM++ model for compatibility checking. Our approach has been validated by a prototype with different Web service policy languages such as WSPL and WS-Policy.

1. Introduction

Web services are becoming the de facto technology for developing e-Business applications [15]. By encapsulating key business logics as remotely interoperable Web services, value-added services can be conveniently created by composing multiple Web services with workflow languages such as WS-BPEL [9]. However, at runtime, whether two component services in a composite service can interact with each other is constrained by the policies asserted on them. For example, an invocation to a service may be disallowed for outside-of-regular-working-hours restriction.

Web service policies capture the capabilities and requirements of Web services from both functional and non-functional perspectives. A policy can be defined based on functional requirements: for example, a hotel booking service may require that a confirmation must be acknowledged within 5 working days. A policy can also reflect the non-functional aspect of a service: for example, to ensure the privacy of customers, a policy of the same hotel booking service may state that after checking out, a customer's name can only be kept on the server for less than two weeks.

Compatibility is the prerequisite for two Web services to interact during runtime. Compatibility checking ensures that any policy entry of one Web service does not conflict with the policy of its interacting Web service. It is straitforward to compare two policies if they are specified in the same language. Unfortunately, many policy specification languages are currently available and different service providers may prefer a particular language to specify service policies. For example, Web Services Policy Language (WSPL) [13] maintained by OASIS and WS-Policy [14] maintained by W3C are two overlapping languages for specifying Web service policies. As a result, the heterogeneity of Web service policies makes automatic policy compatibility checking a difficult task to perform.

In this paper, we propose an approach to check the compatibility of Web service policies specified in different languages. We adopt a model-oriented specification from the *Vienna Development Method* (VDM++) [6]. In particular, we develop an executable formal model of policy languages, which is represented in VDM++. Different policies are then translated to this VDM++ model for compatibility checking. We solve the compatibility checking issue by calculating the containness between VDM++ models. To validate our approach, we have implemented a prototype that can translate WSPL and WS-Policy specifications to VDM++ for compatibility checking.

The rest of this paper is organized as follows: Section 2 briefly overviews VDM++ and the two example Web service policy specification languages: WSPL and WS-Policy. Section 3 presents the architecture, design and implementation of the compatibility checking framework. Finally, Section 4 overviews the related work and Section 5 concludes the paper.

978-0-7695-3708-5/09 $25.00 © 2009 IEEE
DOI 10.1109/SERVICES-I.2009.87

2 Preliminaries

This section gives a brief overview to the Web service policy specification languages, including WSPL and WS-Policy, and the VDM++ formal model.

2.1 WSPL

WSPL is a subset of *OASIS eXtensible Access Control Markup Language* (XACML), which is also known as *XACML profile for Web-services*. WSPL is an XML language that has three top-level elements: `PolicySet`, `Policy`, and `Rule`. `PolicySet` is the container for policies and each policy is a sequence of one or more rules. Rules are listed in order of preference, with the most preferred choice listed first [2]. Using the *Apply* element, each WSPL rule defines a constraint that the service needs to abide. Predefined constraint operators include: *equals*, *greater than*, *greater than or equal to*, *less than*, *less than or equal to*, *set-equals*, and *subset*. Figure 1 is a snippet of WSPL code specifying that `ActiveConnection` (line 10) should be less than 100 (line 12).

```
01  <PolicySet PolicySetId="airlinePS"
02      PolicyCombiningAlgId="deny-overrides">
03  ...
04    <Policy PolicyId="policyID:1" RuleCombiningAlgId="permit-overrides">
05      <Rule RuleId="ruleid:1"Effect="Permit">
06        <Condition FunctionId="function:and">
07          <Apply FunctionId="function:integer-less-than">
08            <Apply FunctionId="function:integer-one-and-only">
09              <SubjectAttributeDesignator DataType="integer"
10              AttributeId="ActiveConnection"/>
11            </Apply>
12            <AttributeValue DataType="integer">100</AttributeValue>
13          </Apply>
14        </Condition>
15      </Rule>
16      <Rule RuleId="ruleid:2" Effect="Permit">
17        ...
18  </Rule></Policy></PolicySet>
```

Figure 1. WSPL example

2.2. WS-Policy

WS-Policy has four elements: `Policy`, `All`, `ExactlyOne` and `PolicyReference`, and two attributes: `wsp:Optional` and `wsp:Ignorable`.

`Policy` is the container for policy assertions/constraints. `All` and `ExactlyOne` are two policy operators for combining policy constraints. Combining policy assertions using `All` operator means that all the behaviors represented by these assertions are required. Similarly, combining policy assertions using the `ExactlyOne` operator means that exactly one of the behaviors represented by the assertions is required. Policy operators can be mixed to represent different combinations of behaviors. To support reuse, a policy assertion can be named and then be referenced using `PolicyReference`.

A policy assertion is marked as optional using the `wsp:Optional` attribute. If a policy assertion is `wsp:Ignorable`, this policy assertion is not related to the requester and can be *ignored* by the requester. Figure 2 is a WS-Policy example stating the same policy as Figure 1 (i.e., `ActiveConnection` is less than 100).

```
01  <PolicySet PolicySetId="airlinePS" PolicyCombiningAlgId="deny-overrides">
02  ...
03    <Policy PolicyId="policyID:1" RuleCombiningAlgId="permit-overrides">
04      <Rule RuleId="ruleid:1" Effect="Permit">
05        <Condition FunctionId="function:and">
06          <Apply FunctionId="function:boolean-equal">
07            <Apply FunctionId="function:boolean-one-and-only">
08              <SubjectAttributeDesignator DataType="integer"
09                AttributeId="ActiveConnection"/>
10            </Apply>
11            <AttributeValue DataType="integer">100</AttributeValue>
12            </Apply></Condition></Rule>
13      <Rule RuleId="ruleid:2" Effect="Permit">
14        ...
15  </Rule></Policy></PolicySet>
```

Figure 2. WS-Policy example

2.3. VDM++

VDM++ is the object-oriented extension of the Vienna Development Method (VDM) specification language [8]. VDM++ models are composed of a series of class definitions, and each of them may contain the instance variables and other definitions. The ordinary simple type definitions are given in terms of basic data type in VDM++. The composite data types are constructed by using compound types in VDM++, such as Set Types, Sequence Types, Product Types. Some high abstract types are included in VDM++, such as token types. The methods or interfaces in each class are represented as the function definitions and operation definitions [3] .

Several reasons make us to choose VDM++ as the formal model for compatibility checking of Web service specification languages. Firstly, the rich data types and operations of VDM++ facilitate the mapping of WSPL or WS-Policy. Secondly, VDM++ has been successfully used in checking the consistency of XACML access control policies [3], which provides a good reference for the mapping of WSPL to it. Last but not the least, VDM++ has a strong tool support. The VDMTOOLs tool set [1] includes a syntax and type checker, an interpreter for executable models, test scripting and coverage analysis facilities, a platform code generation, and pretty-printing.

3 The Compatibility Checking Framework

3.1. Architecture

Figure 3 shows the architecture of our Web services policy compatibility checking framework, consisting of two main components: the *Translator* and the *Checker*. The

Translator is responsible for translating policies in different languages into the VDM++ policy model, while the Checker compares policy documents in VDM++ on their compatibility. Our policy checking approach is based on the following principle:

- *If the merging of two policy documents does not produce an empty document, then they are compatible.*

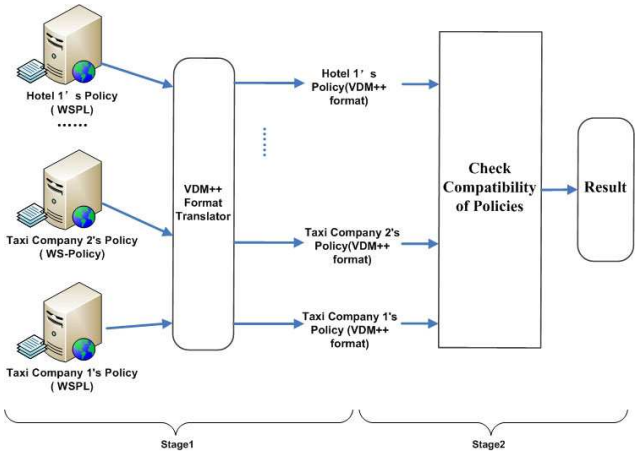

Figure 3. Framework architecture

Formally, supposing *Rules* are the atomic elements of a policy document and every policy document can be represented by a set of Rules: $Policy_i = \{R_{i1}, R_{i2}, .., R_{in}\}$, then the merging of two policy documents can be formulated as: $Merge(Policy_i, Policy_j) = \sqcap(Policy_i \times Policy_j)$, where the intersection operation '$\sqcap$' is defined as: $\sqcap(X \times Y) = \{x \cap y : x \in X, y \in Y\}$.

Figure 4 gives an example of policy merging: both $Policy_1$ and $Policy_2$ have two rules, and the $Combined\_Policy$ set is the Descartian product of the two policies. If we apply the intersection operation on the $Combined\_Policy$, the result is a policy document containing two rules where the empty rule is not shown.

3.2 Translating Policies to VDM++

To translate a Web service policy document to VDM++, the first thing we need to do is to design a VDM++ policy model. The UML class diagram in Figure 5 is the structure of the VDM++ model which is associated with the structure of policy specification languages. The `MappingStructure` class describes the policy mapping structure. The `Condition` class describes the concrete predicates. The `TypeMapping` class describes the simple data types and values. In the sequel, we first introduce the policy model in detail which is related to `MappingStructure`, and then briefly introduce the other two classes.

Policy Model. In the VDM++ model, objects from the `MappingStructure` class contain a single instance variable `policyRef` of type `PolicySet`. Referring to the code in Figure 6, a `PolicySet` type contains a `Target`, a sequence of `Policy`, and a `Policy_combining_algorithm`. A `Policy` contains an optional `Target`, a sequence of `Rule`, and a `Rule_combining_algorithm`. The `Rule` contains an optional `Target`, an `Effect`, and a reference to `Condition` which will be introduced later. The `Target` defines three sets of strings for recording its child elements. The `Effect` indicates the action when the rule is satisfied. It can be assigned with one of the enumerated values. In accordance with [14], we only consider one possibility: `Permit`. Similarly, the `Policy_combining_algorithm` and the `Rule_combining_algorithm` indicate the combination algorithms and are all enumeration type. Currently we only consider two types of combination algorithms. However, there is no difficulty in adding additional ones to the model. The additional algorithms can be added by enumerating them in the definitions of `Policy_combining_algorithm` and `Rule_combining_algorithm`.

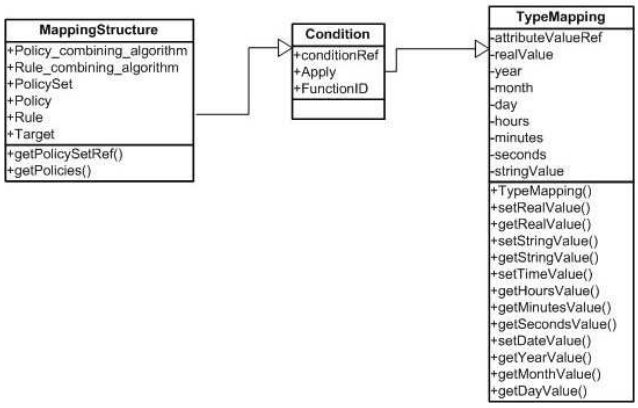

Figure 5. Overview of the VDM++ model

Now we discuss how to translate Web service policy languages into our VDM++ policy model. In this paper, we take two typical Web service policy languages as examples, namely WSPL and WS-Policy[1]. For WSPL, this VDM++ model can be perfectly mapped to the corresponding WSPL structure: the `PolicySet` type is equivalent to the `PolicySet` element in WSPL; the `Policy` type represents the `Policy` element; and the `Rule` is mapped to the `Rule` element. The `Target` element in WSPL contains three types of child elements and each of them can be translated to a string which keeps its identity, attributes,

[1]The approach is generic and not limited to these two policy languages.

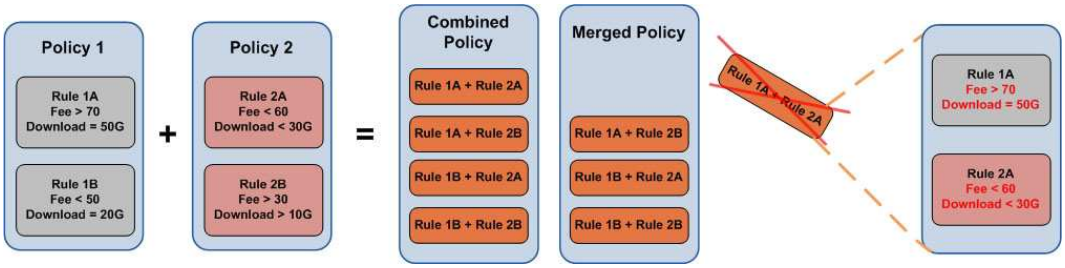

Figure 4. Policy merging

```
public PolicySet::target:Target
                components:seq of Policy
                policyComAlg: Policy_combining_algorithm;
public Policy::target:Target
                rules: seq of Rule
                ruleComAlg: Rule_combining_algorithm;
public Rule::target:Target
               effect: Effect
               condition: ConditionStructure;
public Target::subjects: set of String
                resources: set of String
                actions: set of String;
public Effect=<Permit>;
public Policy_combining_algorithm=<denyoverrides>;
public Rule_combining_algorithm=<permitoverrides>;
...
```

Figure 6. VDM++ model definition

features and related values. These strings are mapped as the `Target` type in the VDM++ model. However, the entire VDM++ model makes one modification: it does not construct the corresponding mapping structure for the second level `PolicySet` element of WSPL. There are two reasons that we made such modification: first, there are no related rules in WSPL indicating the combining algorithms for the second level `PolicySet` element; second, the Web service policy document having second-level `PolicySet` elements can be translated to another document which contains no second-level `PolicySet`. So our approach only consider WSPL policy documents without second-level `PolicySet` elements. Moreover, [13] specifies the combining algorithm for `Policy` elements is `Deny-overrides` and the combining algorithm for `Rule` elements is `Permit-overrides`. Therefore, they are mapped as `Policy_combining_algorithm` and `Rule_combining_algorithm` type respectively in the VDM++ model.

For WS-Policy, the wrapper `Policy` element is mapped as the `Policy` type in the VDM++ model because the wrapper `Policy` element contains only one child element, i.e. `ExactlyOne` operator element. The `ExactlyOne` element has the similar semantics as the Permit-overrides algorithm in WSPL. So the child `All` elements are mapped to the elements of type `Rule` in the VDM++ model. Considering the uniform format for policy compatibility checking, we need to create the element of type `PolicySet` to record a set of elements of type `Policy` and other information.

Conditions. In the VDM++ structure model, the condition class can be considered as a collection of predicates. An instance object from the `Condition` class contains a single instance variable `conditionRef` of type `ConditionStructure`. As shown in Figure 7, a `ConditionStructure` contains a `FunctionID` and a sequence of `Apply`. A `FunctionID` is an enumeration type which is used to indicate the combined algorithm for a sequence of predicates. The `Apply` represents a concrete predicate. It has four components. The first three components are objects of type `String` used to record the datatype, possibly used comparison function, and attributes respectively, and `TypeMapping` is used to record the type and actual value of a specific predicate.

```
...
public ConditionStructure::functionID:FunctionID
                components: seq of Apply;

public Apply::dataType: String
                functionType: String
                attributeID: String;
                attributeValue: TypeMapping;
public Target::subjects: set of String
                resources: set of String
                actions: set of String;

public FunctionID=<AND>;
...
```

Figure 7. Condition class definition

For WSPL, a single predicate is represented by the `Apply` element, therefore a set of `Apply` elements maps to the condition class and a single `Apply` element maps to an `Apply` in the VDM++ model. For WS-Policy, each assertion represents a specific action or a constraint condition that has similar functional effects as the predicate. Thus, a single WS-Policy assertion, which is also a child assertion of a `All` element, maps to an `Apply`.

TypeMapping. Objects from the `TypeMapping` class are used to record the concrete attributes and values of a specific predicate or assertion. It defines the corresponding

datatype mapping between simple datatypes of the two policy specification languages and VDM++. It also provides a group of interfaces for setting the values of each datatype and another group of interfaces for accessing these values in the VDM++ model.

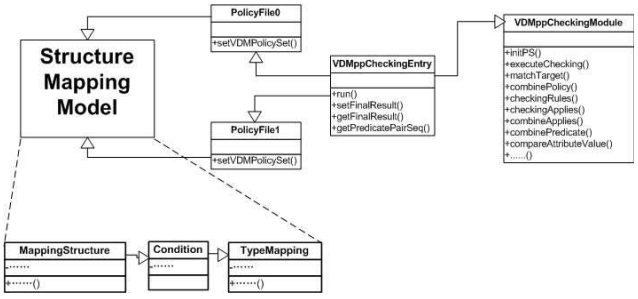

Figure 8. Compatibility checking class diagram

3.3 Compatibility Checking

An overview of the compatibility checking module is shown in Figure 8. There are four VDM++ based policy classes: `PolicyFile0`, `PolicyFile1`, `VDMppCheckingEntry`, and `VDMppCheckingModule`. `VDMppCheckingEntry` is the entry of the policy checking module and `VDMppCheckingEntry` is the core checking class. First of all, the objects from the `VDMppCheckingEntry` class initialize the instance policies from `PolicyFile0` and `PolicyFile1`, and then it invokes the method of `VDMppCheckingModule` to check the compatibility of the two policies. Details of these classes are discusses as follows.

- **`PolicyFile0` and `PolicyFile1`**. These two classes have the same structure, but the corresponding values for the variables are different. Note that the VDM++ based policy files are automatically generated according to the pre-defined mapping model.

  Figure 9 shows the main method of the VDM++ based policy class definition. The `setVDMPolicySet` method defines how to create the VDM++ policy which is based on the corresponding Web service policy. The code between ① and ② is used to construct two instance members (i.e., p0r0a0 and p0r0a1) of type `Apply` and assign the corresponding values to them. The code at ② indicates that an instance of type `Rule`, i.e., p0r0, is created by the definition of the previous members (i.e., p0r0a0 and p0r0a1) of type `Apply`. The code between ③ and ④ is another process for generating instance of type `Rule` (i.e., p0r1) according to the original policy. The code at ④ illustrates using instances of type `Rule`, including p0r0 and p0r1, to construct the object p0 from type Policy. ⑤ shows how to use the `Policy` instance to create a `PolicySet` instance. String from ⑥, the members of type `Apply` invoke the methods provided by the VDM++ mapping model to set the actual values for the predicates or assertion.

- **`VDMppCheckingEntry`**. After the VDM++-based policy files are created, the `run` operation of this class is invoked as the entry of the entire policy checking module. This operation first initializes the instances for the two VDM++ policy classes, and then creates an object of `VDMppCheckingModule` class and invokes the object's method to do the checking. Referring to the compatibility principle defined in Section 3.1, if the checking result contains a non-empty policy which is the merging of the two input policies, then the two input policies are compatible; otherwise, the two input policies are not compatible.

- **`VDMppCheckingModule`**. This class is the core of the policy checking process. It checks whether the two policies are comparable, and then use the combining algorithm to create the merged policy. The policy compatibility checking procedure runs from the outer-most elements to the inner-most elements. This procedure can be divided into four steps:

  - The first step is to combine coincident `PolicySet` objects. The `PolicySet` objects are coincident if and only if their `Target` components are matching. So the elements from the type `Target` need to be checked first before any other checking steps.
  - The second step is to combine coincident `Policy` objects. Two `Policy` components are coincident if and only if their `Target` components are matching.
  - The third step is to combine coincident `Rule` components within the combined `Policy` components. In order to combine `Rule` components, the combining procedure for `Apply` components are involved.
  - The last step is to combine coincident `Apply` components. Two of `Apply` members are coincident if and only if they have the same `attributeID` and `dataType` attribute.

3.4 Implementation

We use Java to implement the translation from WSPL and WS-Policy to VDM++. Especially, Sun's XACML im-

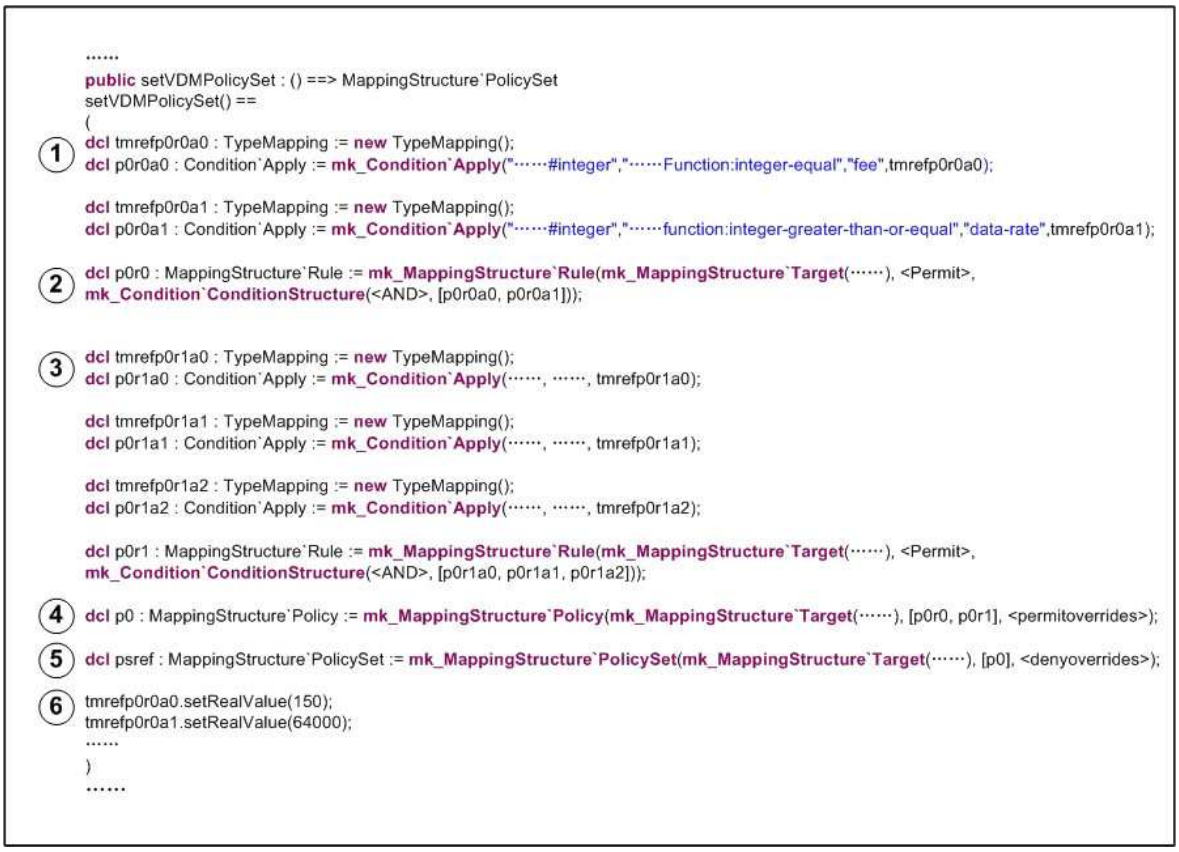

Figure 9. Code snippet of a VDM++ policy file

plementation[2] is used to parse WSPL, and Apache Neethi framework[3] is used to parse WS-Policy. As for the compatibility checking, we use VDM Toolbox[4] as the integrated developing and executing environment.

A hotel-booking scenario is used to showcase and validate our implementation. Supposing Jim wants to book a hotel and his specific policy/requirements for the hotel is: i) the rate is not less than 4-star, and ii) the price of a single room less than $70 per night or a double room less than $120 per night. Suppose that there are three hotels with different policies:

- Hotel A: The rate is 3-star; Single is $40 and double is $75.
- Hotel B: The rate is 4-star; Single is $75 and double is $110.
- Hotel C: The rate is 4-star; Single is $60 and double is $100.

To highlight language heterogeneity, the policy of Hotel A and Hotel C is written in WSPL, and the policy of Hotel B is in WS-Pollicy. In Figure 10, we show Jim's hotel policy (the left-top panel, in WSPL), Hotel B's policy (the left-bottom panel, in WS-Policy), and the compatibility checking result (the right panel). The checking result shows that both Hotel B and Hotel C are compatible with Jim's policy, but not Hotel A since its rate is lower than the requirements. It should be noted that we use this simple example for illustration purposes only. Our approach is sufficient to process complex policies.

4 Related Work

A number of papers have been published to introduce policies in Web services as well as how these policies can be applied in a Web service composition.

In [10], the authors focus on the consistency verification between BPEL process and privacy policy. In the paper, the authors introduce a formal framework that the business operations are checked by the privacy policies, and then make a decision whether the operations conform to these policies. Their study limits in privacy policy compatibility and actually discusses a special aspect of compatibility, i.e. how many business operations can meet the privacy requirements. In [7], the authors do not directly discuss Web service policies, and the alternative terminology for policy is

[2] http://sunxacml.sourceforge.net/
[3] http://ws.apache.org/commons/neethi/
[4] http://www.vdmtools.jp/en/

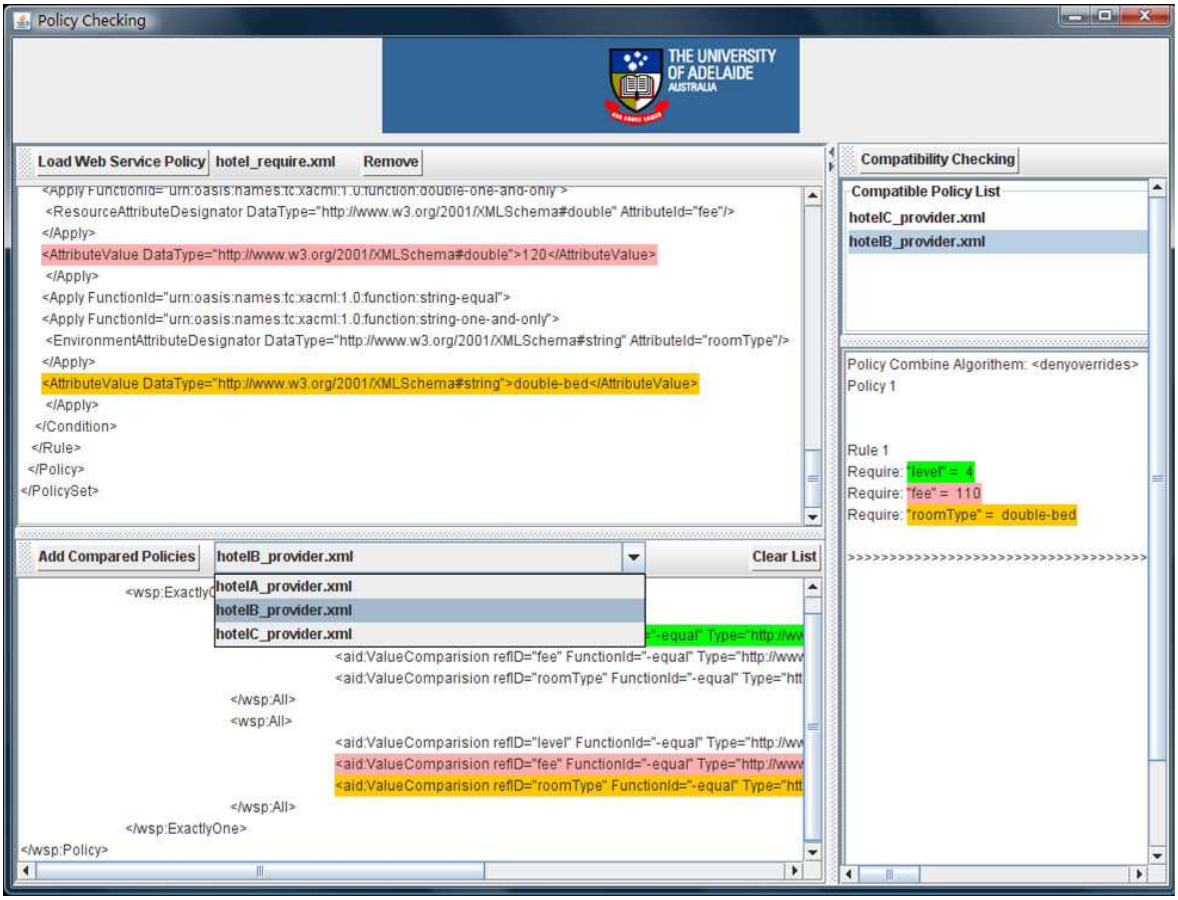

Figure 10. Hotel policy checking results

regulation. The real-time constraints that many regulations contain, such as data retention duration and the obligation to respond the requests within a certain time, are the concrete examples for applying the regulations on the Web services. This paper also introduces Regulations Expressed as Logical Models (REALM) as a language for modeling the regulatory requirements. In [11], Maamar et al. discuss the compatibility problem between policies of Web services. The authors argue that the compatibility problem could prevent Web services automatic composition, if the problem can not be properly addressed. They introduce three types of policies: *business policy*, *behaviour policy*, and *privacy policy*. Compatibility checking between all these types of policies relies on the notion of violation in the sense that policies of different Web services are considered compatible if there is no violation detected. Generally, violation is related to constraint satisfaction. In [4], the authors introduce policy-based Web Service composition. They define three types of policies: *user policy*, *service flow policy* and *service policy*. Compatibility between these types of policies contains two levels: the syntactic level which is involved in syntactic check and the semantic level which is used for semantic compatibility check. The various policies and service thematic ontology are described by OWL, DAML-S, RuleML and RDF standards. They propose an approach for building an agent for automatic Web service composition based on these policies. They also define how to check syntactic, semantic and policy compatibility during composite process. However, their work ignores the issue on how to handle conflicting policies described by different organizations and how to resolve the problem of policy language heterogeneity. The semantic issue addressed in their research area is complementary to that addressed in our project.

It should be noted that most of the research works currently focus on how to check the different types of policies, such as privacy and security aspects or attempt to resolve semantic issue during Web services composition. Very little work is being done on policy compatibility checking within the situation of the policy language heterogeneity. However, the language heterogeneity is an extremely complex issue and it directly affects compatibility checking result of Web services policies in the real world. Therefore the issue should not be ignored when we discuss policy compatibility checking. In [5], Finkelstein et al. indicate that a Web service composition may consist of heterogeneous services, and it is difficult to check consistency between poli-

cies because many different languages can be used to describe policies. Moreover, some implementations of composite Web services are likely to be too low level to allow compatibility checking. Therefore they propose another approach to describe compositions by using WS-CDL and use CLiX language to represent policies. The xlinkit [12], a checking engine, is used to check consistency of CLiX-based policies. The technologies adopted in their approach have very limited supports. Furthermore, although they mention the heterogeneity issue of policy languages in a Web service composition, the proposed solution does not actually take it into account. Another reach effort is presented in [3]. Bryans et al. focus on access control policies expressed by the eXtensible Access Control Markup Language (XACML). They adopt VDM++ to present the executable formal model of XACML access control. The aim is to use formal approaches to support policy analysis and checking. Policy checking can benefit from using formal techniques because of the breadth and rigour of analysis that they afford. The authors provide an implemented model for policy compatibility checking and the mapping between XACML and VDM++. The VDM++ supported tool can directly execute, test and analyze the model and provide a proof to the consistency and completeness of the model. However, the model is only suitable for subset of XACML and could not be directly used to check XACML policies. To execute the checking model, users have to manually translate the XACML-based policy file into VDM++ based test file in advance.

5 Conclusion

In this paper, we have proposed an approach for checking the compatibility of Web service policies specified in difference languages, especially in WSPL and WS-Policy. Our approach applies the model-oriented specification from the Vienna Development Method (VDM++). An executable formal model of policy languages is represented in VDM++ and different policies are then translated to this VDM++ model for compatibility checking. The proposed approach can be used as a stand alone system for policy compatibility checking. It is also easy to be integrated into any Web service composition framework. Our ongoing work includes extending the system to support more policy languages. Another interesting extension to the work is to provide semantic support for compatibility checking.

References

[1] The VDMTools. http://www.vdmtools.jp/en/.

[2] A. H. Anderson. An Introduction to the Web Services Policy Language (WSPL). In *Proc. of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, New York, USA, 2004.

[3] J. W. Bryans and J. S. itzgerald. Formal Engineering of XACML Access Control Policies in VDM++. In *Proc. of the 9th International Conference on Formal Engineering Methods (ICFEM)*, LNCS 4789, pages 37–56, Boca Raton, FL, USA, 2007. Springer.

[4] S. A. Chun, V. Atluri, and N. R. Adam. Policy-Based Web Service Composition. In *Proc. of the 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)*, 2004.

[5] A. Dingwall-Smith and A. Finkelstein. Checking Complex Compositions of web services against policy constraints. In *Proc. of the Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS)*, 2007.

[6] E. Dürr and J. van Katwijk. VDM++: A Formal Specification Language for Object-oriented Designs. In *Proc. of the 7th International Conference on Technology of Object-Oriented Languages and Systems*, pages 63–77, Dortmund, Germany, 1992.

[7] C. Giblin, A. Y. Liu, S. Mueller, B. Pfitzmann, and X. Zhou. Regulations Expressed As Logical Models (REALM). Technical report, IBM Research Division, 2005. RZ 3616.

[8] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Englewood Cliffs, 1990.

[9] D. Jordan and J. Evdemon et al. Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.

[10] Y. Li, H. Y. Paik, B. Benatallah, and S. Benbernou. Formal Consistency Verification between BPEL Process and Privacy Policy. In *Proc. of the International Conference on Privacy , Security, and Trust (PST2006)*, Toronto, Canada, 2006.

[11] Z. Maamar, Q. Z. Sheng, H. Yahyaoui, D. Benslimane, and F. Liu. On Checking the Compatibility of Web services' Policies. In *Proc. of the 8th international Conference on Parallel and Distributed Computing, Applications and Technologies*, Adelaide, Australia, 2007.

[12] C. Mentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.

[13] T. Moses, A. Anderson, S. Proctor, and S. godik. XACML profile for Web-services. http://www.oasis-open.org/committees/download.php/3661/draft-xacml-wsp-04.pdf, 2003.

[14] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and Ümit Yalçinalp. Web Services Policy 1.5 - Framework. http://www.w3.org/TR/ws-policy/.

[15] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed. Deploying and Managing Web Services: Issues, Solutions, and Directions. *The VLDB Journal*, 17(3):537–572, 2008.