A HIGH ABSTRACTION LEVEL APPROACH FOR DETECTING FEATURE INTERACTIONS IN WEB SERVICES

Ahmed Khoumsi1, Zohair Chentouf and Safwan Qasem

Computer Science Department, College of Computer and Information Sciences
King Saud University, Riyadh, Kingdom of Saudi Arabia

ABSTRACT

Composing Web services is prone to feature interactions, which denote undesirable behaviors arising when several Web services are used together. The existing methods for detecting feature interactions suffer generally from state space explosion. We develop a method to detect feature interactions in Web services, which targets to reduce such a state space explosion problem while trying to keep an acceptable power of feature interaction detection. The proposed method is based on the use of a language called Use-Modify which models Web services at a high abstraction level. A Use-Modify model of a Web service provides information such as: who uses what, who modifies what, and specifies the frequency of each operation of use and modify by words like “always”, “sometimes” and “never”. Use-Modify is also used to indicate, for each use and modify, whether there are conditions, without giving any information on those conditions. We demonstrate the applicability of our feature interaction detection method in several examples.

KEYWORDS

Composing web services; Feature interaction detection; High abstraction level; Use-Modify model; Active/passive objects.

1. INTRODUCTION

When existing Web Services (WS) are composed to create new WS, the latter can contain undesired behaviors, which are called feature interactions (FI). Example of FI in WS: we consider a supplier that receives orders. When his stock is empty, a supplier forwards any incoming order to another supplier. Consider two WS Supplier1 and Supplier2 whose stocks are empty. We may have the following situation: Supplier1 receives an order and forwards it to Supplier2 which in turn forwards the order to Supplier1. The FI manifests itself by a blocking situation where each supplier is waiting the answer of the other supplier.

FI have been intensively studied in telecommunication services (or Telecom-services) (Bouma and Velthuijsen, 1994; Cheng and Ohta, 1995; Dini et al, 1997; Kimbler and Bouma, 1998; Calder and Magill, 2000; Amyot and Logrippo, 2003; Reiff-Marganiec and Ryan, 2005; du Bousquet and Richier, 2007; Nakamura and Reiff-Marganiec, 2009), and since more recently in WS. Many methods that have been developed to detect FI are rigorous and have a high power of FI detection. But those methods suffer from state space explosion, such as those applying model-checking techniques. The approach we will adopt to detect FI in WS targets to reduce such a state space explosion while trying to keep an acceptable power of FI detection. We model the behaviors of WS by a so-called Use-Modify language inspired from approaches in (Kimbler, 1998; Chentouf, 2011). Use-Modify is a high abstraction level formalism whose basic principle is to specify “who uses what” and “who modifies what”. The model also specifies coarsely the “frequency” of each “use” and “modify” with words like “always”, “sometimes” or “never”. Moreover, the model also indicates for each “use” and “modify”, whether there is some condition, without specifying any condition.

The structure and the contributions of this paper are as follows. In Section 2, we explain fundamental differences between composing WS and composing Telecom-services. Section 3 presents related work on

---

1 Ahmed Khoumsi is from the Department of Electrical and Computer Engineering, University of Sherbrooke, Canada. He was on sabbatical leave at CCIS, King Saud University, when the study was done.
modeling and composing WS and detecting their FI. In Section 4, we propose a Use-Modify language to model WS at a high abstraction level. Section 5 presents a Use-Modify-based method for detecting FI in WS. Section 6 demonstrates the applicability of our method for detecting all the FI of the benchmark of (Weiss et al, 2007) and one FI of (Weiss and Esfandiari, 2004). Section 7 demonstrates that our method can be used to detect several FI indicated in (Bond et al, 2009). In Section 7, we consider not only WS composition, but also a Telecom-service composition and a mixed composition of WS and Telecom-service. Section 8 concludes.

2. WS COMPOSITION VERSUS TELECOM-SERVICE COMPOSITION

Let us show that composing WS is different from composing Telecom-services.

1. Telecom-services can generally be abstracted by a few parameters. In (Kolberg and Magill, 2005), a service is abstracted by a triggering party, and origin and destination parties. An in (Chentouf et al, 2004), a service is abstracted by processing points that correspond to steps in a phone call. WS cannot be so simply abstracted, because a WS can be any imaginable software providing a service through the Web.


We deduce that WS composition may be much more complex than composing Telecom-services, and hence cannot be automated in general. To address the complexity of WS composition, several models have been developed, such as orchestration and choreography.

Let us draw your attention to an important difference between Telecom-services and WS in FI detection. The presence of FI between two composed Telecom-services depends generally uniquely on those composed services, since their composition consists simply in running them in parallel. On the contrary, the presence of FI in WS depends on the way WS have been composed, because there are many ways to compose WS.

3. RELATED WORK ON MODELING AND COMPOSING WS AND DETECTING THEIR FI

(Weiss and Esfandiari, 2004; Weiss et al, 2007) raise interest of researchers to WS composition and FI detection. Weiss et al (2007) present a case study which can be used as a benchmark to assess FI detection methods. (Bond et al, 2009) show that FI in Telecom-Services are different from FI in WS.


Some work has been done on user-interfacing and software-tooling for WS composition. Examples: Liu et al (2007) propose an environment using Mashup for WS composition; (Chafle et al, 2007) present an integrated development environment for WS composition. FI detection is not studied in (Liu et al, 2007; Chafle et al, 2007).

Turner (2005; 2007) study WS composition by considering theoretical, software-tooling and user interfacing aspects. The CRESS formalism is used which can be translated into BPEL and LOTOS.

4. USE-MODIFY LANGUAGE TO MODEL WS

In the references of Section 3, the developed FI detection methods (if any) suffer from state space explosion, because they are based on formalisms specifying WS exhaustively. The approach we adopt avoids state space explosion while keeping an acceptable power of FI detection. We develop a so-called Use-Modify language to model WS at a high abstraction level, whose principle is to specify “who uses what” and “who modifies what”. The model also qualifies each “use” and “modify” with words like “never”, “sometimes”, “may be” and “always”. Such an omission of details is motivated by the desire to avoid state space explosion during FI detection. With Use-Modify, WS are specified at two levels: their interfaces are specified like objects in object-oriented programming (OOP); and their behaviors are specified by so-called Use-Modify relations (UM-relations). A set of UM-relations modeling the behavior of a WS is called its behavior model, or its UM-model. The UM-model describes how a WS behaves but it does not necessarily correspond to its implementation. The UM-model is targeted uniquely to be manipulated by our proposed FI detection method which is presented in Section 5. While designing (and before deploying) a WS, we construct a UM-model of such a WS which is analyzed to determine whether the WS is FI prone. Therefore, our method is off-line.

4.1 Interface Model Based on Object-Oriented Programming

The interface of a WS is modeled as a class skeleton in OOP, and the interface of each executable instance of WS is modeled as an object skeleton of a class. By skeleton, we mean that the classes and objects are specified by attributes and methods signatures. A method signature specifies a function by its name, its input and/or output parameters and its returned result (if any), and without a body. Object skeleton corresponds to interface in Java. For brevity, we will omit the terms skeleton and signature in class skeleton, object skeleton, and method signature. There exist two types of attributes: basic attributes and complex attributes. Basic attributes are variables of primitive types, like int, float, double, boolean. Complex attributes are objects. For clarity, methods, basic attributes and complex attributes are named differently as follows:

- **Basic attributes** (or primitive variables): they are named in italic with the first letter non capitalized. For example, risk, rate, and amount.
- **Complex attributes** (or objects): they are named in italic with the first letter capitalized. For example, Assessor, Approver, Lender, Supplier.
- **Methods**: they are named in italic with the first letter non capitalized, and they terminate by (). For example, quote(), approve() and assess().

An attribute $a$ and a method $m()$ of an object $O$ are referred to as $O.a$ and $O.m()$, respectively. $O$ can be omitted when there is no ambiguity or when it is irrelevant. We use the notions of feature and WS as follows:

- **Feature**: it is a basic WS which is not composed of other WS. A feature is modeled as an object. When several similar features are used, they are modeled as objects of the same class. A class is named with all letters capitalized, for example, SUPPLIER.
- **WS**: it is composed of features and/or WS. Like features, WS are modeled by objects and classes.

Consider examples of features and WS in (Turner, 2007) and give an idea of how they can be modeled as objects. We do not give details; we just indicate one or two attributes and methods for each feature or WS.

**Example 1**: The feature Approver has a method approve() and a basic attribute rate. approve() evaluates a loan proposal and decides to refuse or approve it.

**Example 2**: The feature Assessor has a method assess() and two basic attributes risk and rate. assess() evaluates the risk of the loan. If risk is low, an acceptance response is returned with a proposed loan rate. Otherwise, approve() of Approver is invoked.

**Example 3**: The WS Lender is composed of the 2 features Approver and Assessor. Lender has two attributes that correspond to Approver and Assessor. Lender also has a method quote() and a basic attribute amount. The method quote() approves or assesses a loan of a given amount in the following way: quote() invokes the method approve() of Approver if amount $\geq$ 10000, or the method assess() of Assessor if amount $< 10000$. 
We have shown how WS have their interfaces modeled as objects. In the remainder of Section 4, we show how WS have their behaviors modeled at a high abstraction level by Use-Modify formalism. In the above three examples, the behaviors are indicated for information, they are not described in the objects.

4.2 Use-Modify Formalism

A method is said active if it modifies (sometimes or always) the value of some attribute (of any object). An object is said active if it contains an active method or a complex attribute which is an active object. A basic attribute cannot be active. A method or object is said passive if it is not active. Intuitively, an active object modifies some attribute. Let active access to an attribute mean an access that modifies the attribute.

The action “use” is used with various “intensities” as follows:
- “use!” means “to have certainly access to”.
- “use?” means “to have sometimes access to”; by sometimes, we mean under some specified or unspecified conditions.
- “use#” means “may have access to”, i.e., we do not know if there is an access.
- “use%” means “has not access to”, i.e., has never access to.

In the same way, the action “modify” is used with various “intensities” as “modify!”, “modify?”, “modify#” and “modify%”. The difference between “use” and “modify” is that “modify” corresponds to an active access, while “use” corresponds to an access which may be passive or active. More precisely, we have the following so-called Use-Modify relations (or UM-relations):

“L use! R” means that R is accessed in every situation where L is executed.
“L use? R” means that R is used in some situation where L is applied (maybe in all situations). This UM relation should be used in the following two cases:
- We know that L uses R in some situations and not all situations; those situations may be specified (as conditions) or unspecified (may be unknown).
- We know that L uses R, but we do not know if it does it in all situations.

“L use# R” means that we suspect that L uses R, but this is not certain.
“L use% R” means that L never uses R.

“L modify! R” strengthens “L use! R” by specifying that R is modified whenever L is applied.
“L modify? R” strengthens “L use? R” by specifying that R is modified in some situation where L is applied. This UM-relation should be used in the following two cases:
- We know that L modifies R in some situations and not all situations; those situations may be specified (as conditions) or unspecified (may be unknown).
- We know that L modifies R, but we do not know if it does it in all situations.

“L modify# R” means that we suspect that L modifies R, but this is not certain.
“L modify% R” means that L never modifies R.

use# and modify# should be avoided by the designer because they model coarse information meaning “do not know”. use# is compatible with use!, use? and use%, and modify# is compatible with modify!, modify? and modify%. use# and modify# have been defined because we will show that they can be deduced by rules.

“!””, “?”” and “#” are not written in some contexts where they are irrelevant. In this case, we write “use” to mean “use!” or “use?” or “use#”, and “modify” to mean “modify!” or “modify?” or “modify#”.

4.3 Semantics of Use and Modify When Applied to Methods and Attributes

To use a method m() means to call m(); to use a basic attribute x means to read x or change its value; to modify a basic attribute x means to change the value of x. Note that is a non-sense to modify a method. Let us define inductively the meaning of use and modify a complex attribute. Note that a complex attribute is an object, and hence it has its own attributes and/or methods:
- to use an object means to use one or more of its attributes or methods;
- to modify an object means to modify one or more of its attributes.

4.4 Contextual Conditions on Use-Modify Relations (UM-Relations)
The behavior of a feature or WS is modeled by a set of UM-relations and possibly by so-called contextual conditions on some UM-relations. Consider for example a WS Supplier, to which an order can be sent, e.g., by calling its method order(). Supplier, can itself call the order() method of another supplier of the same class SUPPLIER. This is specified by the UM-relation “Supplier, order() use? SUPPLIER.order()”. Assuming a supplier does not call its own order() method, we associate to this UM-relation a contextual condition stating that in this context, SUPPLIER does not comprise Supplier,. This is noted: “Supplier, order() use? SUPPLIER.order()” with “SUPPLIER not comprising Supplier,”.

4.5 Some Rules on UM-Relations

Here are some rules that must be respected for the sake of consistency:
- In “L use R”, L must not be a basic attribute.
- In “L modify R”, L must not be a passive attribute nor a passive method.
- In “L modify R”, R must not be a method.

The following five rules are due to the fact that “always” and “sometimes” are incompatible with “never”.

\$
\begin{align*}
R[m! \neq m\%]: & \quad \text{We cannot have at the same time “A modify! B” and “A modify\% B”} \\
R[m? \neq m\%]: & \quad \text{We cannot have at the same time “A modify? B” and “A modify\% B”} \\
R[m! \neq u\%]: & \quad \text{We cannot have at the same time “A modify! B” and “A use\% B”} \\
R[u! \neq u\%]: & \quad \text{We cannot have at the same time “A use! B” and “A use\% B”} \\
R[u? \neq u\%]: & \quad \text{We cannot have at the same time “A use? B” and “A use\% B”}
\end{align*}
\$

In Sections 4.5 and 4.6, we will present rules to derive new UM-relations from existing UM-relations.

When a rule derives a UM-relation Y from a UM-relation X (noted X => Y), we say X is stronger than Y (or Y is weaker than X). The following three rules are due to the fact that modifying is stronger than using:

\$
\begin{align*}
R[m! \Rightarrow u\%]: & \quad \text{“A modify! B”} \Rightarrow \text{“A use\% B”} \\
R[m? \Rightarrow u\%]: & \quad \text{“A modify? B”} \Rightarrow \text{“A use\% B”} \\
R[u\% \Rightarrow m\%]: & \quad \text{“A use\% B”} \Rightarrow \text{“A modify\% B”}
\end{align*}
\$

The following two rules are due to the fact that “always” is stronger than “sometimes”:

\$
\begin{align*}
R[m! \Rightarrow m\%]: & \quad \text{“A modify! B”} \Rightarrow \text{“A modify\% B”} \\
R[u\% \Rightarrow u\%]: & \quad \text{“A use\% B”} \Rightarrow \text{“A use\% B”}
\end{align*}
\$

The above rules \$R[m! \Rightarrow u\%]\$ and \$R[m! \Rightarrow m\%]\$ can be combined as follows:

“A modify! B” => ("A use! B" and "A modify\% B").

The latter five rules consist in deriving a weaker UM-relation. Their relevance is just pedagogical to draw the attention on the fact that some UM-relations are weaker than others, and that the objective of modeling is to identify and keep uniquely the strongest UM-relations. In Section 4.6, we will present more useful rules.

Here are two examples of UM-modeling (i.e., modeling with UM-relations) related to the Approver, Assessor and Lender introduced in Section 4.1.

**Example 4:** Here are UM-relations deduced from the literal descriptions in Examples 1-3 of Sect. 4.1:

1. **M1:** Lender use! Lender.quote() // Lender starts by the execution of its method quote()
2. **M2:** Lender.quote() use! Lender.amount // quote() receive amount as input parameter
3. **M3:** Lender.quote() use? Approver.approve() // quote() calls approve() if amount \(\geq 10000\)
4. **M4:** Lender.quote() use? Assessor.assess() // quote() calls assess() if amount \(< 10000\)
5. **M5:** Approver.approve() use! Lender.amount // approve() uses amount as input parameter
6. **M6:** Approver.approve() modify? Approver.rate // approve() computes rate if the loan is accepted
7. **M7:** Assessor.assess() use! Lender.amount // assess() uses amount as input parameter
8. **M8:** Assessor.assess() modify! Assessor.risk // assess() computes the risk
9. **M9:** Assessor.assess() modify? Assessor.risk // assess() computes the rate if risk is low
10. **M10:** Assessor.assess() use? Approver.approve() // assess() calls approve() if risk is high.

**Example 5:** Let us use the benchmark of (Weiss et al, 2007) to present other examples of use? and modify?. Examples 5-7 of this benchmark are related to accessing the user profile. We consider a WS Supplier that needs to have access to user profiles. We assume that each profile contains two parts: a confidential part and a public part. The two parts can be read and modified by the profile owner. The confidential part can also be read by some trusted entities, while the public part can be read by anyone.
Our FI detection method is based on UM-models of several WS $S_1, \ldots, S_2$ to be composed. To make the FI detection more efficient, we should enrich the UM-models by using rules to derive new UM-relations from existing UM-relations. Each rule is identified in the form $\{\text{Ri}\}$ and also in a form $\{\text{R}[…]\}$ that helps to guess the statement of the rule. In the following, “A and B => C” means that if we have A and B, we synthesize C:

- **R1**: $\{\text{R[u!u!=>u!]}\}$: “L use! U” and “U use! R” => “L use! R”
- **R3**: $\{\text{R[u!m!=m!]}\}$: “L use! U” and “U modify! R” => “L modify! R”
- **R4**: $\{\text{R[m!u!=m!]}\}$: “L use? U” and “U modify! R” => “L modify? R”
- **R5**: $\{\text{R[m!u!=u!]}\}$: “L modify! U” and “U use! R” => “L use! R”
- **R7**: $\{\text{R[m!m!=m!]}\}$: “L modify! U” and “U modify! R” => “L modify! R”
- **R8**: $\{\text{R[m?m!=m?]}\}$: “L modify? U” and “U modify? R” => “L modify? R”

In the above eight rules “L x U” and “U y R” => “L y R”, y equals to “use!” or “modify!”.

If we take y equal to “use?” or “modify?”, we obtain less accurate UM-relations with z that equals “use#” or “modify#”. Let us explain this in the following example: a method L calls a method U in all situations (L use! U), and U reads an attribute R only when it is called by another method than L (U use? R). Hence, L never reads R. Therefore, “L use! U” and “U use? R” do not guarantee that L uses R. In general: “L use! U” and “U use? R” => “L use# R”, which must be interpreted as “we suspect that L uses R and we must verify if this suspicion is correct or not”. Here is a list of rules of the same category, which derive actions “use#” and “modify#”.

- **R11**: $\{\text{R[u!m!=m!]}\}$: “L use! U” and “U modify? R” => “L modify# R”
- **R12**: $\{\text{R[m!u!=m!]}\}$: “L use? U” and “U modify? R” => “L modify# R”
- **R13**: $\{\text{R[m!u!=u!]}\}$: “L modify! U” and “U use? R” => “L use# R”
- **R15**: $\{\text{R[m!m!=m!]}\}$: “L modify! U” and “U modify? R” => “L modify# R”
- **R16**: $\{\text{R[m?m!=m?]}\}$: “L modify? U” and “U modify? R” => “L modify# R”

**Example 6**: Consider Example 4 of Section 4.5. If we apply Rules R1-R16 to the UM-relations M1-M10, we obtain the following UM-relations that enrich the UM-model of Lender. For simplicity, we have omitted to indicate the object of each attribute or method.

- Applying R1 to M1 and M2: $\text{Lender use! amount}$
- Applying R2 to M3 and M5: $\text{quote()} use? amount$ (weaker than M2, hence irrelevant)
- Applying R2 to M4 and M7: $\text{quote()} use? amount$ (weaker than M2, hence irrelevant)
- Applying R2 to M10 and M5: $\text{quote()} use? amount$ (weaker than M7, hence irrelevant)
- Applying R9 to M1 and M3: $\text{Lender use# approve()}$
- Applying R9 to M1 and M4: $\text{Lender use# assess()}$
- Applying R10 to M4 and M10: $\text{Lender quote()} use# Approver.approve()$ (weaker than M3, hence irrelevant)
- Applying R4 to M4 and M8: $\text{quote()} modify? risk$
- Applying R12 to M3 and M6: $\text{quote()} modify# rate$
- Applying R12 to M4 and M9: $\text{quote()} modify# rate$
- Applying R12 to M10 and M6: $\text{assess()} modify# rate$
In this example, the suspected accesses deduced from R9 and R12 are effective, hence we can replace use’ by use?. We have not shown the influence of contextual conditions in the application of rules; we will illustrate their influence in Section 6.1.

5. FI DETECTION METHOD BASED ON USE-MODIFY LANGUAGE

There exist many FI detection methods with a high power of detection, but which are prone to state space explosion. We propose an FI detection method that reduces this problem while keeping an acceptable power of FI detection. Instead of always detecting FI with certitude, our method may indicate in some cases FI with certitude, but in other cases, it just draws the attention on suspected FI, which then need to be checked. This is the price to pay to avoid state space complexity. In this Section, we propose an off-line Use-Modify-based method to detect FI in a WS during its design (from scratch or by composing existing WS). The approach is to construct a UM-model of the WS under design and then to detect FI by analyzing such a UM-model.

5.1 Some Useful Notions of Certain, Possible and Potential FI

- **(un)certain FI**: A FI is *certain* if it occurs in all execution paths. Otherwise, the FI is said *uncertain*.
- **(im)possible FI**: A FI is *possible* if it occurs in some execution paths. Otherwise, the FI is said *impossible*.
- **Potential FI**: A FI is *potential* if we do not know whether the FI is possible or not.

Here are basic relations and properties which may be useful, where \( \neg(R) \) means the negation of R.

- \( \neg(\text{certain}) = \text{uncertain} \)
- \( \neg(\text{uncertain}) = \text{certain} \)
- \( \neg(\text{possible}) = \text{impossible} \)
- \( \neg(\text{impossible}) = \text{possible} \)
- \( \text{certain} \Rightarrow \text{possible} \)
- \( \text{impossible} \Rightarrow \text{uncertain} \)
- Potential is independent of certain/uncertain
- Potential is independent of possible/impossible

Let us propose a two-step method for detecting FI in a WS during its design. The method is applicable for WS being designed from scratch or by composing given WS \( S_1, S_2, \ldots, S_n \). In the design from scratch, the FI detection has no input; while in the design by composition, the inputs are UM-models of \( S_1, \ldots, S_n \). In both cases, Step 1 (Section 5.3) consists in constructing a UM-model of the WS under design. Step 2 (Section 5.4) consists in checking whether there exist FI in the newly constructed UM-model. Let \( \text{NewWS} \) denote the new WS under design. The following subsections describe the inputs and the steps of FI detection.

5.2 Inputs: Use-Modify Models

If \( \text{NewWS} \) is being designed from scratch, we do not need any input. If \( \text{NewWS} \) is being designed by composing several WS \( S_1, S_2, \ldots, S_n \), the UM-models of \( S_1, \ldots, S_n \) are the inputs of the FI detection procedure. An approach is to require that the owner of any WS publishes its UM-model, if the owner authorizes that the WS be composed with other existing WS for the design of a new WS. The designer of \( \text{NewWS} \) can then have access to the UM-models of \( S_1, \ldots, S_n \). If a WS \( S_i \) has no available UM-model, a solution is that we construct a basic UM-model of \( S_i \) from the knowledge we can have of \( S_i \).

5.3 Step 1: Constructing a UM-Model of the WS Under Design

A UM-model of \( \text{NewWS} \) which is being designed by composing several WS is obtained by merging (automatically) the UM-models of the composed WS, followed by non-automatable treatments on the obtained UM-model. A non-automatable treatment consists in removing, adding and/or replacing a UM-relation. If \( \text{NewWS} \) is being designed from scratch, we need to construct manually its UM-model. After that, the obtained UM-model of \( \text{NewWS} \) is enriched “maximally” by synthesizing all the new UM-relations that can be obtained from Rules R1-R16 (Section 4.6). This can be done with a fix-point method which repeats
the application of Rules R1-R16 until no new UM-relation is generated. The method converges because of the finite numbers of Rules (R1-R16) and actions (use*, modify*). Note that this enrichment is automatable.

5.4 Step 2: Detecting Generic or Specific FI

Our method indicates some FI with certitude, while it just draws the attention on other suspected FI, which then need to be checked. FI detection consists in analyzing the UM-model of NewWS obtained in Step 1, and in generating an FI detection verdict. The analysis consists in checking the following FI patterns:

**Case 1.** There exists a UM-relation “a() use! a()” or “a() use? a()” or “a() use# a()”, where a() is a method. This is symptom of looping behavior (Example of Sect. 6.1):
- If it is determined that a() is never executed from the original state, the FI is impossible;
- Else if the action is “use!” and a() is certainly executed from the original state, the FI is certain;
- Else if the action is “use!” and a() is possibly executed from the original state, the FI is possible;
- Else the FI is potential (Example in Section 6.1).

**Case 2.** There exist UM-relation(s) that “modify” and possibly “use” the same entity. That is, we have two or more UM-relations “K m R” and “L n R”, where m = modify and n = use or modify. This is the symptom (hence the potentiality) of resource conflict or race condition, (Examples in Sections 6.4, 6.7, 7.1, 7.2, 7.3)

**Case 3.** There exist UM-relation(s) obtained (in Step 1) by changing (removing, adding or replacing) some UM-relation(s) of S1, …, Sn; this may imply losing mandatory characteristics or adding forbidden characteristics of S1, …, Sn. Hence, the potentiality of FI that needs to be checked (Example in Sect. 6.2).

**Case 4.** There exist UM-relation(s) with restrictions which must be checked. By the generic term “restriction”, we mean the following situations:
- There exist “use?” or “modify?”, which are associated to specified or unspecified conditions (Sect. 4.2). Hence the need to check that every specified condition corresponding to a use? or modify? is satisfied.
- There exist “use %” or “modify %”, hence the need to check that such use or modify is never applied.
  
  The two sub-cases of Case 4 are illustrated in Section 6.5 with use? and modify%.

  In addition to the generic Cases 1-4, we may have the following specific cases.

**Case 5.** There exist UM-relation(s) that “use!” or “use?” methods p() and q() which are incompatible with each other. This is a symptom of inconsistent behavior. Consider the following two sub-cases (a) and (b):

(a) “A use! p()” and “A use! q()”: the inconsistency associated to these UM-relations is certain assuming that A is reached.
- If we determine that A is never reached from the original state, the FI is impossible;
- Else if we determine that A is certainly reached from the original state, the FI is certain;
- Else if we determine that A is possibly reached from the original state, the FI is possible;
- Else (i.e., we cannot determine whether A is possible), the FI is potential.

(b) “A use? p()” and (“A use? q()” or “A use! q()”): the inconsistency associated to these UM-relations is potential assuming that A is reached.
- If we determine that A is never reached from the original state, the FI is impossible;
- Else the FI is potential (Example in Section 6.3).

This case is said specific because it necessitates additional specific information (given as input of the FI detection procedure) indicating the incompatible methods.

**Case 6.** There exist (resp. do not exist) specific UM-relation(s) specified as forbidden (resp. mandatory) (Examples of Sects. 6.2 and 6.6). This case is specific because it res additional specific information (given as input of the FI detection procedure) indicating the mandatory and forbidden UM-relations.

FI detected with Cases 5 and 6 are said specific, while FI detected with Cases 1 to 4 are said generic.

We demonstrate our FI detection method in a benchmark of eight FI which are constructed on the case study of a virtual bookstore (Weiss et al, 2007). The following individual WS are defined: iPassport is an identity management WS that simplifies authentication with multiple service providers; PayMe is a payment processing WS that allows payers to make secure payments online, and simplifies credit card processing for payees; ShipEx is a shipping WS that provides shippers with guaranteed delivery of product, and simplifies tracking of a shipment for shipees; Shark is a caching WS that improves performance by storing the results of previous requests. Three composite WS Amazin, Supplier and Customer are constructed from the above individual WS. Amazin is a virtual bookstore which relies on Suppliers, and gives Customers access to its virtual catalog and the option to order books from the catalog through an Order Processing features.

6.1 Example 1 of (Weiss et al, 2007)

The FI manifests itself by a blocking situation in the following way. An order is sent to Supplier₁ (by calling a method order() of Supplier₁) who forwards the order to Supplier₂ (by calling a method order() of Supplier₂) because his stock is empty. Then, Supplier₂ in turn forwards the order to Supplier₁ (by calling a method order() of Supplier₁) because his stock too is empty too. Hence, we reach the blocking situation where each supplier is waiting the reception of the ordered book from the other supplier. Let us see how our FI detection method detects such FI. Each of Supplier₁ and Supplier₂ is specified by a set of UM-relations with contextual conditions. As seen in Section 4.4, we have the following UM-relations with contextual conditions:

UM₁: “Supplier₁.order() use? SUPPLIER.order()” with “SUPPLIER not comprising Supplier₁”;
UM₂: “Supplier₂.order() use? SUPPLIER.order()” with “SUPPLIER not comprising Supplier₂”.

In Step 1, the UM-relations are composed. We apply rule R10 to UM₁ and UM₂, but after setting SUPPLIER of UM₁ and UM₂ to Supplier₁ and Supplier₂, respectively: UM₁-UM₂: “Supplier₁.order() use# Supplier₂.order()”. We are in Case 1 of Step 2 (Sect. 5.4). Supplier₁.order() can be executed from the initial state, and hence the FI is possible. Note that this scenario can be generalized to a loop involving more than two suppliers: Supplier₁ is waiting Supplier₂ who is waiting Supplier₃ … Supplierₙ who is waiting Supplier₁.

6.2 Example 2 of (Weiss et al, 2007)

The FI manifests itself by the fact that, if an ordered book is in the cache (because it has been previously purchased), then the process payment is shortcut. Hence, the order is completed without payment. Let us see how our FI detection method detects such FI. Supplier and Caching WS are specified by a set of UM-relations. Consider a method completeOrder() which is called in Supplier when everything is ready to start payment and delivery processes. The payment process starts by calling a method pay(). A UM-relation which is particularly relevant in this example is: completeOrder() use! pay().

In Step 1, the UM-relations of Supplier and Caching are composed. This example illustrates the situation where composing two WS requires a human intervention to change the process payment of Supplier as explained above. The present composition has the effect to replace the call of a method pay() by a conditional call. Hence the above UM-relation is replaced by the UM-relation “completeOrder() use! pay()” (i.e., “use!” replaced by “use?”). We are in Case 3 of Step 2 (Sect. 5.4). Another way to detect the FI is to consider that the UM-relation “completeOrder() use! pay()” is specified as mandatory. The FI is deduced by the fact that the composition has modified this mandatory UM-relation. We are in Case 6 of Step 2 (Section 5.4).

6.3 Example 3 of (Weiss et al, 2007)

We consider two situations of FI that may occur when the order of a book is aborted (before its completion):

(a) FI Called “Order Processing – Delivery” in (Weiss et al, 2007): The FI manifests itself when, due to timing errors, a process payment is aborted while the delivery is completed (instead of being aborted). Hence, the possibility to receive an unpaid book (as in Example 2, but for a different reason).

(b) FI Called “Order Processing - Process Payment” in (Weiss et al, 2007): The FI manifests itself when, due to timing errors, a delivery is aborted while the process payment is completed (instead of being aborted). Hence, the possibility to pay for a book which is not received.

A supplier WS is composed of several features such as: ProcessPayment, Delivery, and OrderProcessing, each one being described by UM-relations. In Step 1, these UM-relations are composed to obtain a UM-
model of Supplier. Let us see how our FI detection method detects FI. The UM-model of Supplier uses the following methods: abortOrder() is called to abort the current order, pay() is called to start payment for the ordered product, and deliver() is called to start delivery of the ordered product. abortOrder() is incompatible with deliver() and pay(), because payment and delivery must not be done when an order is aborted. The UM model contains the following UM-relations: “Supplier use? abortOrder()” “Supplier use? deliver()” “Supplier use? pay()”. We are in Case 5 of Step 2 (Section 5.4). The combination of the 1st and 2nd UM-relations is a symptom of inconsistency, because abortOrder() and deliver() are incompatible; this is illustrated by FI (a). The combination of the 1st and 3rd UM-relations is a symptom of inconsistency, because abortOrder() and pay() are incompatible; this is illustrated by FI (b).

6.4 Example 4 of (Weiss et al, 2007)

The FI considered here is due to an ambiguity on the semantics of price. More precisely, the FI manifests itself when some features use the term price, but assigning it different semantics. For example, one feature considers the price including taxes, while another feature considers the price excluding taxes. Let us see how our FI detection method detects such FI. The UM-model and Step 1 are as in Example 3 (Sect. 6.3). The UM-model of Supplier uses two methods orderProcessing() and fulfillOrder() that modify an attribute price, that is, we have the following UM-relations: “orderProcessing() modify? price” “fulfillOrder() modify? price”. We are in Case 2 of Step 2 (Section 5.4).

6.5 Examples 5, 6 and 7 of (Weiss et al, 2007)

We consider Examples 5, 6 and 7 together, because they correspond to several variants of the same problem: non respecting the profile access policy. Intuitively:

- In example 5 (called “Authenticate User - Access profile” in (Weiss et al, 2007)): an untrusted supplier accesses some information in the profile of the customer.
- In example 6 (called “Access Profile - Access profile” in (Weiss et al, 2007)): a trusted supplier accesses some information in the profile of the customer, which must be accessible uniquely to the customer.
- In example 7 (called “Manage Profile - Access profile” in (Weiss et al, 2007)): a supplier accesses some information in the profile of the customer when the latter is not connected.

After Step 1, we obtain UM-relations such as: “Supplier use? profile” “Supplier modify% profile”

We are in Case 4 of Step 2 (Section 5.4). The “use?” corresponds to the specified condition requiring that only the authorized suppliers can read a user profile. The “modify%” corresponds to the restriction specifying that no supplier is authorized to modify a user profile. Hence, the need to check if these authorizations are respected. The FI of Examples 5, 6 and 7 are due to the non-respect of some authorizations.

6.6 Example 8 of (Weiss et al, 2007)

The FI manifests itself by a blocking situation where Supplier1 is waiting Supplier2 who in turn is waiting Supplier1, which corresponds exactly to Example 1. Hence Examples 1 and 8 are identical, but in Example 8, the FI is presented with a different viewpoint: None of the suppliers is available to the other one. A way to detect this FI is given in Section 6.1. Let us present another way to detect this FI. We assume that an attribute available is set to false by Supplier when he cannot treat requests. The UM-model obtained after Step 1 will then contain the following UM-relation “Supplier modify? available”, which is a symptom that availability changes and hence available can be false in some situations. Let us consider that all UM-relations “X modify? available” or “X modify! available” are specified as forbidden. The FI is deduced by the presence of a forbidden UM-relation. We are in Case 6 of Step 2 (Section 5.4).

6.7 Example of (Weiss and Esfandiari, 2004)

The FI manifests itself when the Spell Checker and the Formatter use different languages, e.g., US English and UK English. At the formal level, this FI is similar to the FI of Example 4. In the latter, two methods modify an attribute price. In the present example, two features SpellChecker and Formatter modify an
attribute `lang` specifying the used language. That is, we have the following UM-relations: “SpellChecker modify? `lang`” “Formatter modify? `lang`”. We are in Case 2 of Step 2 (Section 5.4).

7. DEMONSTRATION IN THE DETECTION OF SEVERAL FI OF (Bond et al, 2009)

Bond et al (2009) present an interesting comparative study showing that FI in Telecom-Services are different from FI in WS, and hence FI detection methods developed for the former cannot be easily adapted for the latter. We will apply our FI detection to three types of FI given in (Bond et al, 2009): FI between two WS; FI between two Telecom-services; FI between a WS and a Telecom-service. As we will see, the three FI are detected with Case 2 of Step 2 (Section 5.4).

7.1 FI Between Two WS of (Bond et al, 2009)

The FI manifests itself when the Logging WS uses the encrypted information (purchase order or payment information) while Logging needs to use the information before it is encrypted. After Step 1, we obtain UM-relations where an attribute `paymentInfo` is modified by a method `encrypt()`, while another method `logging()` reads the attribute `paymentInfo`. That is, we have the following UM-relations: “encrypt() modify! `PaymentInfo`” “logging() use! `PaymentInfo`”. We are in Case 2 of Step 2 (Section 5.4).

7.2 FI Between Two Telecom-Services of (Bond et al, 2009)

Contrary to previous examples, here we consider Telecom-services instead of WS. The FI manifests itself when a caller rejected by Call-Blocking (CB) of a callee is able to leave a (potentially unwanted) voicemail via Voicemail (VM). The particularity of Telecom-services is that Step 1 may be automated (see Section 6.3). We obtain UM-relations where an attribute `callStatus` is modified by `CB` (to busy status) and read by `VM` (busy status is the trigger of VM). That is, we have the following UM-relations: “CB modify! `callStatus`” “VM use! `callStatus`”. We are in Case 2 of Step 2 (Section 5.4).

7.3 FI Between a Telecom-Service and a WS of (Bond et al, 2009)

This is a special case, in the sense that we have a mixed composition, i.e., a WS is composed with a Telecom-service. The FI manifests itself when a customer wants to be joined by an agent to talk with him (WS called TTA), while he has configured the Telecom-service Do-Not-Disturb (DND) to reject all calls. After Step 1, we obtain UM-relations where the attribute `callStatus` (already used in the example of Section 7.2) is modified by `DND` (to the status busy, for example) and read by a method `tta()`. That is, we have the following UM-relations: “DND modify! `callStatus`” “tta() use! `callStatus`”. We are in Case 2 of Step 2 (Section 5.4).

8. CONCLUSION

We have developed a method to detect FI among WS, which makes a trade-off between reducing state space explosion and increasing the power of FI detection. The proposed method is based on a language called Use-Modify which describes WS at a high level by indicating uniquely information such as: who uses what, who modifies what, and the frequencies and restrictions of each use and modify. We have demonstrated the applicability of our FI detection method in several cases. Indeed, we have applied our method to detect all FI of the benchmark of (Weiss et al, 2007) and for the detection of an FI in (Weiss and Esfandiari., 2004). We have also applied our method to detect several FI indicated in (Bond et al, 2009), where the composed services can be WS and/or telecommunication services. We think that our FI detection approach can be better than (Weiss et al, 2007) because in the latter many modeling formalisms have to be used: Goal-oriented Requirement Language (GRL), Use-Case Maps (UCM), and Finite State Processes (FSP).
As a future work, we plan to study FI resolution, which consists in solving the detected FI. Another planned future work is to develop a Use-Modify-based prototype to validate and improve our approach.

REFERENCES