Web Services Feature Interaction Detection based on Situation Calculus

Jiuyun Xu[1,2], Wengong Yu[1], Kun Chen[1]
[1]School of Computer and Communication Engineering
China University of Petroleum
Dongying, China
e-mail: jyxu@upc.edu.cn; yuwengong@163.com;
ck_star@126.com

[2]State Key Lab of Networking and Switching Technology
Beijing University of Posts and Telecom
Beijing, China

Stephan Reiff-Marganiec
Dept. of Computer Science
University of Leicester
Leicester, UK
e-mail: srm13@le.ac.uk

***Abstract* – Feature interaction has been identified as a problem in the telecommunications domain in the 1980s, but since it has been shown to be a problem of systems that are composed of individually designed components. Clearly Web service composition is a way of building services from independently designed components and hence is subject to the same problem. This paper investigates the detection of feature interactions in Web services at runtime and proposes a novel detection method by taking inspiration from the Situation Calculus. Two case studies show that it is effective for detecting feature interactions in composite Web services.**

Keywords: Feature Interaction; Web services; Interaction Detection; Situation Calculus; OWL-S;

I. INTRODUCTION

Features as well as services are units of functionality which are correct on theire own, but when used in combination they might influence their behavior in undesired ways. This problem has been known as Feature Interaction, a term coined by Bellcore in the late 1980s. The Feature Interaction (FI) problem [1, 2, 3] has become one of the important bottlenecks for the deployment of new services. FI is not a bug in the implementation of individual services but an emergent behavior if features are used in conjunction. However, the feature interaction problem is not limited to telecommunications, it can occur in any software system that is subject to changes and build out of individually designed components.

Service-oriented architecture (SOA) holds the promise for businesses of allowing for quick adaptation of systems. Web services are a way of encapsulating application functionality in a location and implementation transparent manner. However, if services are composed the potential feature interaction arises. Akin to FI, Web services may interact with each other in unexpected and often undesirable ways negatively affecting service quality and user satisfaction. [4] describes this as Web Services Feature Interaction (WSFI) problem.

Interaction is fundamental to Web services and service-oriented architecture, and in general is desirable – however there are situations that need to be avoided as they are undesirable. Feature interaction detection and resolution would be concerned with the latter. While telecoms markets have traditionally been closed and tightly controlled, the FI problem was manageable due to in house design knowledge, small numbers of features and good available of working details [5]. As the telecoms market became more open, the need for solutions to FI increased and new challenges were posed. The Web services market has always been open, with many people providing services that are supposed to work seamlessly together. Hence lessons learned in telecomms, should be considered in the context of the WSFI problem. Predominately, the detection and resolution of WSFI problem will become important to increase introduction of new services and the robustness of composite services.

There has been plenty of work on the prevention, detection and resolution of FI in the telecommunication systems [3], but the traditional detection methods are not suitable for the problem in Web services as: (1) web services are not centrally controlled and there is no global understanding of side effects and the operation of the services and (2) FI in Web services is based on undesirable side effects such as an inconsistent states, or data inaccuracies rather than inconsistent events as is often the case in telecoms.

Hence there is a need for methods that operate at runtime to detect interactions which are caused by services encountering each other in their operation and producing data based side effects that can lead to inconsistencies and violation of assumptions.

The rest of this paper is organized as follows: section 2 introduces some related work, section 3 provides required background on the Situation Calculus and OWL-S. Section 4 describes our online detection method in detail, including some case studies. In sections 5 and 6, we discuss our method and conclude the paper.

II. RELATED WORK

In the telecommunications domain feature interaction has long been established as a problem leading to a slowdown in the introduction of new services. As a similar problem can be seen in the area of web services, called Feature

Interactions in Web Services (WSFI), a similar negative impact on the introduction of new services can be expected.

Traditional attempts to address the feature interaction problem include offline method and online methods. Offline methods are applied during design time or for web services composition time of services, online method are applied while the features or services are being executed. Offline methods typically depend on the internal service logic (modeled at some level of abstraction). Online methods either use negotiation or feature interaction managers (FIM). Negotiation based approaches regard the components of the networks (user, terminal and value-added service, etc.) as different intelligent entities and detect and resolve FI by exchanging intentions of those entities. The FIM method adds a FIM into the network to detect and resolve FI. A more detailed overview of FI methods can be found in [3].

In the existing work on feature interaction in Web services some investigations have been conducted into offline methods and have yielded some results. Weiss et al. [6] presents a goal-oriented approach for detecting feature interactions among Web services. The authors also distinguish explicitly between functional and non-functional feature interactions. In [7] this is extended with emphasis on the classification of feature interactions. Moreover, they analyze the different types of the potential WSFI in a "Virtual Bookstore" scenario.

Turner [8] extends the feature notation CRESS to graphically and formally describe Web services and service composition. Moreover, he briefly discusses the WSFI detection using CRESS and a scenario notation called MUSTARD.

Zhang et al. [9] propose a Petri net-based method to detect race conditions, which can be seen as one type of functional feature interactions. Moreover, in [4] they propose a multi-layer WSFI detection system.

Offline methods require insight into the internal service logic, details of which might not be publically available; furthermore they require a knowledge of all available services and cannot consider interactions that occur because of run-time data. Similar issues have driven online FI work in telecoms and we show that some types of interaction cannot be caught by offline methods. Moreover, in the Web service arena the number of available services is very large, growing rapidly and services are offered by a large number of providers (a very open market), which does not lend itself much to offline methods.

Based on these observations we are proposing an orthogonal method, a runtime method based on Situation Calculus to detect FI in the Web services area. Our method analyses the semantics of the interacting messages rather than utilizing the internal information of service logic in atomic services. The necessary details are obtained from OWL-S (Web Ontology Language for Services) service descriptions and observed SOAP messages.

III. BACKGROUND

Our detection method is based on the use of Situation Calculus and OWL-S. In this section, we briefly introduce the Situation Calculus concentrating on the basic concepts of the language that will be used. For a more detailed introduction we refer to [11, 12]. We also provide a concise introduction to OWL-S, more information can be found in [13].

A. Situation Calculus

The Situation Calculus is a many-sorted, first-order logical language extended with induction. It is intended for representing a dynamically changing world. The main idea is that any system can be defined by a fixed initial situation. From any situation another situation can be reached through an action or an action set. The sorts *S*, *A*, *F* and *D* for *situations*, *actions*, *fluents*, and *domain objects*, respectively. *Situations* represent a snapshot of the world plus a history of the evolution. *Actions* are regarded as the only mean by which the world evolves from one situation to another. *Fluents* are first-order functional terms which denote properties of the world that are static. For example, the binary fluent $on(x, y)$ could mean that x is placed on y. Fluent formulas are first-order formulas in which every atomic sub-formula is a fluent, e.g. the fact that all objects x are on the table can be written as $\langle \forall x. on(x, table) \rangle$. The following are elements of the language:

s_0: a constant denoting the initial situation, that is the state of the world before anything has occurred.

$do: A \times S \rightarrow S$: for an action $a \in A$ and a situation $s \in S$ $do(a, s)$ is the situation resulting from executing action a in situation s.

$holds: F \times S$: $holds(f, s)$ is true if and only if fluent or fluent formula f holds in a situation s.

$poss \subseteq A \times S$: $poss(a, s)$ is true if and only if it is possible to execute action a in situation s.

$\prec \subseteq A \times S$: is a binary predicate which represents a partial order between situations. $s \prec s'$ is true if and only if it is possible to reach situation s' from s by executing a positive number of actions.

A particular domain is modeled by the definition of several axioms (informally we might call these "domain specific fluents").

We regard, as some other authors, a composite Web service as a situation and execution of a Web service is regarded as an action. Before a service is executed, the composite service is in the initial state.

B. OWL-S

OWL-S is the major description language for semantic web services. It is based on an ontology of service concepts that supply a Web service designer with a core for describing the properties and capabilities of a Web service in an unambiguous computer-interpretable form.

OWL-S organizes a service description into four parts: the process model, the profile, the grounding, and the

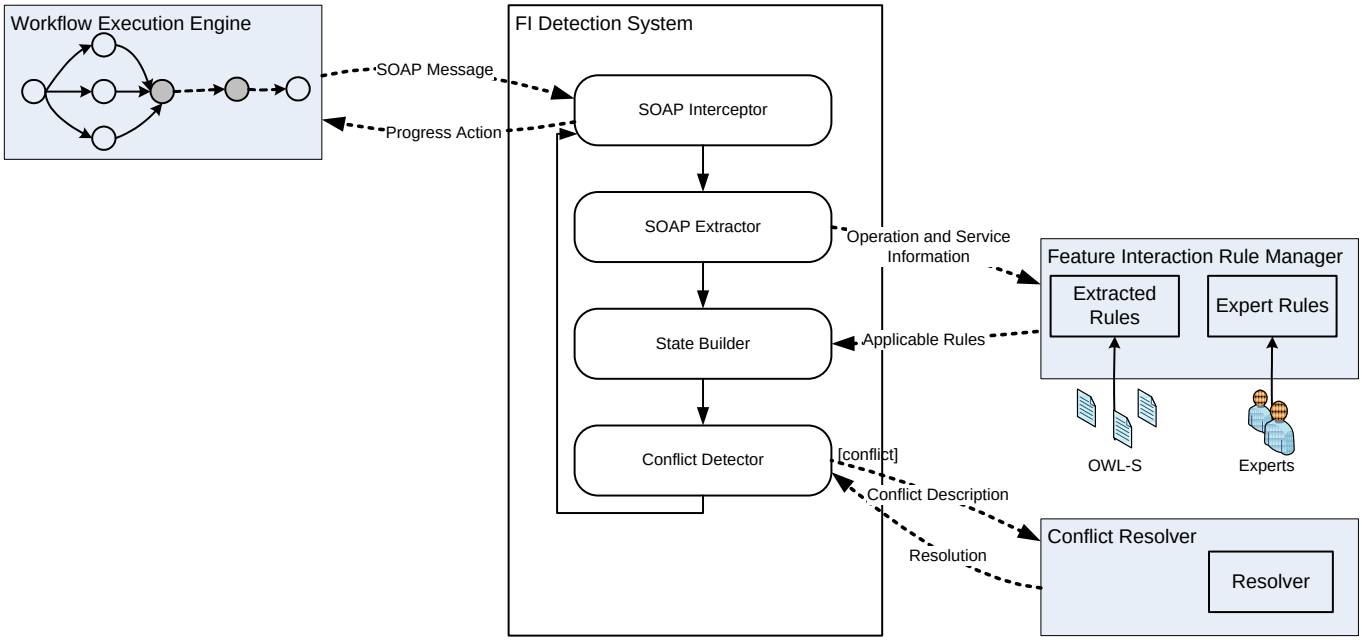


Fig. 1: Overview of the architecture of the detection system

service. The OWL-S process model is most useful for the work presented here, as it provides the required metadata about the Web services.

Each OWL-S process is based on an IOPR (Input, Output, Precondition, Result) model. *Input* represent the information that is required for the execution of the process, *Output* represents the information that the process returns to the requester. *Preconditions* need to hold over *Input* for the process to be successfully invoked. *Result* allows to specify details about the (possible many) results. Each result can be associated with a result condition, called *inCondition*, which specifies when a particular result can occur. Therefore, an *inCondition* binds inputs to the corresponding outputs. It is assumed that such conditions are mutually exclusive, so that only one result can be obtained for each possible situation. When an *inCondition* is satisfied, there are properties associated that specify the corresponding output (*withOutput* property) and, possibly, the *Effects* (*hasEffect* properties) produced by the execution of the process. *Effects* will become true when the service completes and will change the state of the world. The OWL-S conditions (*Preconditions*, *inConditions* and *Effects*) are represented as logic formulas.

In our method, we firstly transform process descriptions (mainly *Precondition* and *inCondition*) into sets of rules expressed in an ontology-aware rule language, namely Semantic Web Rule Language (SWRL) [14]. This is based on the method presented in [15], which discusses this in more detail. Then we define relevant predicates using the rules to express the composite Web services state.

IV. THE METHOD

Our method is used to detect Web service feature interactions during the execution of the service composition. In this section, we present an overview of the architecture and describe the detection process in detail. Two case studies show examples for the detection of the two conflict types that exist (the lack of resources to complete a latter part of a workflow and the attempt to invoke a service whose pre-conditions are not met anymore after an earlier service execution) and are detected.

A. The System Architecture

The FI detection system interacts with three possibly external systems: (1) a standard workflow engine executing the service composition (the engine needs to allow for the interception of SOAP messages as well as temporary blocking of the execution), (2) a feature interaction rules manager and (3) a conflict resolver. The rules manager provides descriptions of the domain knowledge as to what constitutes an undesirable interaction and the conflict resolver provides a solution for recovering the system from conflicts. Clearly the rules are not tied to a specific instance, but are generic and there is a number of resolution strategies possible ranging from manual resolution via automatically applied priorities to possibly more complex schemes.

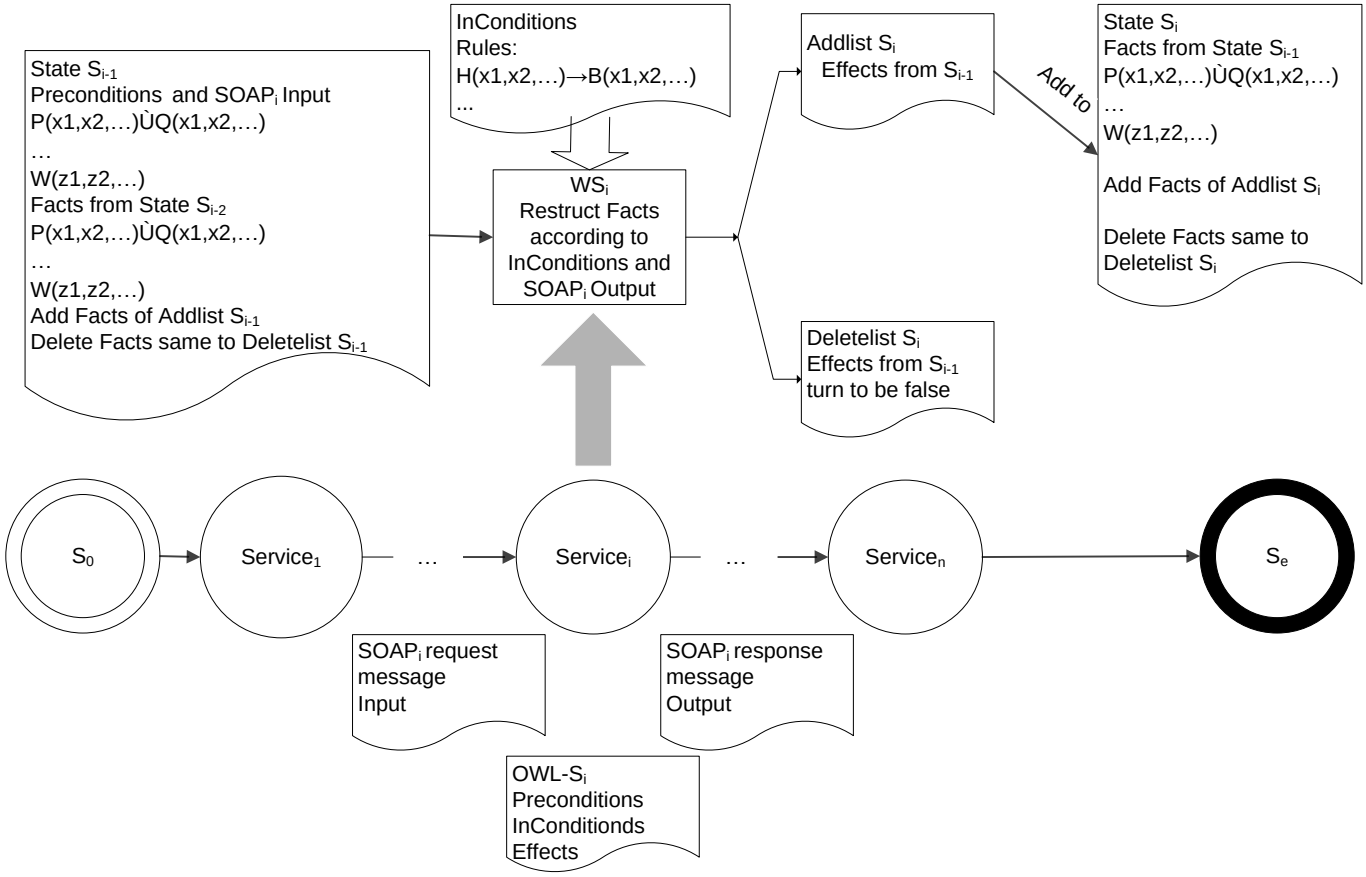


Fig. 2: Outline of the situation calculus based detection process

The architecture is designed to support service interaction detection during the execution of composite services. It allows for the detection of undesired interactions by detecting conflicts between services that could lead to a failure of the plan execution. Figure 1 provides an overview of the architecture and we will describe the main components next.

The ***Soap Interceptor*** is in charge of intercepting the SOAP message for the detection system while services interact in the workflow execution. Both SOAP request and SOAP response messages are intercepted and sent to the detection system. The workflow systems execution is halted for a brief period of time until a response comes from the Interaction detection system. If no conflicts was detected the original SOAP message will be sent with no changes, if conflicts were detected a new or adapted SOAP message might be inserted into the system. Note that this type of interception has proven to be realistic in telecommunications systems, which tend to be extremely time critical [17].

The ***SOAP Extractor*** parses the intercepted SOAP message to extract the input or output data (SOAP request and response respectively). The input and output data contains information on the service and operation used as well as the concrete data of the specific interaction. Information regarding the service and operation are sent to the FI rules manager so that applicable rules can be retrieved.

The ***State Builder*** instantiates the relevant FI rules and generates relevant predicates to express the Web service state. It uses the run-time data obtained extracted from the SOAP message and information about the service (from the OWL-S document) which is provided by the FI rules manager.

The ***Conflict Detector*** is the core part of the detection system and identifies whether there is a conflict in the current state. If conflicts are detected, the Conflict Resolver is queried for a resolution. If no conflict is detected the SOAP interceptor will notify the workflow engine to proceed; if a conflict was detected the resolution will be transmitted back to the workflow engine which will react accordingly.

The Conflict Detector consists of three subsystems: a *Knowledge Base*, an *Inference Engine* and a *Management Interface*. The Knowledge Base stores axioms of the world and reasoning rules, the Inference Engine is used to determine whether service states are consistent assuming the domain knowledge and reasoning rules. The Management Interface allows for human intervention, experts can utilize the interface to add inference rules into

the Knowledge Base according to the requirement of new services. Moreover, experts can control or adjust the detection process through the interface. While ideally the process is automatic, in practice it is useful to have the manual mechanisms available to increase detection accuracy in the light of unknown feature interactions.

In addition, two more subsystems can be attached to the Conflict Detector: an *Interaction Information Base* and an *Event Recorder*. The Interaction Information Base is a library storing information on all known services interaction phenomena [4, 12], which aids in detecting known interactions more quickly and accurately. The contents of this library will be periodically updated to contain new knowledge. The Event Recorder records the detection activities and logs the related information to a database. Experts can analyze this to improve the detection process.

B. Service Interaction Detection Process

The detection method is based on situation calculus. At the beginning, the composite service is in the initial state. After each atomic service within the composite service is executed (that is an action is taken), we get a new service state (or situation), and so on. If the former state is inconsistent with the latter one, or some predicate (fluent or fluent formula) becomes false we have identified a feature interaction. Figure 2 outlines the detection of feature interactions based on situation calculus.

Figure 3 provides an overview of the detection process, which consists of six steps as follows:

Step 1. In the first step the SOAP request or SOAP response message of the current service in the workflow execution engine is intercepted, processing is put on hold until a reply message is injected in the system.

Step 2. We extract *Preconditions*, *Effects* and *inConditions* from the OWL-S document using Mindswap OWL-S API [16], which can conveniently read or write OWL-S document. This task is performed by the Feature Interaction Rules Manager. The required data to invoke the functions is available in the SOAP message as described earlier and is transmitted in this step.

Step 3. This is the first key step in the detection process as it builds the service state information and prepares the extracted data and obtained rules for the detection phase. In more detail, we require two state pools, a *Former_state_pool* and a *Latter_state_pool*. The former denotes the state before the execution of the current service (initially this is the initial situation s_0). The latter contains information of the state after executing the service, or in situation calculus terms (assuming action a to be the service execution and s to be the current state) the state reached after $do(a, s)$. Before the service is executed, *Preconditions* and input data from the SOAP request message generate facts (predicates whose values are definitely true); these are put into the *Former_state_pool*. We also maintain two lists, called *Addlist* and *Deletelist* for storing the new predicates during the execution process of the service. The new predicates whose values are definitely true are put into *Addlist*, the predicates whose values change from true to false are put into *Deletelist*. *inConditions* is used to generate FI rules. After the service is executed, FI rules affect the service state and the state is changed according to *Effects* and the output data from the SOAP response message. In particular, we delete each predicate that occurs in both the *Deletelist* and the *Former_state_pool* from the latter. Then all remaining predicates from the *Former_state_pool* and all the predicates from the *Addlist* are added to the *Latter_state_pool*. The two state pools now represent two states during the execution process of the service composition which will be evaluated in step 4.

Step 4. In this step we determine whether a Web service feature interaction occurs. There are two situations that can lead to feature interaction (the two types of interaction mentioned earlier). One is that the *Former_state_pool* doesn't satisfy the *Preconditions* of the current service. The other is that *Former_state_pool* and *Latter_state_pool* are inconsistent. Using the Knowledge Base and the Inference Engine, we identify whether either of the two situations will occur.

Step 5. In this step information on newly detected

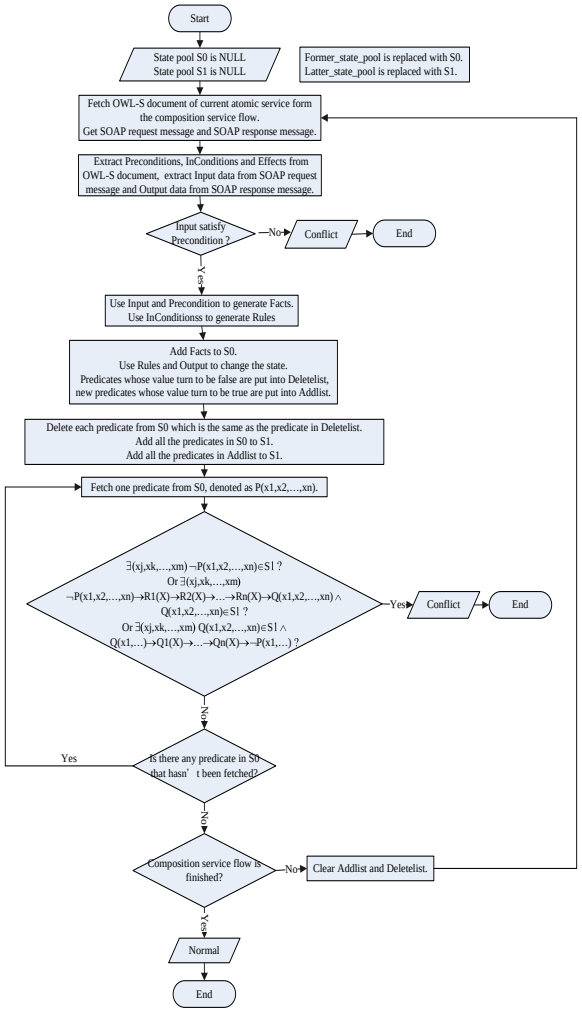


Fig. 3: The detection algorithm

interactions is recorded in the Interaction Information Base, and the events are recorded to the log. This data helps with future detection.

Step 6. If an interaction is detected the conflict resolver will be queried to provide a solution. This step will lead to transmitting progress information to the workflow execution engine and allow for the processing of the workflow to continue.

C. Case Study

In this section we present two scenarios to show how to carry out the detection process. Each example shows one of the kinds of feature interaction introduced earlier.

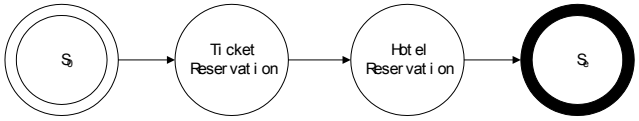
1) Case study 1: A reservation service

One person wishes go to another place in the country to attend a conference and uses a reservation service to reserve an airline ticket and a hotel. The composed service consists of an airline ticket reservation service and a hotel reservation service. Clearly the reservations include some form of payment and the respective payment features are here seen as part of the reservation service functionality. Here we only provide the OWL-S description of the charge service, the other two perform the obvious functionality. Expected behavior of a payment service occurs: the charge goes through if the card is not overdrawn; if the card is overdrawn, the only output is a failure notification as the card limit cannot be exceeded.

```
<process:AtomicProcess rdf:ID="Purchase">
   <process:hasInput>
      <process:Input rdf:ID="ObjectPurchased"/>
   </process:hasInput>
   <process:hasInput>
      <process:Input rdf:ID="PurchaseAmt"/>
   </process:hasInput>
   <process:hasInput>
      <process:Input rdf:ID="CreditCard"/>
   </process:hasInput>
   <process:hasOutput>
      <process:Output rdf:ID="ConfirmationNum"/>
   </process:hasOutput>
   <process:hasResult>
     <process:Result>
        <process:hasResultVar>
          <process:ResultVar rdf:ID="CreditLimH">
            <process:parameterType
            rdf:resource="&ecom;#Dollars"/>
          </process:ResultVar>
       </process:hasResultVar>
       <process:inCondition>
         <expr:KIF-Condition>
           <expr:expressionBody>
            (and (current-value (credit-limit
            ?CreditCard) ?CreditLimH)
            (>= ?CreditLimH ?purchaseAmt))
           </expr:expressionBody>          </expr:KIF-
         Condition>
       </process:inCondition>        <process:withOutput>
         <process:OutputBinding>
           <process:toParam
              rdf:resource="#ConfirmationNum"/>
           <process:valueFunction
               rdf:parseType="Literal">
               <cc:ConfirmationNum
               xsd:datatype="&xsd;#string"/>
           </process:valueFunction>
         </process:OutputBinding>
       </process:withOutput>        <process:hasEffect>
         <expr:KIF-Condition>
           <expr:expressionBody>
              (and (confirmed (purchase ?purchaseAmt)
            ?ConfirmationNum)                    (own
            ?objectPurchased)                    (decrease
            (credit-limit ?CreditCard) ?purchaseAmt))
            </expr:expressionBody>
          </expr:KIF-Condition>
       </process:hasEffect>
     </process:Result>
  </process:hasResult>
</process:AtomicProcess>
```

Let us consider the core aspects of the interaction detection – clearly the whole process would be applied.

The composite service flow is as follows:



Suppose that the balance of the card can afford for either the airport ticket reservation service or the hotel reservation service respectively, but cannot meet both simultaneously. Further assume that both of the two reservation services do not exceed the consumption limitation of the card. Four predicates are defined: predicate *enoughMoney* is used to denote that the remaining spent on the card is sufficient to afford the service request; predicate *noExceed* denotes that the cost of the service request will not exceed the consumption limit of the card; predicate *notOverdrawn* denotes that the card is not overdrawn and predicate *chargeCard* denotes the decreases in available spending power. The FI rule is distilled from the reservation service:

FI rule:

```
kb:enoughMoney(?process:Balance,?process:PurchaseAmt) ∧
kb: noExceed(?process:PurchaseAmt? process:CreditLimH)
→
kb:notOverdrawn(?process:CreditCard) ∧
kb:chargeCard(?process:CreditCard,?process:PurchaseAmt)
```

The service states for ticket reservation are shown in Table 1.

TABLE 1: SERVICE STATES FOR TICKET RESERVATION

Former_state_pool	Latter_state_pool
enoughMoney() noExceed()	enoughMoney() noExceed() notOverdrawn()

After the ticket reservation service is executed, we conclude that the two service states do not conflict. Then the hotel reservation service is active. The *Precondition* of the hotel reservation service is that the credit card can afford the hotel rent. But the balance of the card is now insufficient due to having executed the ticket reservation service (and of course the card is not allowed to be overdrawn). A feature interaction is detected, so the hotel reservation service is withdrawn. This case of feature

interaction is a type of resource deficit and is always dependent on the service data.

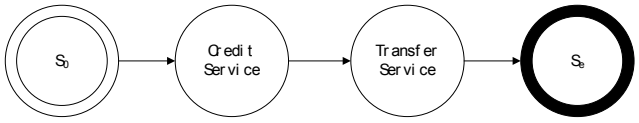
2) Case study 2: Credit risk

A credit bank provides a credit bank service. People can request the credit service on the condition that they are not in the blacklist of the credit service. There is also a transfer service, which allows to transfer loans to other people. Suppose that one person is qualified to request the credit service, which they do before requesting the transfer service, transferring the loan to person B. Now, further assume that person B is in the blacklist of the credit service. Clearly this is a case of credit risk for the bank, or in technical terms this is a feature interaction. It is desirable to detect this feature interaction and prevent for the credit risk to occur by removing the interaction.

The OWL-S description of identity validation and the credit service are as follows:

```
<process:inCondition>
   <expr:ConditionType>
     <expr:expressionBody>
       (identity-validate(?current-card ?Blacklist))
     </expr:expressionBody>
   </expr:ConditionType>
</process:inCondition>
<process:withOutput>
  <process:OutputBinding>
    <process:toParam rdf:resource="#Qualification"/>
  </process:OutputBinding>
</process:withOutput>
<process:hasEffect>
   <expr:ConditionType>
     <expr:expressionBody>
       (grant-qualification(?current-card))
     </expr:expressionBody>
  </expr:ConditionType>
</process:hasEffect>
```

The Composite service flow is as follows:



The predicate *notInBlacklist* denotes people who request the credit service are not in the blacklist of the service while *inBlacklist* denotes people who are in the blacklist of the credit service. The blacklists are integral to the credit service and can only be obtained when the service is executed. The predicate *isCreditIdentity* denotes one person is a qualified customer of the credit service, while predicate *creditTransfer* denotes credit identity transfers from one person to the other. Predicate *creditIdentity* denotes that a credit identity is granted to one person. We can distill two rules from the two services: FI rule 1 from credit service and FI rule 2 from transfer service.

FI rule 1:

```
kb:notInBlacklist(?process:PersonA) ∧
kb:Qualified(? process: Qualification) →
kb:creditIdentity(?process:PersonA)
```

FI rule 2:

```
kb:isCreditIdentity(?process:PersonA) ∧
kb: Qualified(? process: Qualification) →
kb:creditTransfer(?process:PersonA,?process:PersonB) ∧
kb:creditIdentity(? process:PersonB)
```

The service states before and after the credit service are shown in Table 2, while Table 3 shows the respective states for the transfer service.

TABLE 2. SERVICE STATES FOR CREDIT SERVICE

Former_state_pool	Latter_state_pool
notInBlacklist(PersonA) inBlacklist(PersonB)	notInblacklist(PersonA) isCreditIdentity(PersonA) inBlacklist(PersonB)

TABLE 3. SERVICE STATES FOR TRANSFER SERVICE

Former_state_pool	Latter_state_pool
notInblacklist(PersonA) isCreditIdentity(PersonA) inBlacklist(PersonB)	notInblacklist(PersonA) isCreditIdentity(PersonA) inBlacklist(PersonB) isCreditIdentity(PersonB)

From these tables it is obvious that the fact *inBlacklist(PersonB)* in the *Former_state_pool* conflicts with the fact *isCreditIdentity(PersonB)* (informally these two mean that PersonB is not credit worthy while at the same time being credit worthy) in the *Latter_state_pool* and hence a feature interaction is detected.

Note that the feature interaction is not a fault in the service composition – in general the two services would happily work with each other, but a situation that is caused by the data of the services which makes the specific flow undesirable.

V. DISCUSSION

An effective method for detecting Web service feature interaction is capable of detecting dynamically not only all kinds of specific known feature interactions, but also unknown feature interactions, in a uniform manner. According to this criterion, we present a Situation Calculus based detection method. The method overcomes the drawbacks of static detection method mostly employed in the current research and the limitation of classification-based approaches [6] which only allow for detecting interactions that are known a-priori. Our approach has the following beneficial properties:

1) The method presented is a runtime method for WSFI detection. Being a runtime method has several advantages, one of which is that it allows to work in an environment where new services might arrive and where there is no real potential for statically checking all possible combinations. So, this method will also work if the executed services in the workflow are dynamically identified and bound to. The

actual service execution data is being used in the expression of service states.

2) The presented method is especially effective for feature interactions based on instance data of the effects, which could include interactions related to security and privacy concerns. Such feature interactions cannot be detected by static methods as the occurrence of the interaction depends on instance data. As our method detects the interaction by finding inconsistencies in the service state, data sensitive interactions can easily be detected as long as the service profile specifies the preconditions and effects of an individual service correctly.

3) The method avoids full exploration of large state spaces as it only considers services that are actually invoked together rather than all possible combinations of services and furthermore only looks at inconsistency of the service state. In that way, independent of the number of atomic services involved in the service composition, we only need to consider two states: one state before an atomic service is being executed and the state after that execution. The respective state pools might contain a large number of terms for each instance, but that data would need to be considered anyhow; however the state pools are renewed after each detection step, meaning that the information considered is local to the services of current interest.

As far as feature interaction is concerned there is a general perception that approaches in feature interaction attempt to statically determine the absence of a feature interaction. However, as also shown in [3], the field of feature interaction research is quite wide and there exist run-time approaches that attempt to deal with the problem by detecting interactions and resolving them during system execution time. These approaches have inspired this work, as they are particularly adept at dealing with large numbers of services from different providers that might encounter each other for the first time when the system is running.

VI. CONCLUSION AND FUTURE WORK

With the rapid development of Web services and growing use of composite services Web services feature interaction will become a growing obstacle. While some researchers have started to address feature interaction in the web services domain, results are still very limited. By using the semantics of Web services and inspiration from the situation calculus, we proposed a novel framework and method to detect and allow for resolution of feature interactions in web services at execution time.

In future work, we intend to investigate how to decentralize the detection system. We also will test our system against more complex case studies to better evaluate efficiency and accuracy of the method. As this paper focused mostly on the feasibility of the approach in terms of detecting interactions, we are planning a more detailed evaluation of performance – clearly an important consideration for a run-time approach.

VII. ACKNOWLEDGMENT

This work is jointly supported by the National Natural Science Foundation of China (No.60672121), National Key Basic Research Program of China (973 Program) (2009CB320406) and the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No. 60821001).

VIII. REFERENCES

[1] Blair, L., Blair, G., Pang, J., and Efstratiou, C. Feature Interaction Outside a Telecom Domain. In: Proceedings of the Workshop on Feature Interactions in Composed Systems (ECOOP'2001) 15-20, 2001.

[2] Amyot, D. and Logrippo, L. Guest Editorial: Directions in Feature Interaction Research. In: Computer Networks, Special Issue on Feature Interactions in Emerging Application Domains 45 5, 563-567, 2004.

[3] Calder, M., Kolberg, M., Magill, E.H. and Reiff-Marganiec, S. Feature interaction: a critical review and considered forecast. International Journal of Computer Networks. 41: pp. 115-141, 2003.

[4] Zhang, J., Yang, F., and Su, S. Detecting the Web Services Feature Interactions. In K. Aberer et al. (Eds.): WISE 2006, LNCS 4255, pp. 169-174, 2006.

[5] Magill, E.H.. Feature Interactions: Old Hat or Deadly New Menace? In: K. Turner, E. Magill and D. Marples (Eds.), Service Provision: Technologies for Next Generation Communications, Wiley, pp. 235-252, 2004.

[6] Weiss, M. and Esfandiari, B. On Feature Interactions among Web Services. In: Proceedings of the International Conference on Web Services (ICWS), IEEE, 2004.

[7] Weiss, M., Esfandiari, B., and Luo, Y. Towards a Classification of Web Service Feature Interactions, In: Proceedings of the Third International Conference on Service Oriented Computing (ICSOC05), Amsterdam, Netherlands , 2005.

[8] Turner, K.J. Formalising Web Services. In: Proceeding of Formal Techniques for Networked and Distributed Systems (FORTE XVIII), LNCS 3731, 473-488, 2005.

[9] Zhang, J., Su, S., and Yang, F. Detecting Race Conditions in Web Services, In Proceedings of the International Conference on Internet and Web Applications and Services (ICIW'06), 2006.

[10] Zhang, J., Yang, F., Shuang, K., and Su, S. Immune-Inspired Online Method for Service Interactions Detection. In Jan van Leeuwen et al. (Eds.): SOFSEM 2007, LNCS 4362, pp. 808-818, 2007.

[11] Pirri, F. and Reiter, R. Some Contributions to the Metatheory of the Situation Calculus. Journal of the ACM, 46(2):261-325, 1999.

[12] R. Reiter. Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems. MIT Press, 2001.

[13] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/.

[14] Horrocks, I., Patel-Schneider, P.F., Bechhofer, S. and Tsarkov, D.: OWL rules: A proposal and prototype implementation. J. of Web Semantics 3 (2005), pp.23-40.

[15] Redavid, D., Iannone, L., Payne, T. and Semeraro, G. OWL-S Atomic Services Composition with SWRL Rules. Lecture Notes in Computer Science: Foundations of Intelligent Systems, vol. 4994, pp. 605-611, 2008.

[16] MINDSWAP: Maryland Information and Network Dynamics Lab Semantic Web Agents Project, OWL-S API. http://www.mindswap.org/2004/owl-s/api/.

[17] Turner, K.J., Reiff-Marganiec, S., Blair, L., Pang, J., Gray, T., Perry, P. and Ireland, J. Policy Support for Call Control. Computer Standards and Interfaces, vol 28/6 pp 635-649, 2006.