Offline Detection of Functional Feature Interactions of Web Services

Michael Weiss, Alexander Oreshkin, and Babak Esfandiari

Abstract— **This paper proposes an offline method for detecting feature interactions related to the functionality of a composite web service. There are several important ways in which functional features of web services can adversely affect each other through interaction. This problem is a particular challenge in the web services domain, since these services evolve rapidly and independently, that is, not under the control of a single party. Our approach uses labeled transition systems (LTS) to model service compositions. An LTS allows us to model the salient behavioral aspects of each web service, and to define properties for composite services through which we can detect different types of feature interactions, including assumption violation, race conditions, and incorrect invocation order. We have performed a number of case studies that demonstrate the different types of functional feature interactions, and their detection. One emphasis in the paper will be on the approach, the other on the case studies.**

I. Introduction

A service-oriented architecture (SOA) approach holds the promise for businesses that they will be able to adapt quickly and easily to changes. Services are a way of encapsulating application functionality in a location and implementation transparent manner. They package features and make them accessible to other businesses as distributed software components. However, rapid changes in the services a business provides or uses can lead to undesirable results and poor service quality: services may interact with each other in unexpected and undesirable ways. In the literature, this problem has been studied as the feature interaction problem.

This paper proposes an offline method for detecting feature interactions related to the functionality of a composite web service. There are several important ways in which functional features of web services can affect each other through interaction. A feature interaction is an unexpected and undesirable side effect of the composition of services (also referred to as features in this context). There are various causes for interactions, including race conditions, violation of assumptions, goal conflicts, and invocation order. For a categorization of the sources of feature interactions among web services see [1].

The remaining sections are organized as follows. In Section II, we give an introduction to the feature interaction problem, and to how it applies to the web services domain. In Section III, we describe our offline detection approach based on Labeled Transitions Systems (LTS), and in Section IV, we present results from our case studies that demonstrate different functional feature interactions. This is followed, in Section V, by a recap of related work. Section VI concludes the paper.

II. Feature Interaction Problem

The problem of undesirable interactions between components of a system can occur in any software system that is subject to changes. It was originally described for the problems occurring during the design of telecom software [2]. Some progress has been made recently towards explicitly modeling and analyzing feature interaction in other domains [3], [4]. The first description of undesirable side effects of web service composition as feature interactions was given in [5].

The problem concerns the coordination of interacting features such that their cooperation yields a desired result. Many hundreds of features can interact directly or indirectly, and can affect each other's behavior. Some interactions are desirable (even required as part of the design), while other interactions can lead to undesirable side effects such as an inconsistent system state, an unstable system, or data inaccuracies.

In [5], a distinction between functional and non-functional interactions is made. This distinction reflects that many of the side effects affect service properties such security, privacy, or availability. Functional feature interactions are those undesirable side effects of the composition of features that render the system no longer functional. Non-functional feature interactions, on the other hand, are undesirable side effects in a system that is working from a purely functional point of view.

As web services technology matures, it is becoming crucial to manage the interactions among web services. The feature interaction problem is presenting new challenges for the web services domain. Our focus in this paper is on functional feature interactions, which have not been covered in earlier work on feature interactions among web services (although work from the area of web service verification, and web service composition using AI planning techniques is certainly applicable to this problem). Causes for functional feature interactions

Michael Weiss is with the School of Computer Science, Carleton University, Ottawa, ON, K1S 5B6, Canada (corresponding author; phone: 613-520-2600x1642; fax: 613-520-4334; e-mail: weiss@scs.carleton.ca).

Alexander Oreshkin was with the School of Computer Science, Carleton University, Ottawa, ON, K1S 5B6, Canada (e-mail: oreshkin@comnet.ca).

Babak Esfandiari is with the Department of System and Computer Engineering, Carleton University, Ottawa, ON, K1S 5B6, Canada (e-mail: babak@sce.carleton.ca).

in web services have been categorized as follows [1]:

- —Goal/policy conflicts**:** Each feature has a specific task or goal it is trying to achieve, or policy that is follows. When there is only one web service, there is one goal or policy. However, when services are combined into a higher-level service, each with its own goals or policies, it may be that the goals or policies of those services are in conflict, and we cannot guarantee their achievement.
- —Resource contention**:** Service users may be competing with each other through access to limited resources on a service provider. Examples of such resources are: disk space, memory, CPU, network bandwidth, database access, etc. The correct operation of one service user may be compromised by the interference of another user that is using more than its share of resources.
- —Deployment and ownership: Decisions about where services are deployed, and who provides them lead to performance, scalability and quality issues, as well as to conflicts of interest. This is prevalent in web services.
- —Assumption violation: Web service developers need to make some assumptions about how a web service will be used by service users. When service users break those assumptions, the service may no longer operate correctly. Similarly, the expectations of service users may be violated by the implementation of a service.
- —Encapsulation/information hiding: If encapsulation is used, service users are not aware of the inner workings of service providers. This necessarily means that service users must make some assumptions about providers. If those assumptions are wrong, the correct operation of the service is questionable. This is another area where web service present previously unexplored challenges.
- —Invocation order: The correct operation of a composite web service may also depend on the order of invocation of some of its features. The service may assume a certain order in which events will take place. If a service consumer breaks this order, the correctness of the results is no longer guaranteed. This includes race conditions.

As noted, several of these types of interactions are specific to web services, or, in any case, more prevalent than in closed telecom systems, and require novel approaches. In [1], we have argued that web services also evolved from a closed-system assumption, where services are provided over existing, trusted relationships, to an open-system assumption.

III. Offline Detection Approach

The hierarchical architecture of building larger services from smaller services, together with object-oriented principles such as encapsulation and information hiding, creates many challenges in dealing with service interactions. It is, thus, desirable to develop formal approaches to modeling web services and detecting problematic interactions. Such techniques have been previously applied to other types of domains.

A. Process

As noted above, the most significant source of feature interactions are changes to features. These include modifications to existing features, the introduction of new features, as well as new uses of features. As the system evolves, features may be modified, new features added, and old features removed. Thus, at a given point in time t, we may have engineered a composite service that is feature-interaction free. However, after changes have been made to the features at time t+1, new feature interactions can arise, and the system needs to be reassessed.

Fig. 1 summarizes our approach. The top of the diagram represents the set of features comprising our composite service at times t and t+1. The dashed arrows indicate additions, deletions, and modifications to those features and the feature set. Given a set of features to be composed, we translate the specifications or implementations (if available) of the features into an LTS model. We then specify properties that the composite system has to meet. If Service Level Agreements (SLAs) and contracts have been explicitly provided, these can feed into the specification of the properties. Otherwise the properties encode assumptions that the features under our control rely on.

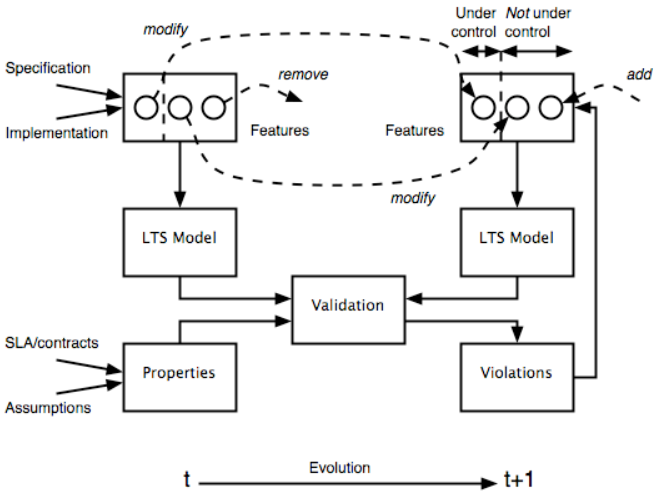

Fig. 1. Stages of the offline detection process

Validation of the properties may result in violations that correspond to feature interactions. As a result, a composite service that met the specified properties at time t may no longer meet them at time t+1, after the features have evolved. The feedback arrow indicates that we will then try to correct the interaction. Property violations can occur even though the features we are responsible for have not changed between t and t+1. As the figure indicates, not all the features we use are under our control. Although this is a known problem in general component-based systems, it is exacerbated in the web services domain due to the open nature and rapid change of web services.

Next we provide the necessary background on Labeled Transition Systems, which are used by our approach.

B. Labeled Transition Systems

The approach presented in this paper is based on Labeled Transitions Systems (LTS). The interaction models are analyzed using the LTS Analyzer (LTSA) from [6] for violation of properties that we can specify.[1] LTSA uses well-established model checking techniques based on state-space exploration to automatically analyze safety and progress properties of models. This approach lays a foundation for developing a formalized methodology to address the feature interaction problem in web services. However, we should note that the general approach for detecting feature interactions outlined in this section can used with other model checking tools such as SPIN.

A Labeled Transitions System (LTS) is a form of state machine for the modeling of concurrent systems in which transitions are labeled with action names. For small systems, a LTS can be analyzed using a graphical representation of the state machine description, but for large number of states and transitions, an algebraic notation for describing process models is required. Such a notation is provided by FSP (Finite State Processes), the notation supported by the LTSA [6]. An FSP description of an LTS can be verified to satisfy specified safety and progress properties. Informally, a property is an attribute of a program that is true for every possible execution of that program. A safety property is a statement of what is considered to be a correct execution of the system. If anything happens in the system that goes against the specifications of the safety property, the system is considered to be in error. A progress property asserts that some part of the system will eventually execute. A common example of a violation of this property is a deadlock. The analysis of a system is based on (exhaustive) state-space exploration. Its main benefit is that can be automated, thus avoiding the inherent error introduced when using manual methods such as inspection of MSCs.

An FSP model comprises a collection of constant definitions, named processes, and named process compositions. FSP offers rich syntactic features including guards, choices, variables, and index ranges. It also supports process parameters, relabeling and hiding of actions, which allow the compact modeling of component-based concurrent systems.

The analysis of a composite web service for functional feature interactions start with modeling the salient parts of the behavior of each component service (feature) as a process. The next step is to define safety and progress properties that can detect specific types of feature interactions. For example, a property that defines an expected sequence of transitions enables us to detect order of invocation interactions.

Then we use the LTSA to analyze the model of the composite service, which comprises instances of the features, and any safety properties we want to validate. As part of a service engineering approach, we can then resolve each detected feature interaction, and update the LTS model accordingly, and thus in an iterative manner complete the design by eliminating interactions. Resolution is outside the scope of this paper.

In Section IV, we will provide examples of the approach, and also introduce specifics of the modeling notation.

[1] The version used in our case studies is LTSA 2.2, which is available for download from http://www.doc.ic.ac.uk/~jnm/book/ltsa-v2/.

IV. Case Studies

To date, we have applied the offline detection approach to four case studies, and through them achieved coverage of each type of feature interaction listed in Section II. They include:

- Hotel booking service with user profile management (invocation order, assumption violation)
- Remote environment management (goal conflict)
- Pay-per-view news (assumption violation)
- Virtual bookstore (invocation order, resource contention, assumption violation, goal/policy conflict, deployment)

Two of these, pay-per-view news and portions of the virtual bookstore case study will be presented in the following.

A. News Service

The first case study involves a News service that provides clients with access to full-text articles on a pay-per-view basis. It obtains recent headlines and articles from a News Catalog service. Furthermore, the News Catalog service has been designed with the expectation that requests will be logged with a a Logging service. At the end of each billing period, News Catalog consults the log maintained by the Logging service to compile a statement and charge the client's account for their usage.

Following the process outlined in Fig. 1, we create the LTS model shown in Fig. 2. This diagram depicts processes representing features and their interconnections. This type of diagram is also known as a structure diagram [6]. Processes are represented as boxes, and externally visible actions are shown as circles on the perimeter of the box. Shared actions (that is, actions that two processes need to execute simultaneously) are shown as lines connecting two action circles. Relabeling of action names is not required in this example, as the action names are the same at either end of the lines.

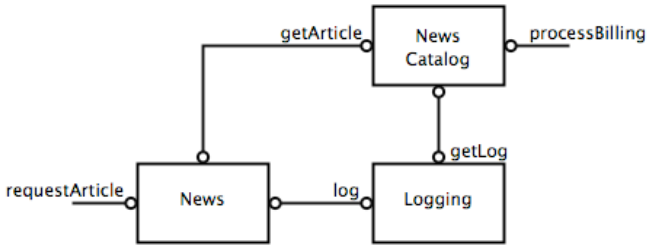

Fig. 2. Structure diagram of the initial News service

The News process is triggered by the requestArticle action, and the billing service of the News Catalog can be triggered by the processBilling action. The processes interact through the shared actions getArticle, log, and getLog. These shared actions define the external interfaces of the service components. Fig. 3 shows an FSP model of the News service. When reading the FSP model it helps to refer back to the structure diagram.

```
// Logging
LOGGING = (log->LOGGING | getLog->LOGGING).

// News Catalog
NEWS_CATALOG = (getArticle-> NEWS_CATALOG |
 proccessBilling-> BILLING),
BILLING = (log.getLog->process->NEWS_CATALOG).

// News
NEWS = (requestArticle->ACCESS_CATALOG),
ACCESS_CATALOG = (log.log->catalog.getArticle->
 NEWS).

// Composite process
||NEWS_SERVICE = (NEWS || catalog:NEWS_CATALOG ||
  log:LOGGING || P).

// Check that each article request is logged
property P = (requestArticle->log.log->P).
```

Fig. 3. Initial structure diagram for the News service (version t)

This model contains three processes (News, News Catalog, and Logging), one safety property (P), and one composite process (News Service). Each process describes possible sequences of actions. When a service provides multiple operations on its interface, this is modeled as a choice between multiple action sequences in the FSP model. Complex sequences can be made more readable by using subprocesses. For example, the News Catalog service contains a subprocess to handle Billing.

The composite process News Service represents the feature in interaction. Here, we associate labels with process instances, through which we can refer to actions in other processes from a process definition. For example, we associate the label catalog with a News Catalog process instance. The composite process also includes the safety property P. This property captures the requirement that every article request should be logged. Using the LTSA, we can perform a safety check analysis to see whether the property can be violated. The trace will tell us, if there is a sequence of events, where an article request is not properly logged. Fig. 4 shows the result of this safety check. The analysis shows that there are no violations.

```
Composition:
NEWS_SERVICE = NEWS || catalog:NEWS_CATALOG ||
 log:LOGGING || P
State Space:
 3 * 3 * 1 * 2 = 2 ** 5
Analysing...
Depth 5 -- States: 9 Transitions: 25 Memory used:
1493K
No deadlocks/errors
Analysed in: 18ms
```

Fig. 4. Safety check for the initial News service (version t)

Consider a possible evolution of the News service. Now the News service also maintains a cache of the most recently retrieved article. This feature was added to avoid charging a user more than once for retrieving the same article, and to speed up the retrieval of full text articles. When a client requests an article, the News service now first checks the cache. Only if the article is not in the cache already, is an external request made to the News Catalog service, and the retrieved article is subsequently cached. Otherwise, the cached copy of the article is immediately returned without an external request.

Fig. 5 shows a structure diagram of the evolved service.

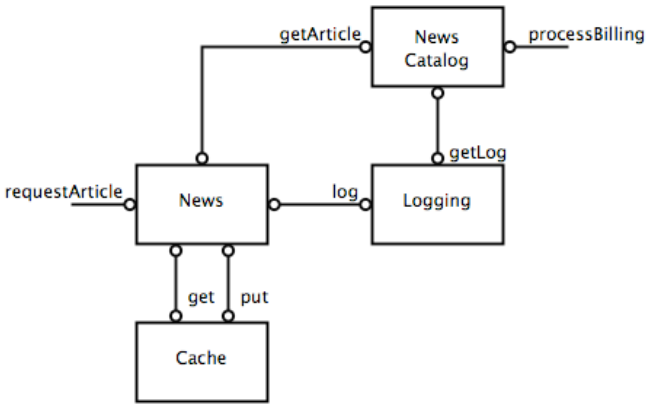

Fig. 5. Structure diagram of the evolved News service (version t+1)

Version t+1 of the service adds a Cache feature with a get and a put action, for retrieving and storing articles in the cache. This time, we extract the LTS model from the implementation of the News service. With respect to Fig. 4, Fig. 6 adds a new process to represent the Cache feature, and inserts actions into the process for the News feature to check the cache before the News Catalog is accessed. That is, the actions in the Access Catalog subprocess will only be executed, if the article is not already in the cache (simulated by the cache.notFound action).

```
// Logging
LOGGING = (log->LOGGING | getLog->LOGGING).

// News Catalog
NEWS_CATALOG = (getArticle-> NEWS_CATALOG |
 proccessBilling-> BILLING),
BILLING = (log.getLog->process->NEWS_CATALOG).

// Cache feature
CACHE = (put->NON_EMPTY_CACHE),
NON_EMPTY_CACHE = (put->NON_EMPTY_CACHE |
 get->NON_EMPTY_CACHE).

// News
NEWS = (requestArticle->CHECK_CACHE),
CHECK_CACHE = (cache.found->cache.get->NEWS |
 cache.notFound->ACCESS_CATALOG),
ACCESS_CATALOG = (log.log->catalog.getArticle->
 cache.put->NEWS).

// Composite process
||NEWS_SERVICE = (NEWS || cache:CACHE ||
  catalog:NEWS_CATALOG || log:LOGGING || P).

// Check that each article request is logged
property P = (requestArticle->log.log->P).
```

Fig. 6. FSP model of the evolved News service (version t+1)

The trace of events in Fig. 7 demonstrates that, in the new implementation of the News service, the safety property P can

be violated. This happens when a client requests an article, and subsequently another client requests the same article. Since the article was cached after the first request, the next time it is retrieved from the Cache rather than the News Catalog service. While it is fine for requests from the same user to be answered from the Cache, returning the article no matter which user sent the request, leads to an unexpected behavior.

```
Composition:
NEWS_SERVICE = NEWS || cache:CACHE ||
 catalog:NEWS_CATALOG || log:LOGGING || P
State Space:
 6 * 2 * 3 * 1 * 2 = 2 ** 7
Analysing...
Depth 9 -- States: 26 Transitions: 76 Memory used:
1580K
Trace to property violation in P:
  requestArticle
  cache.notFound
  log.log
  catalog.getArticle
  cache.put
  requestArticle
  cache.found
  cache.get
  requestArticle
Analysed in: 71ms
```

Fig. 5. Safety check for the evolved News service (version t+1)

The graphical representation of the News service can provide additional insight in the scenario that led up to the feature interaction. Fig. 6 shows the LTS for the News service, indicating (in red) the violating transition (`requestArticle`). The interaction is an example of an assumption violation. Only those client requests are being logged for which the requested article is not found in the cache. What is worse is that nobody is charged for those requests, except the first client that accessed this article and caused it to be retrieved and cached. So it is possible to gain access to an article for free (as long it does not get expelled from the cache). The behavior of the service depends subtly on when article requests are logged.

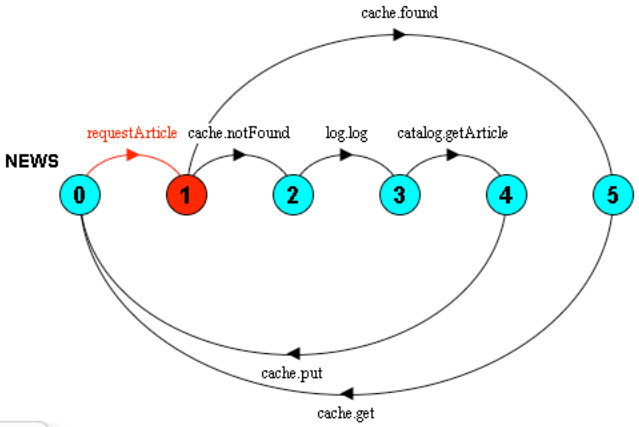

Fig. 6. LTS for the News service showing the violating transition (version t+1)

The essence of the problem, from the point of view of the News Catalog service, is that it is unaware of the Cache feature present in one of its service consumers, the News service. It (incorrectly) expects that all requests for articles made by the clients of the News service will be logged with the Logging service associated with the News Catalog. It does not expect that these consumer services can have mechanisms that will prevent some of the requests from being logged.

While we may have obtained this result from a manual inspection of the implementation, for large composite web services with many potential feature interactions a manual analysis is generally not feasible. Also, from a test-driven development view [7], it is desirable to perform the detection of problematic feature interactions automatically using formal approaches.

B. Virtual Bookstore

The second case study is part of a larger case study that aims to provide a benchmark for web service feature interactions (both functional and non-functional). In this case study, we consider a virtual bookstore (Retailer) that does not maintain an inventory of its own, but relies on its Suppliers to fulfill book orders. On receiving an order, the Retailer selects a Supplier that stocks the book, and places an order with it, in turn. The Supplier determines the availability of the ordered book, and, if successful confirms to the Retailer that the order has been fulfilled. If the chosen Supplier cannot deliver the book, the Retailer selects another Supplier, if one is available.

In our model, we focus on the Retailer-Supplier relationships, and are not concerned with the details of supplier selection, or the exception handling required when a book cannot be sourced from one of the Suppliers, or is out-of-print. Fig. 7 depicts the system architecture resulting from the Retailer-Supplier contract. The up and down labels indicate the direction of the order flow. Suppliers are downstream (down) from the Retailer in the supply chain, whereas the Retailer is upstream (up). Note that we include a Publisher process in the model, which is assumed to always fulfill an order. (Of, course, ordering from the Publisher is the least desirable option, because of the additional delay in fulfilling the order)

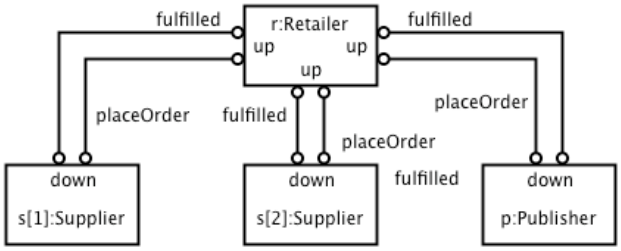

Fig. 7. Initial structure diagram for the Virtual Bookstore service (version t)

Fig. 8 shows the FSP model of the Virtual Bookstore service. The placeOrder and fulfilled actions are prefixed with the up and down labels, respectively, to indicate which process interface to use. When composing these processes we will relabel to match up down actions with corresponding up actions. Stock availability is modeled as a non-deterministic choice in the SUPPLIER

process between inStock and notInStock. If a book is not in stock, a Supplier will place an order to its downstream Supplier and wait for it to confirm order fulfillment before, in turn, confirming order fulfillment to its upstream customer.

```
RETAILER =
 (down.placeOrder->down.fulfilled ->RETAILER).
SUPPLIER = (up.placeOrder -> PROCESS_ORDER),
PROCESS_ORDER = (
 inStock -> up.fulfilled -> SUPPLIER |
 notInStock->down.placeOrder->
 down.fulfilled->up.fulfilled->SUPPLIER).
PUBLISHER =
 (up.placeOrder->up.fulfilled->PUBLISHER).
```

Fig. 8. FSP model of the Virtual Bookstore service

These features can be composed to model the scenario shown in Fig. 7. In this scenario, supplier s[1] forwards unfulfilled orders to supplier s[2], and a publisher p fulfills any orders that s[2] cannot fulfill, as shown in Fig. 9. Since the suppliers do not form a loop, we refer to this as an open supply chain, so we can distinguish between scenarios.

```
||VIRTUAL_BOOKSTORE(N=2) = (r:RETAILER ||
  forall [i:1..N] s[i]:SUPPLIER || p:PUBLISHER)
  /{
    chan[0]/r.down,
    chan[i:0..N-1]/s[i+1].up,
    chan[i:1..N]/s[i].down,
    chan[N]/p.up
  }.
```

Fig. 9. Composite service for an "open" supply chain (version t)

The N=2 indicates the number of SUPPLIER processes in the VIRTUAL_BOOKSTORE process. The forall [i:1..N] s[i]:SUPPLIER creates N composed SUPPLIER processes. The entries enclosed between /{ and } symbols are relabeling operations, each taking the form of new label/old label. A key modeling choice is to use channels (chan[i]) as the shared actions, and to map corresponding down and up actions to the same channel. This leads to a model, where the service invocations are represented as synchronous messages. If we wanted to model them as asynchronous messages, we could add separate channel processes.

Running a progress check against this composite service will not turn up any problems, as expected (see Fig. 10).

```
Composition:
VIRTUAL_BOOKSTORE = r:RETAILER || s.1:SUPPLIER ||
 s.2:SUPPLIER || p:PUBLISHER
State Space:
 2 * 6 * 6 * 2 = 2 ** 8
Progress Check...
-- States: 10 Transitions: 12 Memory used: 1530K
No progress violations detected.
Progress Check in: 64ms
```

Fig. 10. Progress check for the open supply chain scenario (version t)

However, as the virtual bookstore service evolves, some Suppliers may decide to add a capability that allows them to source orders for books they do not have in stock through their own network of Suppliers. Their incentive would be to keep the Retailer happy, even if that means sourcing a book at cost, and losing profit on some of the orders. However, this can lead to a scenario, where the order is sent along a chain of suppliers, which includes the originator of the order. Fig. 11 shows a model of a chain of suppliers. Here our goal for feature interaction analysis is to detect such chains in a given model.

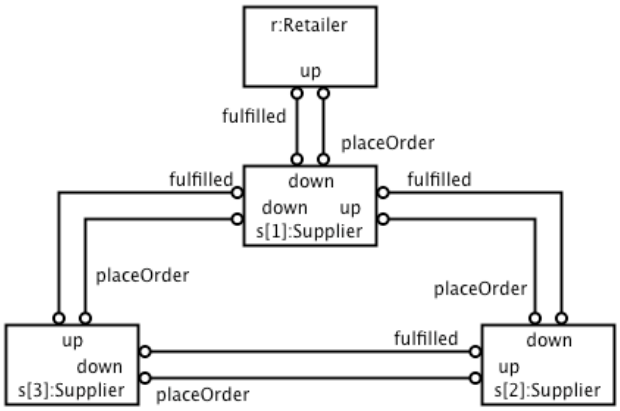

Fig. 11. Revised structure diagram for the Virtual Bookstore service allowing for Suppliers to source books from their own network of Suppliers (version t+1)

In order to model the closed supply chain depicted in Fig. 11, we only need to change the relabeling of the down action of s[3] to match up with up action of s[1]. The resulting composition is shown in Fig. 12. This demonstrates that the model in Fig. 8 is generic; it can model the impact of different supply chain topologies, and analyze them for interactions.

```
||VIRTUAL_BOOKSTORE(N=3) = (r:RETAILER ||
forall [i:1..N] s[i]:SUPPLIER || p:PUBLISHER)
/{
  chan[0]/r.down,
  chan[i:0..N-1]/s[i+1].up,
  chan[i:1..N]/s[i].down,
  chan[N]/s[1].up
}.
```

Fig. 12. Composite service for a "closed" supply chain (version t+1)

Repeating the progress check for the closed supply chain scenario, LTSA will now report a deadlock, and the sequence of actions that leads up to it, which is reproduced in Fig. 13. We could also specify an explicit progress property that the down.fulfilled action of the retailer must eventually execute.

Additional insight into the nature of the deadlock is provided by the LTS of the supplier s[1] in Fig. 14. This supplier receives the initial placeOrder request from the Retailer, as well as the forwarded request from the supplier s[3]. Since each Supplier is now waiting for its downstream Supplier to confirm order fulfillment, none of the Suppliers can make progress.

```
Composition:
VIRTUAL_BOOKSTORE = r:RETAILER || s.1:SUPPLIER ||
 s.2:SUPPLIER || s.3:SUPPLIER || p:PUBLISHER
State Space:
 2 * 6 * 6 * 6 * 2 = 2 ** 11
Progress Check...
-- States: 14 Transitions: 32 Memory used: 1735K
Finding trace...
Depth 6 -- States: 15 Transitions: 34 Memory used:
 1948K
Progress violation for actions:
  {chan[0..3].{fulfilled, placeOrder},
  s[1..3].{inStock, notInStock}}
Trace to terminal set of states:
  chan.0.placeOrder
  s.1.notInStock
  chan.1.placeOrder
  s.2.notInStock
  chan.2.placeOrder
  s.3.notInStock
Actions in terminal set:
  p.up.{fulfilled, placeOrder}
Progress Check in: 57ms
```

Fig. 13. Progress check for the closed supply chain scenario (version t+1)

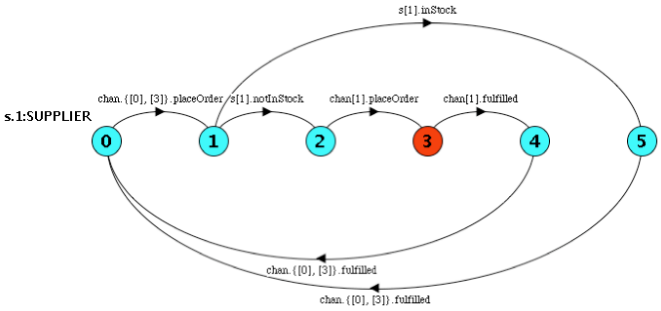

Fig. 14. LTS for the closed supply chain with progress violation (version t+1)

This deadlock is an indication of a resource contention feature interaction. It is ultimately the result of an assumption violation. As Suppliers independently decide to fulfill orders for books that are not in stock through other Suppliers, the implicit assumption is that the orders would not be forwarded back to this Supplier, but this assumption is broken in version t+1.

V. Related Work

The work on web service verification and on formal approaches in traditional feature interaction research are most relevant to this work. Examples of recent work on web service verification are [8] and [9]. Foster et al [8] describe an approach for modeling BPEL processes as LTS and verifying properties about them. Lu et al [9] present work towards reasoning about assumptions in service compositions.

A representative example of the work on formal approaches for detecting feature interactions using process algebras in the telecom domain is that by Amyot et al [10]. This paper describes a scenario-based approach to generating validation test suites and feature interaction detection by identifying scenarios with overlapping preconditions. Features are modeled both as use case map scenarios, and LOTOS processes. For a general overview of existing approaches see Calder et al [2].

However, we are not aware of previous work that looks at web service verification from a feature interaction perspective.

The closest approach in terms of analyzing service compositions using LTSA is [8]. This paper describes a tool for round-trip engineering of web service compositions using BPEL. The inclusion of BPEL within our scope is helpful, if our goal is to validate executable web service compositions. However, our level of abstraction is higher: we are operating at the specification level, and don't assume a particular implementation model for web service composition. Our work is also complementary to this work, as [8] provides very little guidance on defining the properties that we want to verify, whereas our analysis is driven by a model of the kinds of feature interactions can occur (although more work is needed to fully meet this objective in our work, as noted in Section VI). Looking forward, however, our work can be extended to include the implementation layer, and BPEL would be a prime target for our analysis.

VI. Conclusion

In this paper we proposed LTS modeling of composite web services as basis of a formal methodology for detecting functional feature interactions of web services. The approach is based on modeling the main behavioral (that is, functional) aspects of each individual web service as a process, and to define safety and progress properties that can detect potential undesirable feature interactions in the composite service.

We have applied this approach to a number of case studies. Our current goal is to define a benchmark against which different detection approaches for detecting feature interactions of web services can be compared. From our experience with the case studies it is apparent that many traditional feature interactions also occur in the web services domain. Such is the case of an incorrect order of invocation, or a forwarding loop. However, there are also issues germane to web services such as deployment and ownership, and information hiding.

Open issues of our approach include dealing with state explosion (not a specific issue to our application of LTS modeling techniques to web services), but more importantly, the systematic development of safety and progress properties. A desirable outcome of this research would be a catalog of properties (or patterns) corresponding to different types of feature interactions from which specific properties can be derived.

We would also like to note that use of a AI planning technique alone doesn't necessarily prevent the causes that we have listed all by itself: planning might allow various possible compositions, while only some of them might respect a correct invocation order or not violate assumptions. What is proposed in the paper is not incompatible with a planning technique; it simply verifies whether the generated plan (the proposed composition) is correct wrt feature interactions. It would be interesting to investigate whether our techniques could be formulated in terms of a planning problem as well (for example by specifying constraints and properties as preconditions), making the combined approach more homogeneous and elegant.

ACKNOWLEDGMENT

This research was, in part, funded through an NSERC Discovery Grant. The authors also want to thank the following fourth year project students for their participation in the design of the case studies: Yi Lin and Zhiyong Liu.

REFERENCES

[1] Weiss, M. and Esfandiari, B., Towards a Classification of Web Service Feature Interactions, *Intl. Conf. on Service-Oriented Computing (ICSOC)*, LNCS 3826, 101-114, Springer, 2005.

[2] Calder, M., Kolberg, M., Magill, E., and Reiff-Marganiec, S., Feature Interaction: A Critical Review and Considered Forecast, *Computer Networks*, 41 (1), 115-141, 2003.

[3] Pulvermüller, E., Speck, A., et al, Feature Interaction in Composed Systems, *Workshop on Feature Interactions in Composed Systems*, TR 2001-14, 1-6, Universität Karlsruhe, Fakultät für Informatik, 2001.

[4] Turner, C.R., Fuggetta, A., et al, A Conceptual Basis for Feature Engineering, *Journal of Systems and Software*, 49:1, 3-15, December 1999.

[5] Weiss, M., and Esfandiari, B., On Feature Interactions among Web Services, *Intl. Conf. on Web Services (ICWS)*, 88-95, IEEE, 2004.

[6] Magee, J., and Kramer, J., *Concurrency: State Models and Java Programs*, Wiley, 1999.

[7] Beck, K., *Test-Driven Development*, Addison-Wesley, 2003.

[8] Foster, H., Uchitel, S., Kramer, J., and Magee, J., Model-Based Verification of Web Service Compositions, *Automated Software Engineering (ASE) Conf.*, 152-163, IEEE, 2003.

[9] Lu, Z., Li, S., and Ghose, A., Web Service Conflict Management, *Intl. Workshop on the Design of Service-Oriented Applications (WDSOA) at ICSOC*, IBM Research Report RC23819, 69-78, 2005

[10] Amyot, D., Charfi, L., et al, Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS, *Intl. Workshop on Feature Interactions in Telecommunications and Software Systems*, 274-289, IOS, 2000.