Invocation Order Matters: Functional Feature Interactions of Web Services

Michael Weiss[1], Alexander Oreshkin[1], and Babak Esfandiari[2]

[1] School of Computer Science, Carleton University, Canada
weiss@scs.carleton.ca, oreshkin@comnet.ca

[2] Department of System and Computer Engineering, Carleton University, Canada
babak@sce.carleton.ca

Abstract. This paper proposes a 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 undesirable side effect of the composition of services (also known as features in this context). There are various causes for interactions, including race conditions, violation of assumptions, goal conflicts, and invocation order. We have categorized the sources of feature interactions among web services in related work. In this work, we present the results of ongoing work on the formalization of functional interactions between web services. In our approach we use on labeled transition systems to model service compositions. These models are analyzed using the LTS Analyzer to detect undesirable feature interactions. As a specific example, we look at invocation order as a source of functional feature interaction.

Keywords. Validation and verification, service composition, causes of feature interaction, invocation order, and labeled transition systems.

1 Introduction

Service-oriented approaches promise to provide businesses with the freedom they need to serve their customers no matter what software or hardware configuration their clients are using. With this technology, businesses would be able to adapt quickly and easily to changes that occur both on the client as well as the business-side.

Business services are implemented as functional software features. These features are then made accessible to other businesses as distributed software components. (Of course, services also have non-functional properties, but it is not yet standard practice to include these in the service interface.) However, rapid changes in the services due to the dynamic nature of businesses can lead to undesirable results and poor service quality, when these services are interacting with each other in undesirable ways.

In the literature this problem has been studied as the feature interaction problem. The problem of undesirable interactions between components of a system can occur in any software system that is subject to changes. However, the problem itself and approaches to address it have been largely unknown outside a small community of people specialized on the design of telecommunication switches. However, some progress has been made recently towards explicitly modeling and analyzing feature interaction in other domains [4, 5]. Undesirable side effects of web service composition as feature interactions have first been described in [6], and further developed in [7]. In [8], we presented a classification of feature interactions among web services.

Feature interactions are interactions between independently developed features, which can be either intended, or unintended and result in undesirable side effects. In [6] we make a distinction between functional and non-functional interactions. This distinction reflects that many of the side effects affect service properties such security, privacy, or availability. However, our focus in this paper is on functional feature interactions, which have not been covered in earlier work on feature interactions among web services. These include race conditions, violation of assumptions, goal conflicts, and invocation order. These are also the categories of interactions that have traditionally been studied by the feature interaction community.

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. The approach presented in this paper is based on Labeled Transitions Systems (LTS). Similar work has been presented, for example, in [1]. What our work adds is a model of the type of interactions to expect, and ways of detecting them.

The interaction models are analyzed using the LTS Analyzer (LTSA) [3] for violation of properties that we can specify. The LTSA tool uses well-established model checking techniques based on state-space exploration to automatically analyze properties of models. This approach lays a foundation for developing a formalized methodology to address the feature interaction problem in web services.

2 Feature Interaction Problem

The feature interaction problem first appeared in the domain of telecommunications [2]. The problem concerns coordination of feature interactions 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, while other interactions can lead to undesirable side effects such as an inconsistent system state, an unstable system, or data inaccuracies.

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. Causes for functional feature interactions in web services have been categorized as following [8]:

- **Race conditions.** A race condition occurs between multiple components of a composite web service (which we will refer to as feature in line with the feature interaction terminology), when the outcome of executing the service depends on timing delays (also known as glitches) between the executing of features.

- **Violation of assumptions.** Web service developers need to make some assumptions about how a web service will be used by service consumers (including other web services). When service consumers break those assumptions, the service may no longer operate correctly. Similarly, the expectations of consumers may be violated by the implementation of a service.
- **Order of invocation.** 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.
- **Competition for resources.** Service consumers 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 correctness of one consumer may be compromised by the interference of tan other that is using more than its share of resources.
- **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 these services are in conflict, and we cannot guarantee their achievement.
- **Encapsulation/information hiding.** If encapsulation is used, service consumers are not aware the inner workings of service providers. This necessarily means that consumers must make some assumptions about providers. If those assumptions are wrong, the correct operation of the service is questionable.

3 Feature Interactions in a News Service

To provide a concrete idea of how functional feature interactions can arise, we will look at a specific interaction of web services. For the example, we first present the web services involved, giving a description of each web service and its main features. Then, we illustrate and discuss the feature interaction problems arising.

This is one of several scenarios we looked. The other scenarios included a reservations system, a remote environment management system, and a supply chain. Each example illustrated different types of interactions: race conditions, violation of assumptions, conflicting goals, information hiding, and order of invocation. However, in this paper, we deliberately focus on order of invocation as one type of interaction for a better exposition of the approach in the limited space provided.

Consider a News service that provides clients with access to full-text articles on a pay-per-view basis. It uses a News Catalog service as a source of recent headlines and articles. A service that wants to use the News Catalog service must also use a Logging service associated with News Catalog. It needs to provide payment information with each request for the body of an article, which will be logged. At the end of each billing period, News Catalog will use the log maintained by the Logging service to compile a statement and charge the client's credit card (or similar) for their usage.

News Catalog provides two features, Get Headlines and Retrieve Article. The features of the Logging service are Log and Get Log. The News service provides the Get News and Get Article features. It also has a Cache feature, which is used in the implementation of Get Article. The intent of this feature is to avoid charging a user more than once for retrieving the same article, and to speed up the retrieval of full text articles. It maintains a single cache of the most recently retrieved articles. When a client requests an article, the Get Article feature executes. If the article is not in the cache, an external request is made to the News Catalog service, and the article is retrieved and subsequently cached. If the article is in the cache, then it is immediately returned to the client, and no external request to the News Catalog service is made.

As described, News Catalog relies on the information provided in the log maintained by Logging. As part of the usage contract between News Catalog and a service consumer such as News, the service consumer must record all article requests made by the client with Logging. News Catalog relies on this log for charging clients for their usage. However, the News service also contains a Cache feature, which saves recently accessed articles. When a user requests to read an article that happens to be saved in the cache, the article is retrieved from the cache instead of from the News Catalog.

A manual analysis of this scenario tells us the only those client requests are being logged for which the requested article is not found in the cache. In the article is found in the cache, the requests are not being logged. What is worse is that no one is charged for those requests except the very first client that caused this article to be cached. So it is possible to gain access to an article for free (as long it stays in the cache). This is an example of a feature interaction due to an incorrect *invocation order*. The behavior of the service depends subtly on when requests are logged. This could occur either before (no interaction), or after the cache is inspected (interaction).

The situation demonstrated here also demonstrates another type of undesirable feature interaction between services, which occurs due to the *information-hiding* nature of web services. 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, it turns out) expects that all requests for articles made by the clients of the consumer services will be logged with the Logging service associated with the News Catalog. It does not expect that these consumer services have mechanisms that will prevent some of the requests from being logged.

4 Formal Analysis of Feature Interactions

For larger composite web services, which may exhibit many potential feature interactions, the manual analysis described for the example is not feasible. It is, therefore, desirable to perform the detecting problematic interactions using formal approaches. Our approach is based on Labeled Transitions Systems (LTS), 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) [3]. The LTS Analyzer (LSTA) [3] supports the analysis of a system described in FSP to verify that the LTS model satisfies 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.

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 supports action non-determinism and process parameters. In the LTS analysis that follows, our goal is to detect the invocation order problem given a model of the News service in FSP, and a definition of safety properties that have to hold. Fig. 1 shows an FSP model of the News service features.

```
// Logging service
LOG = (log_request-> LOG | get_log-> LOG).

// News Catalog service
CATALOG = (get_article-> CATALOG | proccess_billing-> BILLING),
BILLING = (log.get_log->process->CATALOG).

// Cache feature for the News service
CACHE = (add_article->NON_EMPTY_CACHE),
NON_EMPTY_CACHE = (add_article->NON_EMPTY_CACHE |
 retrieve_article->NON_EMPTY_CACHE).

// The News service
NEWS_SERVICE = (request_article-> CHECK_CACHE),
CHECK_CACHE = (cache.found->cache.retrieve_article->NEWS_SERVICE
 | cache.not_found->ACCESS_CATALOG),
ACCESS_CATALOG = (log.log_request->catalog.get_article->
 cache.add_article->NEWS_SERVICE).
```

Fig. 1. FSP model of the News service features

The main concern from the point of view of correct logging is, that for every article request there should be an invocation of the Logging service, which will log that article request. We express this concern as a safety property as shown in Fig. 2.

```
// Check that each request for an article is logged.
property P = (request_article->log.log_request->P).
```

Fig. 2. Safety property for the News service

Finally, the composite process that represents the News Catalog service and the News service interacting is expressed simply as shown in Fig. 3

```
// Composite process that represents the News service
||NEWS = (NEWS_SERVICE || cache:CACHE || catalog:CATALOG ||
 log:LOG || P).
```

Fig. 3. Result of composing the News Catalog and the News service features

The composite process includes the safety property `P`. 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 a client request for a news article is not properly logged. Performing this safety check, we get the result in Fig. 4.

```
Trace to property violation in P:
     request_article
     cache.not_found
     log.log_request
     catalog.get_article
     cache.add_article
     request_article
     cache.found
     cache.retrieve_article
     request_article
```

Fig. 4. Result of safety check analysis

The trace of events demonstrates that there is a situation, where the safety property is violated. This happens when a client requests an article, and then another client requests the same article. Since the article was cached after the first request, the next time it was retrieved from the Cache instead of from the News Catalog service. This analysis reveals that the cache feature of the News service can cause problems, when it is paired with the incorrect order of the Logging service invocation.

The graphical representation of the News service can provide us with additional insight in the scenario that led up to the feature interaction. Fig. 5 shows the LTS for the News service, indicating (in red) the violating transition (`request_article`).

Although our focus in this paper was on order of invocation types of interactions, the situation demonstrated at hand can also be characterized as an undesirable interaction that occurs due to the encapsulation or information hiding nature of web services. The essence of the problem from the point of view of the News Catalogue service is that it is unaware of the Cache feature present in one of its service consumers, the News service. It expects that all article requests made by the clients of the consumer services will be logged with the Logging service associated with the News Catalogue. It does not expect that these consumer services have mechanisms that will prevent some of the requests from being logged. This is an example where an interaction can be described as both a functional, as well as a non-functional feature interaction.

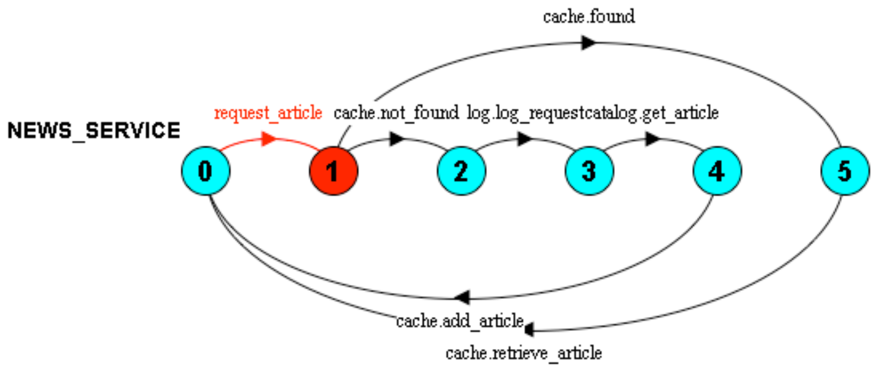

Fig. 5. LTS for the News service

5 Conclusion

LTS modeling of web services can be a significant aid in developing a formal methodology for functional feature interaction problems detection. Rather than modeling the feature interactions explicitly, it was only necessary to model the main aspects of behaviour of each web service, and to define safety properties that detect assumption violations, race conditions, and incorrect order of invocation types of interactions.

One advantage of using this approach is that we can detect potential feature interactions before web service implementation. Any undesirable interaction that is identified at this stage saves us the cost of correcting those errors during the implementation. By developing LTS models of web services and instrumenting them with appropriate safety and progress properties we are able to detect interactions automatically using the LTSA tool. LTSA can not only detect the presence of a feature interaction problem, but also find the exact sequence of events that leads to the problem.

A well-known problem with LTS analysis is the explosive nature in which the number of states increases. This may limit the size of systems that can be analyzed, even after optimizations such as state hiding [3] have been applied. Another issue is that the safety and progress properties have been developed to target specific known interaction problems. However, currently there is no systematic process for developing these properties to detect unknown interaction problems. Here, we rely on the completeness of our understanding of the types of interactions that can occur. Also, algebraic notations such us the pi-calculus should be explored.

Future work will include the application of the approach to larger case studies. For example, we are currently investigating its application to a supply chain. The potential interactions in these kinds of systems are significantly more complex, in particular, because we are not focusing on isolated interactions, one at a time. In our earlier work, results were also determined for the other classes of feature interactions problems (that is, race conditions, violation of assumptions, competition for resources, goal conflicts and information hiding), but mostly in isolation. We are also interested in extending the work on offline detection to the detection of feature interactions at runtime, and ultimately, their runtime resolution. This would be of great relevance to situations where web services are dynamically composed.

References

1. Foster H., Uchitel S., et al, Compatibility Verification for Web Service Choreography, International Conference on Web Services (ICWS), IEEE, 738-741, 2004.
2. Keck, D, and Kuehn, P., The Feature and Service Interaction Problem in Telecommunications Systems, IEEE Transactions on Software Engineering, 779–796, 1998.
3. Magee, J., and Kramer, J., Concurrency: State Models and Java Programs, Wiley, 1999.
4. Pulvermüller, E., Speck, A., et al (2001), Feature Interaction in Composed Systems, ECOOP Workshop on Feature Interactions in Composed Systems, Technical Report 2001-14, 1-6, Universität Karlsruhe, Fakultät für Informatik.
5. Turner, C.R., Fuggetta, A., et al, A Conceptual Basis for Feature Engineering, Journal of Systems and Software, 49:1, 3-15, December 1999.
6. Weiss, M., and Esfandiari, B., On Feature Interactions among Web Services, International Conference on Web Services (ICWS), 88-95, IEEE, 2004.
7. Weiss, M., and Esfandiari, B., On Feature Interactions Among Web Services, International Journal of Web Services Research, 2(4), 21-45, October-December, 2005.
8. Weiss, M. and Esfandiari, B., Towards a Classification of Web Service Feature Interactions, International Conference on Service-Oriented Computing (ICSOC), LNCS, Springer, 2005 (to appear).