Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 359–381
Computer Networks
www.elsevier.com/locate/comnet

Towards a classification of web service feature interactions

M. Weiss *, B. Esfandiari, Y. Luo

School of Computer Science, Carleton University, Ottawa, Ont., Canada K1S 5B6

Available online 14 September 2006

Responsible Editor: H. Rudin

Abstract

The rapid introduction of new web services into a dynamic business environment can lead to undesirable interactions that negatively affect service quality and user satisfaction. In previous work, we have demonstrated how such interactions between web services can be modeled as feature interactions. In this paper, we outline a classification of web service feature interactions. The goals of this classification are to understand the scope of the feature interaction problem in the web services domain, and to propose a benchmark against which to assess the coverage of solutions to this problem. As there is no standard set of web services that one could use as examples, we illustrate the interactions using a fictitious e-commerce scenario.

Keywords: Feature interaction; Web services; Classification

1. Introduction

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

Feature interactions are interactions between independently developed system units (features), which can be either intended (as in a uses or extends relationship between features), or unintended and result in undesirable side effects. The *problem of feature interactions* has first been formally investigated in the telecommunications domain [5]. It concerns the coordination of features such that they cooperate towards a desired result at the application level. However, the feature interaction problem is not limited to telecommunications. The phenomenon of undesirable interactions between components of a system can occur in any software system that is subject to changes. Here, we consider the occurrence of feature interactions in web services as first described in [19].

* Corresponding author.
E-mail addresses: weiss@scs.carleton.ca (M. Weiss), babak@sce.carleton.ca (B. Esfandiari), rayluo@rogers.com (Y. Luo).

1389-1286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2006.08.003

Interaction is, of course, the very foundation of a service-oriented architecture. Web services must interact, and useful web services will "emerge" from the interaction of more specialized services, as higher-level services are composed from lower-level services. As the number of web services available increases, their interactions will also become more complex. Many of these interactions will be intended, but others may be unintended, and we need to prevent their side effects from occurring. As noted in [15], a large number of these side effects are related to security and privacy concerns.

Unlike in traditional telecommunication systems, where operators oversee the integration of services from different vendors and act as "central authorities" [13] for managing feature interactions, systems using web services are largely built from third-party services, over whose implementation service users have little control. Instead of a few operators who perform the task of coordinating features, the large number of service users and small- and medium-sized service providers cannot handle interactions in the same manner (i.e., by wielding their power over the service providers), and the problem is greatly exacerbated. A further significant differentiator is the trend towards the automated discovery and composition of web services for creating applications on demand.

This paper builds on our previous work on web service feature interactions [20,21] by providing a classification of interactions by their nature and causes, following a similar approach as Cameron et al. [6] in their classic survey on feature interactions in telecommunications systems. We also propose a unified, realistic, and quite generic case study (the "Amazin" virtual bookstore) that illustrates all of the causes, while remaining technology-agnostic and easily translatable to other domains. We believe that this case study can be used as a benchmark for future studies on feature interactions in web services. The paper expands on the presentation in [22] by making significant changes to the organization and discussion of the interaction examples, providing more background on the case study and its implementation, and exploring the notion of feature and feature interaction in the context of web services.

While in [20,21] we had hand-crafted our examples in order to high-light the potential for feature interaction in a composite web service, in this work we use a controlled approach in which features are first described individually, and without consideration regarding their possible participation in feature interactions. The feature interactions that we can observe only "emerge" from the composition of the services for the scenarios in the case study. We believe that this approach strengthens our claims with respect to the pervasiveness of the feature interaction problem in web services.

The paper is organized as follows. Section 2 gives an introduction to the feature interaction problem as it applies to web services, and summarizes our approach to modeling web services in terms of features. As there is no standard set of web services that one could use as examples, Section 3 introduces our case study of a fictitious virtual bookstore. Section 4 presents our classification of web service feature interactions by their nature. This section also describes the feature interactions examples from the case study. Section 5 provides a classification of web service feature interactions by their cause. The paper concludes with a discussion and an outlook on future research in Section 6.

2. Feature interaction problem and feature modeling

2.1. Feature interaction problem

The first generation of web services did not exploit the benefits of a "web of services".[1] They were either of a simple, non-composite nature (often information services, such as a stock quote lookup service), or provided access to application functionality over pre-existing business relationships. By contrast, the current generation of web services are typically composite (i.e., they are constructed from other, more primitive web services), and offered by third-party service providers, and thus not grounded in existing relationships.

First generation web services were predicated on two implict assumptions: (1) that services developed in isolation would either be used in isolation, or, if part of a composite service, would not interact in unexpected ways, and (2) that users had full control over the services they used, or there was a common understanding of the operation and side effects of

[1] The concept of a "web of services" refers to the notion that the key characteristic of web services is not the use of web technology, but their potential for providing universal interoperability, manifested by the web-like network of services created by the composition of lower-level web services into higher-level web services.

those services. We argue that these assumptions are no longer valid for current web services.

Consider the example of a word-processing service that uses two third-party services, hyphenation[2] and formatting [20]. Assume that the user has set her language preference for the word-processing service to UK English. However, let us also assume that, hidden to the word-processing service, the formatting service itself incorporates a hyphenation service (not such an unrealistic assumption). This time, the formatting service does not specify a language preference to the hyphenation service. Suppose that the hyphenation service uses US English hyphenation rules by default. The result of the service composition is that the hyphenation will be changed as part of formatting.

This is a case of an undesirable feature interaction. As noted, the concepts of feature and feature interaction originated in the telecommunication domain, but the concepts are quite general. A *feature* is the minimum user-visible service unit [11]. Features are often independently developed and deployed. A *feature interaction* occurs when a feature invokes or influences another feature directly or indirectly. Although many of these interactions are intended (even required), other interactions can lead to undesirable side effects a such as an inconsistent system state, or data inaccuracies. Next, we discuss how the notions of feature and feature interaction can be applied to web services. This is followed by an overview of our feature modeling approach.

2.2. Modeling web services in terms of features

Our approach [21,25] is to model web services as a set of one or more features. Each feature is a bundle of closely related *operations* that can be invoked through the web service. Web services can, themselves, be composed from lower-level web services, resulting in hierarchy of services. The UML diagram in Fig. 1 shows the relationship between web services and features in our definition. It follows the well-known composite design pattern [8].

This model of web services retains the key aspects of the current standards [1] for describing and composing web services, namely WSDL (web services description language) and BPEL (business process execution language). A WSDL specification describes a web service in terms of schemas, messages, operations and bindings (protocols to use, e.g., SOAP). Operations are grouped into port types, and a service is a set of ports, each of which associates a service endpoint (address through which the service can be invoked) with a port type (i.e., the operations the service implements), and a binding.

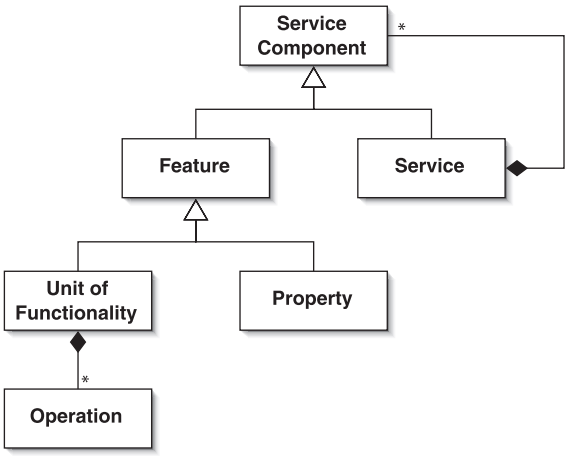


Fig. 1. Relationship between web services and features.

A BPEL specification represents a composite web service. It defines a (partial) order in which operations on the component services are invoked (i.e., a process), a set of roles (service providers) that provide these services, and the port types that need to be supported by those roles. The composite service can again be exposed through a WSDL specification. Features are, thus, units of functionality that bundle the operations a service provider exposes to (potential) service users. Web services are compositions of features (the features they provide), as well as other web services (the features they require).

In a UML model, features can be represented as interfaces, where operations become methods. Fig. 2 shows the components of the word processing service. They include one service user (Word Processor), two service providers (Formatter and Hyphenator), and two features (IFormat and IHyphenate). In this example, each provider only implements one interface, but in general it could implement several. For example, Formatter implements the IFormat interface with a format method. The Word Processor uses the IFormat and IHyphenate features. The Formatter also uses the IHyphenate feature.

While two features can have the same functionality, they will likely differ in terms of their

[2] Thanks to K. Turner for suggesting the hyphenation service to make the example more compelling than the spell-checking service we had used originally.

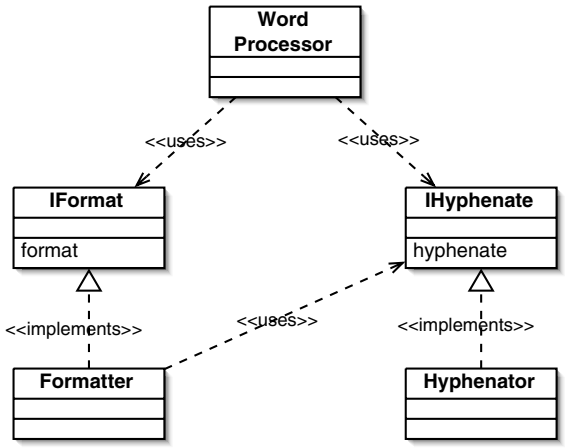


Fig. 2. Components of the word processing service.

non-functional impact on the system. In order to incorporate *non-functional properties* of services into our model, we extend our earlier definition of feature to include service properties. A feature is now either a bundle of operations with a common purpose (such as order processing), or a non-functional property of the service (such as privacy). We also refer to the former as functional and to the latter as non-functional features. This extended definition is reflected in the two subclasses of Feature in Fig. 1. The distinction between features as functional units and properties was first made by [14]. There is no way to model non-functional properties in UML, so we introduce a graphical notation to represent them in Section 2.3.1.

The definitions of services and features provided is consistent with our use in the case study below. While we deem it fairly general, and certainly applicable beyond the scope of the case study, other authors will without doubt identify different feature models, designed to capture other aspects such as variability or traceability. One such approach based on feature engineering is [7]. In alignment with our definition, it proposes to model services as groups of features, and features in terms of service interfaces. However, it does not model non-functional service properties. The purpose is also different: there, feature models are used to extract service interfaces from legacy applications.

2.3. Feature modeling approach

Our approach focuses on the requirements stage. The feature modeling approach uses the *User Requirements Notation* (URN) [2] to model the intentional and behavioral aspects of the web services and their composition, and *Finite State Processes* (FSP) [12] for their formal specification and the validation of safety and progress properties. These notations are complementary: URN is a visual notation that allows designers to describe goals and scenarios; and FSP brings the established benefits of process algebra to the analysis of distributed systems and feature interaction detection. In this section we can only provide an overview of the approach; see [21,25] for further details.

Goals, scenarios and processes describe feature *intent* and *behavior* at different levels of abstraction. Intentional aspects of web services (the *why*) are modeled in the form of goals (both functional and non-functional in nature). Their behavioral aspects (the *what* and *when*) are modeled as scenarios. Service providers are represented as actors or components. Finally, detailed aspects of service behavior are modeled in the form of processes. The interaction of features is captured in the form of links between goals at a high-level, as well as in the form of safety and progress properties at a low-level.

These models allow us to reason about feature interactions, and to document detection and resolution strategies. In our approach, we progress from higher-level to lower-level models. First, the intent and side effects of a feature are modeled as URN goals, and their high-level operation in the form of URN scenarios. The interaction of features is captured in the form of links between goals. Then we describe the allocation of features to system URN actors or components, and the relationships between these actors. Scenarios are then refined at a greater level of detail in FSP, which enables their validation using tools. Features are represented as processes, and their interaction as process composition, and undesirable feature interactions as property violations.

Fig. 3 summarizes the stages of our feature modeling approach. The arrows indicate the idealized information flow between the stages. Iteration is not shown in the diagram, although the process is highly iterative, and some stages can be skipped. The output of the three requirements stages is fed into the design stage where a UML as outlined in Section 2.2 will be produced. Actual service design and implementation is outside the scope of this approach.

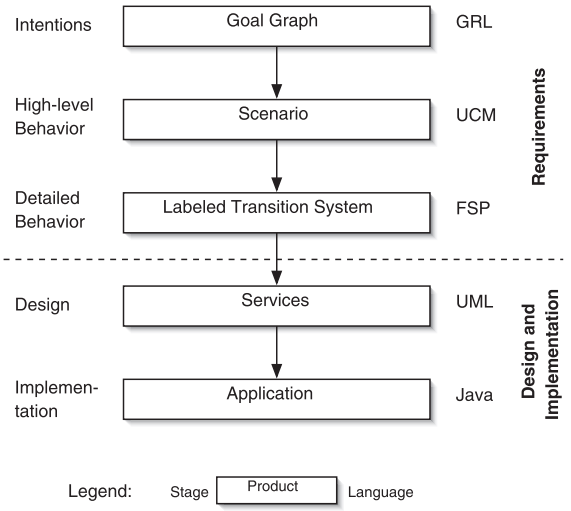


Fig. 3. Stages of the feature modeling approach.

2.3.1. User requirements notation (URN)

URN contains two complementary notations: the *Goal-oriented Requirements Language* (GRL) [9], and the *Use Case Map* (UCM) notation [17].

In GRL, requirements are modeled as *goals* to be achieved by the design of a system. It builds on well-established goal-oriented analysis techniques. The main elements of the notation are summarized in Fig. 4. Softgoals represent non-functional requirements, their shape suggesting that there are no clear-cut criteria for achieving them. Functional requirements are represented as (hard) goals. Tasks are solutions that achieve goals or softgoals. Resources are entities that are required to perform a task or goal. During the analysis, a set of initial goals is iteratively refined into subgoals. These goals and their refinement relationships form a *goal graph* that shows the influence of goals on each other, and can be analyzed for goal conflicts.

The perspectives of different stakeholders (actors) can also be described in GRL. For each stakeholder we model their goals, as well as their dependencies on one another to achieve those goals. These goals of one stakeholder can now also compromise the goals of other stakeholders, resulting in a *conflict*. Conflicts indicate potential feature interactions. One feature may subvert or "break" the intent (goals) of other features. Where there is a choice between different means (i.e., features or feature implementations) to achieve a goal, the objective of the analysis is to determine those that resolve goal conflicts in a way that best satisfies the initial goals of all stakeholders.

The UCM notation provides a way of describing scenarios without the need to commit to system components. The main elements of the notation are summarized in Fig. 5. A *scenario* is a causally ordered set of responsibilities that a system performs. Responsibilities can be allocated to components by placing them within the boundaries of that component. This is how we will be modeling feature deployment. With UCMs, different structures suggested by alternatives that were identified in a GRL model can be expressed and evaluated

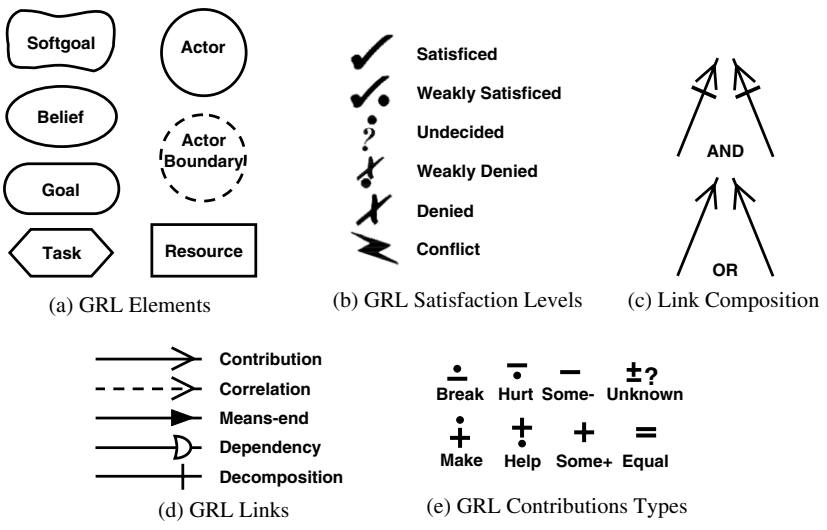
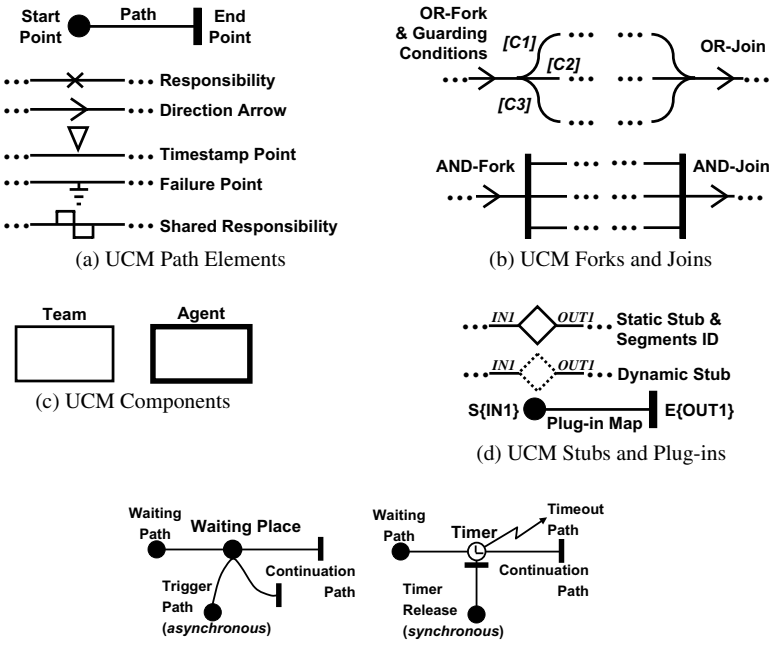


Fig. 4. Summary of the goal-oriented requirements language (GRL).



(a) UCM Path Elements

(b) UCM Forks and Joins

(c) UCM Components

(d) UCM Stubs and Plug-ins

(e) UCM Waiting Places and Timers

Fig. 5. Summary of the use case map (UCM) notation.

by moving responsibilities from one component (which is the UCM equivalent of a GRL actor) to another, or by restructuring components.

The UCM and the GRL perspectives are complementary. The UCM perspective allows us to refine the goals identified in a GRL model into greater detail (by refining them into tasks/responsibilities), as necessary. Generally, when creating these models we would iterate between both views. That is, we cannot decide on the allocation of goals to actors simply within a GRL model, but only after repeatedly refining both the GRL and UCM perspectives.

At the higher-level of abstraction, as described in this subsection, feature interaction analysis iterates over the following three steps:

1. Model the features to be analyzed as a GRL goal graph. Goal graphs allow us to represent features, and to reason about conflicts between them.
2. Analyze the goal graph for conflicts. Conflicts point to possible feature interactions, in particular, if a conflict "breaks" expected functionality.
3. Resolve the interactions. During this step, UCM models allow us to explore the different alternatives suggested by the GRL models.

2.3.2. Finite state processes (FSP)

The hierarchical architecture inherent in SOA 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 also desirable to develop formal approaches to modeling web services and detecting problematic interactions automatically.

Finite state processes (FSP) [12] is an algebraic notation for describing labeled transitions systems. The main elements of the notation are summarized in Table 1. 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. Their behavior can be analyzed using well-established model checking techniques based on state-space exploration.

The analysis involves the validation of safety and progress properties. Informally, a *property* is an attribute of an LTS that is true for every possible execution of that LTS. 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.

Table 1
Subset of the finite state processes (FSP) notation

Construct	Notation
Action prefix →	$(x \rightarrow P)$ describes a process that initially engages in the action x and then behaves exactly as P
Choice |	$(x \rightarrow P \mid y \rightarrow Q)$ describes a process which initially engaged in either of the actions x or y
Parallel composition ‖	$(P \| Q)$ models the concurrent execution of P and Q
Process labeling:	$a{:}P$ prefixes each label in the alphabet of P with a
Safety property	Property asserts that any trace including actions in the alphabet of P is accepted by P
Progress property	Progress $P = a1, \ldots, an$ asserts that at least one of the actions $a1, \ldots, an$ will be executed

For example, a safety property that defines an expected sequence of transitions enables us to detect the order of invocation interactions. 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.

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. A *structure diagram* is a visual representation of FSP model. In a structure diagram, processes are represented as boxes, and externally visible actions are shown as circles on the perimeter of the box. Shared actions (i.e., actions that two processes need to execute simultaneously) are shown as lines connecting two action circles.

At the lower-level of abstraction, the analysis of a composite web service for feature interactions iterates over the following steps:

1. Model the salient parts of the behavior of each component service as a process, and define properties that detect specific feature interactions.
2. Analyze the model of the composite service, which comprises instances of the features, and any safety properties we want to validate.
3. Resolve each detected feature interaction, and update the FSP model accordingly, and thus in an iterative manner eliminate interactions.

3. Case study: A virtual bookstore

As there is no standard set of web services that one could use as examples, we illustrate the interactions using a fictitious virtual bookstore. From a methodological point of view, we do not want our design of individual services to be aware of other services they may be used together with. Therefore:

1. We first describe the *individual web services* that we will use. These services are developed without knowledge of how they will be composed later. Often they include third-party services that provide certain supplementary functionality such as identity management or payment.
2. We then create a *composite service* for an virtual bookstore from these services. We analyze the feature interactions that can occur as a result. As the services have been implemented independently, they may embody assumptions that cause unexpected behavior as the services are composed.

This purpose of this division is to reproduce the problems that can result in the actual development of service-based applications. Each web service feature is implemented with developers making assumptions that are individually valid, i.e., they faithfully implement the service interfaces. Feature interactions only result when these features are combined, often in unanticipated ways.

While clearly, some of these interactions can be anticipated by the service designers from past experience, it is impossible to plan for all potential interactions, including with services and features to be developed in the future. In our example, we therefore do not try to anticipate possible interactions during service design, but focus on implementing the specified interfaces.

3.1. Examples of services and their features

The following web services have one aspect in common: they all focus on one narrow type of

functionality, and are usually employed in a supporting role. Examples of such *supplementary services* are identity management, payment processing, and shipping. In principle, any of these services could be provided by the service user (such as the virtual bookstore), but usually at a significant development cost, or risk of poor quality (and thus liability issues).

3.1.1. iPassport

An identity management service simplifies authentication with multiple service providers. It allows service requesters to authenticate themselves once with one service provider (known as single sign-in), and to access other service providers linked to the initial service provider through a circle of trust. It simplifies the implementation of service providers, as well, because they no longer need to provide their own authentication component.

Fig. 6 is a GRL model of the iPassport service. It models the service provider, as well as each type of requester as an actor (circle). As shown, iPassport mediates between Requesters [Service] and Providers [Service], and thus acts in the role of a broker. Requesters [Service] use iPassport to manage their profiles through the Manage Profile feature, while Providers [Service] can authenticate users and access their profiles through the Authenticate and Access Profile features.

Service provisioning relationships are modeled as goals (rounded rectangles) that an actor wants to achieve with the help of another actor. For example, the Requester [Service] (depender) depends on iPassport (dependee) to achieve the Manage Profile goal. Non-functional needs, which cannot be achieved in an absolute manner, are specified as softgoals (clouds). From the diagram, iPassport ensures the requester's Legitimacy. Resource dependencies (rectangles) represent physical or informational entities that must be available.

Internally, the Identity Management goal of the iPassport actor is decomposed into two functional components: Authentication and Profile Management. Authentication is responsible for verifying a user's login information, and creating a delegation proxy (also known as a "ticket") that can substitute for an explicit login in order to access a service. Profile Management is responsible for storing profiles, and giving access to profile information. These functional components are shown as tasks (hexagons), which specify ways of achieving a goal.

Decomposition is used to represent how a service is composed from other internal or external functional components. External functionality is provided through a feature from another service, and indicated via a dependency link in the GRL model. In the context of modeling web services, a goal dependency should be read as a functional feature provided by one service (dependee) to another service (depender). It can be subsequently mapped into a service interface. A softgoal dependency indicates an associated service property.

More insight into the sequencing of service invocations (i.e., the temporal relationship of tasks), and deployment aspects of features (the *what*, *when* and *how*) can be gained from a UCM model. It can be derived from a GRL model by refining each tasks into one or more responsibilities, and adding information about their execution sequence. Fig. 7 shows the UCM model for iPassport. Note that this is only

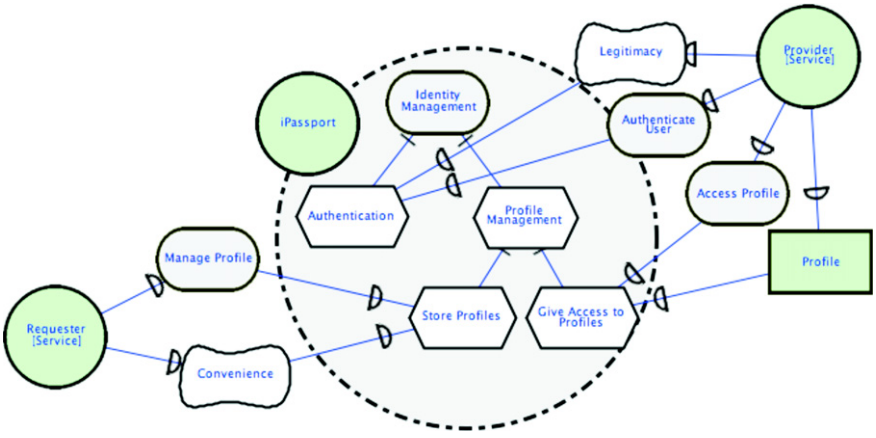


Fig. 6. GRL model of the iPassport service.

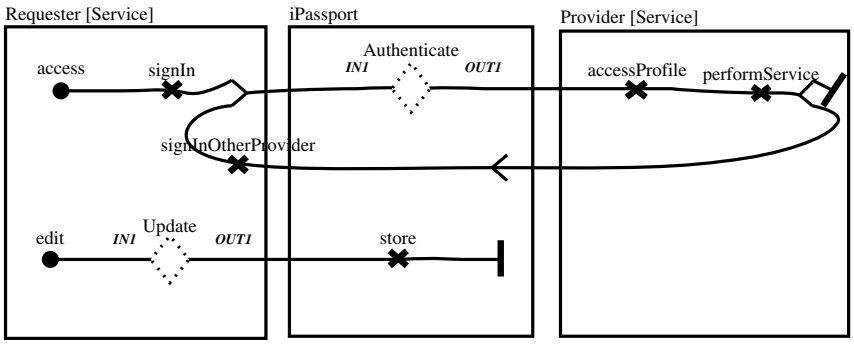


Fig. 7. UCM model of the iPassport service.

a top-level model with placeholders (also known as stubs) for submaps that define the Authenticate and Update behaviors.

This model captures that service requesters, having signed up with a participating service provider, are authenticated by iPassport each time they want to access a service. A Provider[Service] can subsequently access the requester's profile (accessProfile) for information required to perform its service. This may involve other service providers, upon which a delegation proxy is passed to these providers, automatically signing in the provider without the user's involvement (signInOtherProvider). This detail is added in the UCM model, and was not explicit in the GRL model. In the diagram, crosses represent responsibilities, filled circles start points, and bars end points of paths.

3.1.2. PayMe

A payment processing service allows payers to make secure payments online, and simplifies credit card processing for payees, while contributing to increased sales for them. As shown in Fig. 8, PayMe provides two service interfaces (features): one to the Payer [Order] to Manage Accounts, and one to the Payee [Order] to receive payment for an order (Process Payment). The Process Payment feature includes functionality to submit payment details, as well as to cancel payments. The non-functional properties of the PayMe service (Security and Increase Sales) are again modeled as softgoals.

The PayMe service provides separate interfaces for accessing buyer and seller accounts. For example, buyers can only Deposit, while sellers can Withdraw and initiate a Transfer between accounts. The Security goal is achieved by not disclosing buyers' payment information such as credit card numbers or bank accounts to sellers. Sellers achieve the Increase Sales goal by offering buyers a payment option that is less risky than credit card transactions.

3.1.3. ShipEx

A shipping service provides shippers with guaranteed delivery of product, and simplifies tracking

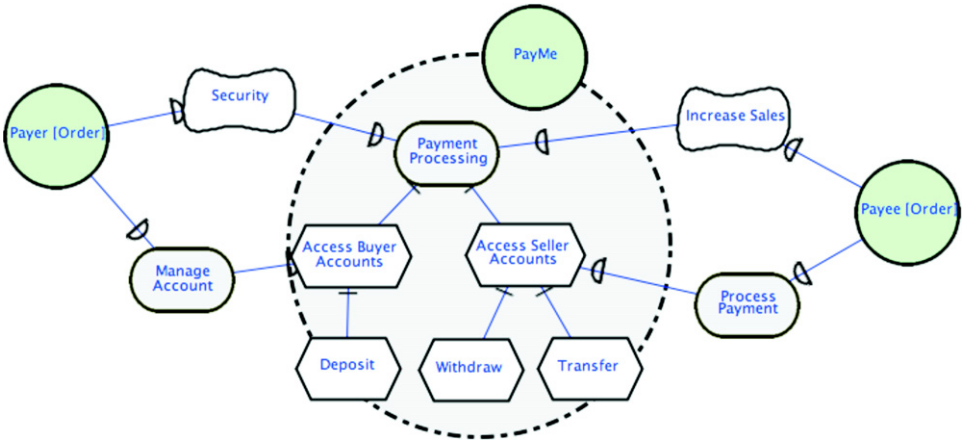


Fig. 8. GRL model of the PayMe service.

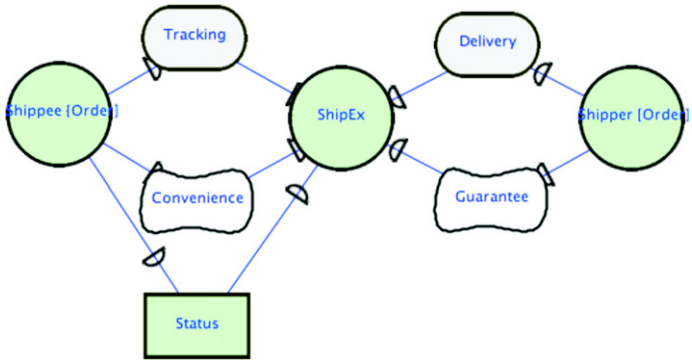


Fig. 9. GRL model of the ShipEx service.

of a shipment for shipees. As shown in Fig. 9, ShipEx offers a Delivery feature to the Shipper [Order] with functionality for initiating shipment of an order, and canceling shipments, as well as a Tracking feature for the Shippee [Order] to check on the status of a shipment.

We do not show the internal structure of this service to save space. However, the required steps are similar to those for the earlier examples.

3.1.4. Shark

Caching improves performance by storing the results of previous requests, thus reducing the number of service requests that are made. As shown in Fig. 10, the Shark proxy service provides a Caching feature through which a Consumer [Content] can cache the results of popular service requests.

3.1.5. Prototype

We implemented prototypes of these features using the Glue web services framework [18], and tested them independently. Then we combined the features into composite services, and analyzed the result for feature interactions. The largest of these case studies (a virtual bookstore) will be described in Section 3.2. Although space does not allow us to go into great detail on the implementation, a brief overview will be provided below.

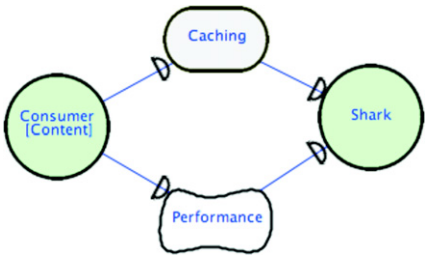


Fig. 10. GRL model of the Shark service.

Fig. 11 shows a UML model of the implemented features. As shown, there is a direct correspondence between service interfaces in the UML model and goal dependencies in the above GRL models. For example, the GRL model of the ShipEx service in Fig. 9 has goal dependencies Tracking and Delivery. These are represented in the UML model as the interfaces ITracking and IDelivery, together with the operations exposed through these interfaces.

Service operations can be discovered either from the textual description of the features, or from the UCM models. In [23], we outline an approach for systematically deriving service interfaces from UCMs representing business process models. The basic approach is to extract messages exchanged between UCM components that correspond to causal paths in a scenario. These messages are then grouped into operations according to message exchange patterns (such as request-reply), and related operations into service interfaces. The names of operations can often be based on the names of responsibilities.

3.2. Composite service: virtual bookstore

As a case study, we created the Amazin virtual bookstore, which gives Customers access to its virtual catalog, and the option to order books from the catalog through an Order Processing feature. The actor diagram for the Amazin service is shown in Fig. 12. This diagram only shows the relationships between Amazin and other actors (service users and providers), not those between the other actors themselves. For clarity, these will be shown in separate diagrams.

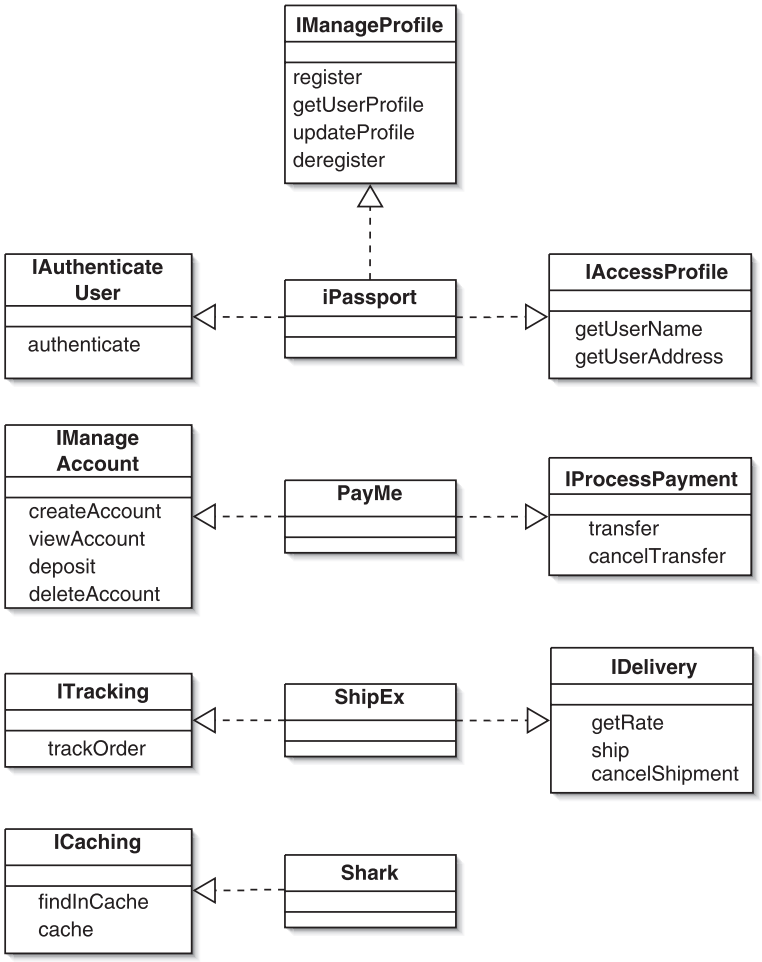


Fig. 11. UML model of the implemented feature prototypes.

3.2.1. Amazin

Amazin relies on a number of Suppliers to fulfill customer orders. Customer logins are handled through the iPassport identity management service, which provides an Authenticate User and an Access Profile feature. On receiving a customer order, Amazin authenticates the customer, and accesses the customer's profile. It then selects a Supplier which stocks the ordered book and invokes its Order Processing service, passing along the customer's identity.

An internal structure of the Amazin service that fits this description is also shown in Fig. 12. This design makes assumptions that, while in agreement with service interfaces, may cause feature interactions. One potential source of interactions is a performance optimization: in order to improve order throughput, Amazin caches copies of popular ebooks, which can be stored electronically. Doing so may result in the supplier of the ebook not getting paid.

A non-obvious aspect of this diagram is the specification of alternative ways of achieving the Place Order goal. If the order is for an ebook, the Order eBook task will be selected; otherwise, the service proceeds with Order Book. Ordering ebooks and regular books are handled differently, since an ebook can be cached. It is in this way that the alternatives Check Cache and Submit Order for achieving the Order eBook goal need to be evaluated. We should first try Check Cache, and execute Submit Order only if the ebook is not in the cache.

It should be pointed out that such alternatives are optional. We can remove all but one alternative, and will still have a functional service. The Amazin service without caching of ebooks (i.e., the only

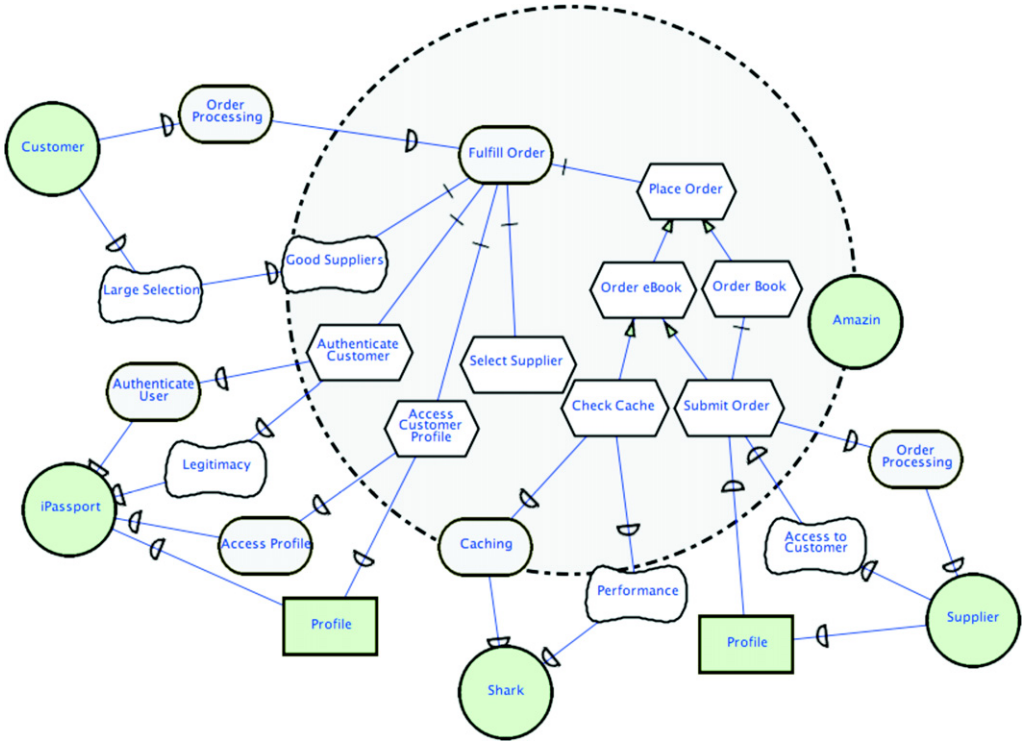


Fig. 12. GRL model of the Amazin virtual bookstore.

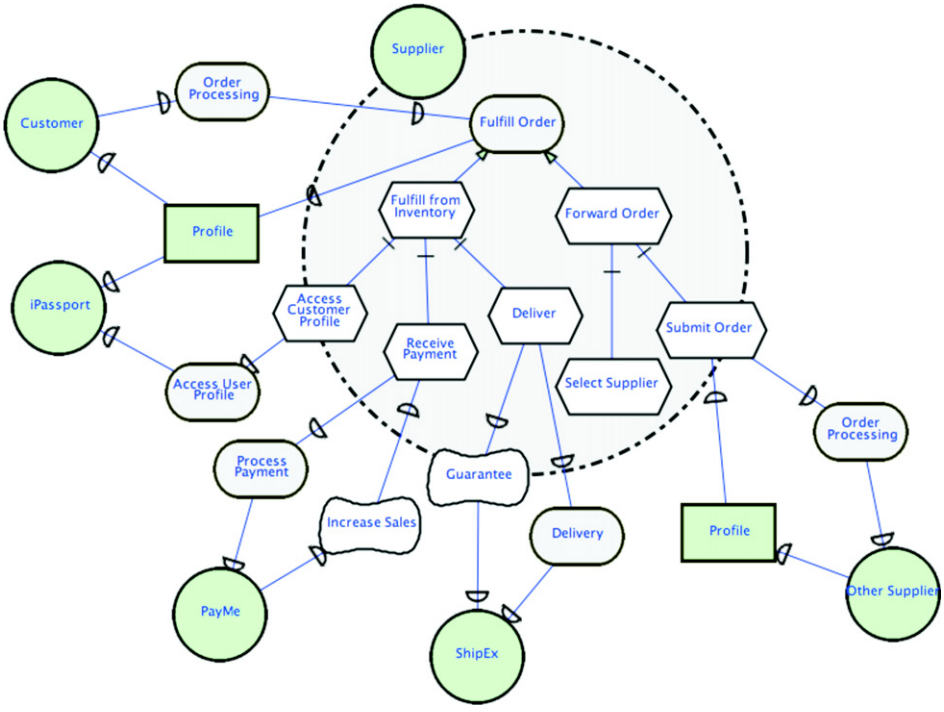


Fig. 13. GRL model of the Supplier service.

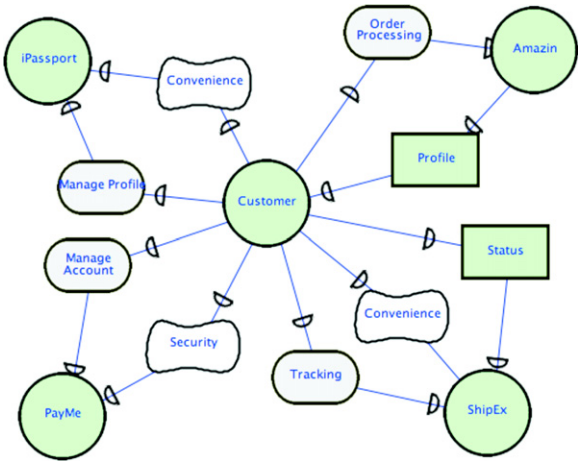


Fig. 14. GRL model of the Customer application.

option to Place Order is to Submit Order) works, although access to popular content will not be optimized. We will return to that observation in the discussion of Example 2 below.

3.2.2. Supplier

Suppliers themselves rely on other services. Fig. 13 shows the internal structure of the Supplier service, documenting its fulfilment process, and its relationships with other services. The Supplier determines the availability of an ordered book, and, if successful, obtains the Customer's payment and shipping preferences from the iPassport service. It then invokes the Process Payment feature provided by the PayMe financial service provider, and the

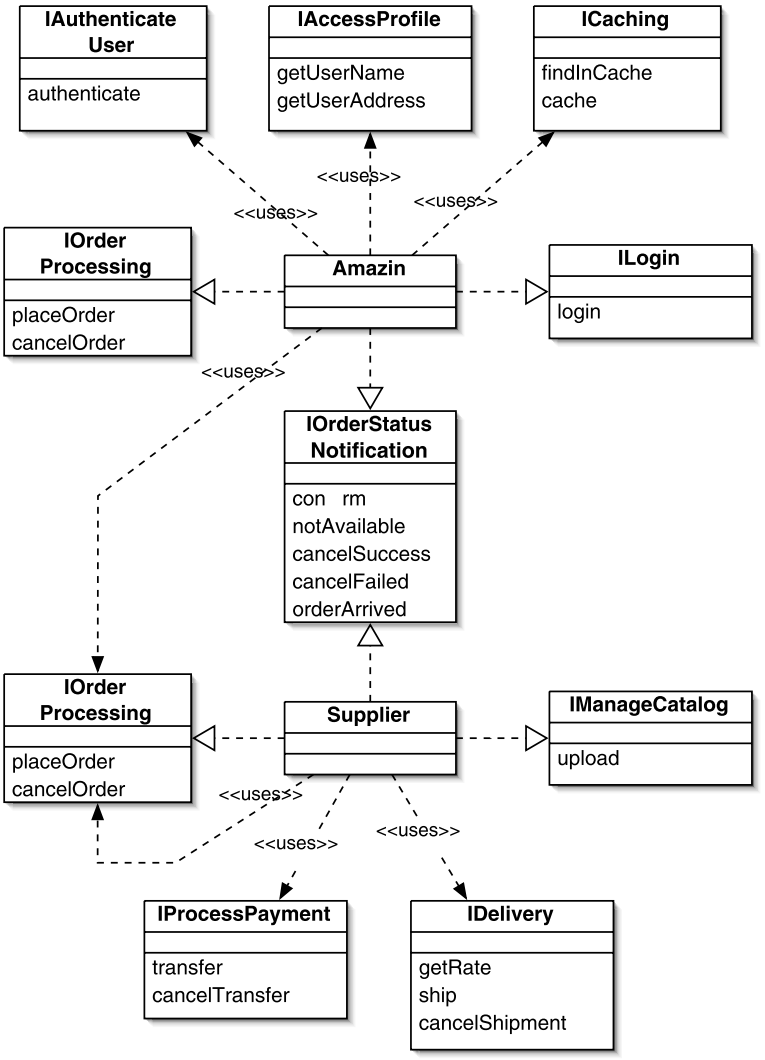


Fig. 15. UML model of the Amazin and Supplier services.

Delivery feature of Amazin's ShipEx fulfillment partner.

If a Supplier cannot fulfill an order, it will attempt to satisfy it from its network of Other Suppliers. Although the corresponding service relationships have been omitted from the diagram, the chosen Other Supplier will use the same payment processing and shipping services as Supplier. Accordingly, Fig. 13 indicates two means to achieve the Fulfill Order goal: Fulfill from Inventory or Forward Order. It is interesting to view these alternatives as service components that have been added at different stages of evolution of the service. Here we suppose that the initial version of the Supplier service could only Fulfill Orders from inventory, and later the service was changed to Forward Orders that could not be locally satisfied. This observation is further discussed in Section 5.4.

3.2.3. Customer

Customers can track the progress of their orders via the Tracking service. They can also manage their profiles (Manage Profile), and accounts (Manage Account) through the iPassport and PayMe services, respectively. The service relationships of the Customer application are shown in Fig. 14.

3.2.4. Prototype

Fig. 15 shows a UML model of the implemented composite services. The service interfaces only differ from the goal dependencies in the GRL model in two additional interfaces added during the design stage that were only implicit in the requirements model. The ILogin interface is required as part of implementing the IOrderProcessing interface of Amazin, as Amazin is the entry point into the single sign-in network. In the GRL model, the transfer of profile information is indicated using the Profile resource dependency. However, this leaves unspecified whether a customer needs to login, or has already been authenticated. The service design resolves this ambiguity. The IOrderStatusNotification interface is a technical side product of current web service frameworks, which require a separate interface for implementing callbacks.

4. Classification by the nature of interactions

As discussed in Section 2.2, it is helpful to distinguish between functional and non-functional features. Feature interactions can, thus, be categorized as either being of a functional or a non-functional nature, depending on what kind of side effect they have.[3] *Functional feature interactions* are those undesirable side effects of the composition of features that render the system no longer functional. *Non-functional feature interactions* are undesirable side effects in a system that is working from a purely functional point of view.

4.1. Functional interactions

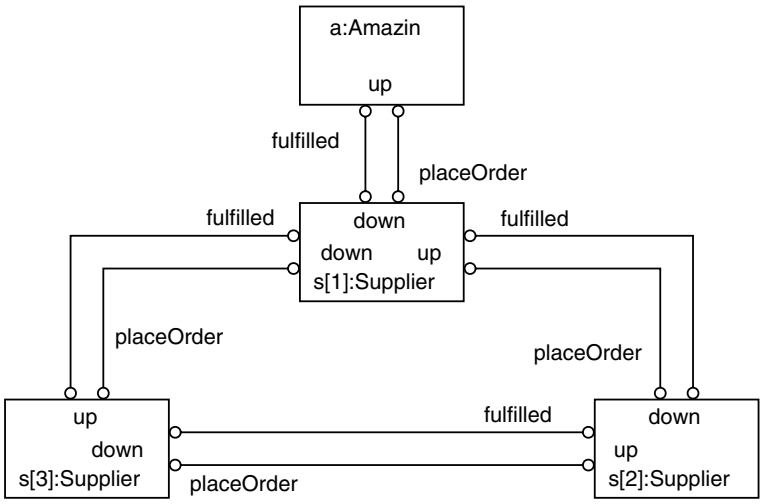
Example 1 (*Order Processing* with *Order Processing*). When a Supplier tries to fill orders for items that it temporarily does not have in stock from their network of suppliers, a loop can be created that causes another order to be placed to the initial Supplier. At that point, the OrderProcessing features could deadlock (if requests are synchronous), or a buffer overflow would eventually result as requests are repeatedly sent along the same chain of suppliers.

This is a feature interaction of multiple implementations of the same OrderProcessing feature, as executed by different actors.[4] An FSP model (structure diagram and process definitions) of a situation where the interaction will occur is shown in Fig. 16. It depicts a "closed" chain of three suppliers, where supplier s[1] receives the initial placeOrder request from Amazin, as well as the forwarded request from supplier s[3]. The up and down labels indicate the direction of the order flow. The structure diagram can be mapped into the composite process Virtual Bookstore. Relabeling associates the down and up actions of adjacent suppliers in the chain via channels (the chan[i] actions). Validation of this model will flag a deadlock with each Supplier trying to complete a placeOrder action. More details are presented in [25].

Example 2 (*Caching* with *Process Payment*). Caching digital content using Shark can prevent that access to the content will be properly billed. Since the Amazin service works correctly without caching (this point was discussed in Section 3.2), an assumption may have been built in that for every order, a respective order will be placed with a supplier, and thus no internal accounting (which keeps track of the

[3] This is not an exclusive categorization. Sometimes, an interaction shows aspects of both a functional and a non-functional interaction.

[4] It is also an instance of the well-known Call Forwarding loop in telephony. This high-lights that many known interactions can be found in emerging domains.



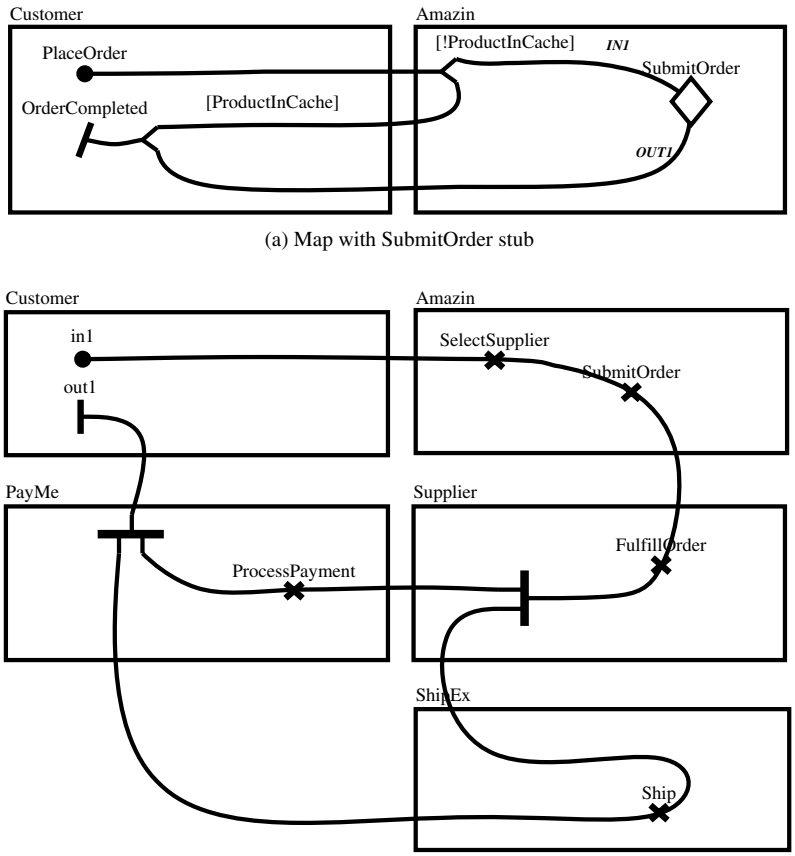

```
AMAZIN = (down.placeOrder->down.fulfilled->AMAZIN).
SUPPLIER = (up.placeOrder->ORDER_PROCESSING),
ORDER_PROCESSING = (
  inStock->up.fulfilled->SUPPLIER |
  notInStock->down.placeOrder->
  down.fulfilled->up.fulfilled->SUPPLIER).

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

Fig. 16. FSP model of OrderProcessing with OrderProcessing.

number of requests) is required. When caching is added to improve the performance of the service, there is a potential that the implications of this change are not fully understood by the designers.

Fig. 17(a) shows a UCM map for processing an order for an ebook. It contains a SubmitOrder stub that describes the process for placing an order with a supplier. The corresponding plug-in is defined in Fig. 17(b). This is also the basic flow of operations before adding a Caching feature to the service. Suppose Caching is added in such a way that before submitting an order, the cache is checked. If ProductInCache is true, the return path labeled [ProductInCache] will be taken, and the ebook is returned from the cache. It is important to note that adding the Caching feature is non-invasive. The map in Fig. 17(a) creates a strict wrapper around the basic order processing behavior in the Submit Order plug-in, i.e., it does not modify the basic behavior.

Example 3 (*Order Processing* with *Process Payment* or *Delivery*). There is a potential conflict between Process Payment and Order Processing, or Process Payment and Delivery due to timing errors. The interaction can result in either the customer being charged without the product having been shipped, or the customer receiving the product for free. Both errors exploit timing glitches. For example, when the customer cancels his order, it could be that payment still gets processed (because Process Payment was started before the order was canceled) but Delivery is aborted. The cancellation request was sent just before payment started, but arrived after



(a) Map with SubmitOrder stub

(b) Plug-in for SubmitOrder

Fig. 17. UCM model of Caching with Process Payment.

Process Payment has proceeded. A similar explanation can be given for the second case.

The UCM model in Fig. 18 provides the basis for understanding the cause of the interaction. The Process Payment and Ship responsibilities are initiated in parallel (the vertical bar after ProcessOrder indicates concurrency), and can execute in any order of one another. The Cancel requests to a component only take effect, if the Process Payment and Ship requests have not been started yet. This means that there are two successful cancellation scenarios, and two unsuccessful ones (where one of these requests has already been performed).

The reason for this feature interaction is missing transaction behavior. While there are protocols to ensure transaction behavior (e.g., WS-Transaction [4]), the scope of the atomic region that needs to be protected is generally difficult to identify in a service-oriented system due to its lack of a central authority.

Example 4 (*Order Processing* with *Fulfill Order*). Different interpretations of elements of an order may result in incorrect processing of an order. Heterogeneities may arise, when two elements with the same semantics are assigned different concrete names, types or cardinality, or two elements with the same value have different semantics [3]. As an example, Amazin may use the name price in the interpretation of raw price, while a Supplier interprets it as price including tax. In this example, we are dealing with a situation when the same concrete name is given to two elements with different semantics.

Detection of such interactions is currently outside the scope of our feature modeling approach, however, there are already ways to detect them [3]. In support of a point that was made in Section 1, this interaction demonstrates one of the key differences between service-oriented architectures and traditional telecommunication systems. Given the large number of web services, and many points of

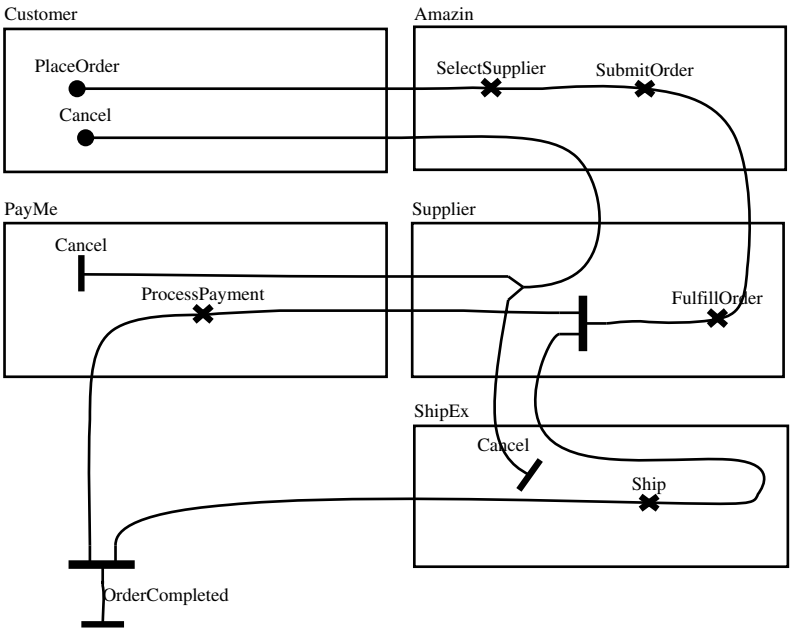


Fig. 18. UCM model of Order Processing with Process Payment or Delivery.

integration where web services are composed, an approach based on a central integration authority is, therefore, not feasible. The problem of semantic heterogeneity is not simply a problem of inaccurate ontologies, but heterogeneity must be accepted as a defining characteristic of web services. Therefore, solutions that can resolve such conflicts dynamically, i.e., through negotiation, have to be developed. In the web services domain, runtime feature interaction detection and resolution will be an essential requirement.

4.2. Non-functional interactions

Example 5 (*Authenticate User* with *Access Profile*). Any iPassport member organization can access the Customer's profile, including those organizations with whom the Customer has no trusting relationship: customers have no control over who has access to their profile. As shown in the GRL model of the interaction in Fig. 19, two non-functional goals, Convenience (satisfied) and Privacy (denied), of the Customer conflict with one another.[5]

This interaction can be detected from the GRL model. Fig. 19 adds labels to Fig. 6 to indicate to what degree goals in the graph are satisfied. For example, by placing a ✓ next to the Authentication task, we indicate that it is satisfied. The impact of assigning labels to goals can be evaluated using qualitative label propagation as implemented by GRL tools such as OpenOME [9]. The evaluation procedure propagates label values along links between goals. For instance, as there is a positive contribution from Authentication to Convenience, the Convenience goal will also be labeled as satisfied. Similarly, it follows that Authenticate User is satisfied, since it depends on Authentication.

In this example, the Privacy and Convenience goals cannot both be satisfied. Privacy is denied (as indicated by the *X* symbol) because, although there is a trusting relationship between Customer and Amazin, the relationships between Customers and Suppliers are untrusted, and there is no guarantee that a Supplier will adhere to Amazin's privacy policy. This violates the intent of the Customer, who (as a privacy-conscious individual) expects both goals to be satisfied. This is an unexpected and undesirable side-effect of combining the Authenticate User and Access Profile features within the same iPassport service.

This problem could be resolved by disassociating Authenticate User from Access Profile. In an

[5] This diagram is a refinement of Fig. 6. It is customary for GRL models to be developed over multiple iterations, during which primary contributions are initially identified, and side effects (modeled as correlations) at a later stage of analysis.

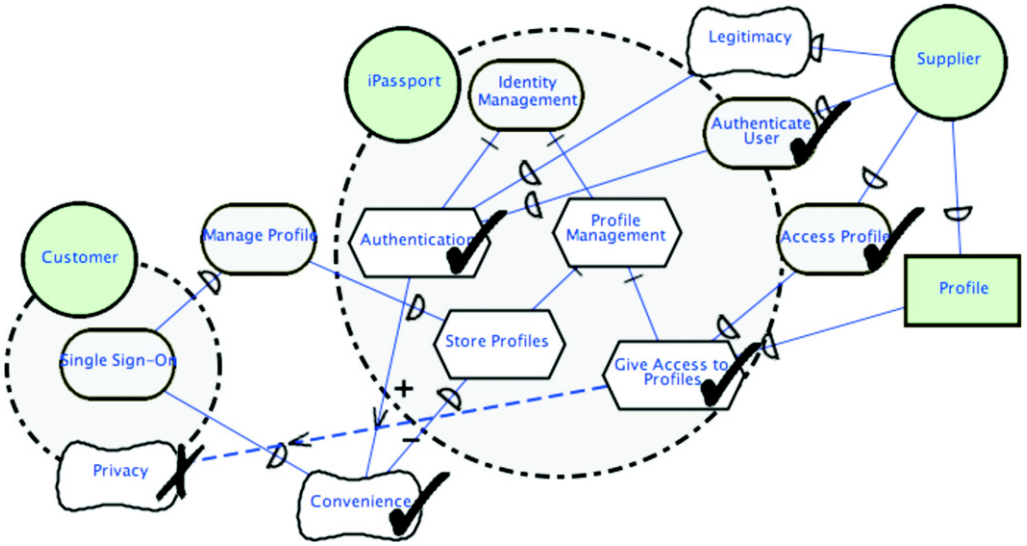


Fig. 19. GRL model of Authenticate User with Access Profile.

alternative design, the Access Profile feature could be required to obtain permission from the Customer before accessing the profile. This solution is instructive, because the undesirable interaction was not due to the existence of the Access Profile feature, but to how it was provided. Another solution would be to establish a chain of trust between Amazin and its Suppliers, in which Amazin trusts its Suppliers to keep profiles confidential. However, in an environment where service relationships are established dynamically (e.g., Amazin may consult a service directory for potential Suppliers), it may be hard to establish such trust, giving rise to new types of feature interactions.

Example 6 (*Access Profile* with *Access Profile*). Profile access cannot be restricted to specific service providers, as a way of mitigating the privacy violation in Example 5. The user can only choose to mark sections of the profile as either accessible by all service providers, or not accessible at all. No finer level of access can be specified (such as giving access to selected service providers only). This is another facet of the Convenience versus Privacy conflict.

We consider this a feature interaction of multiple instances of the Access Profile feature with itself, as profiles are transparently shared between providers.

Example 7 (*Manage Profile* with *Access Profile*). iPassportstores customer profiles on the customer's behalf. While this alleviates the need to store profiles at the customer's end, and allows access to the profiles even when customers are not online, it also makes iPassport a likely target of attacks, and introduces a Security issue, which further reduces the customer's Privacy.

Example 8 (*Order Processing* with *Order Processing* revisited). Example 1 can also be interpreted as a non-functional interaction. If either a deadlock or buffer overflow occurs, the Availability of the Suppliers will suffer. The user-perceived effect is that the Order Processing service is unavailable. Fig. 20 shows a scenario where Amazin is both a client and a supplier to a given Supplier. This can lead to a situation where the order is sent back to Amazin itself, which, in this case, is treated just like an Other Supplier.

In Fig. 20, the dependencies at the source of the issue have been high-lighted for emphasis. The *X* next to Availability indicates that this goal cannot be satisfied, and the high-lighting under the dependency links connected to Availability via contribution links signal the links at the root of the problem.

5. Classification by the causes of interactions

As the examples in Section 4 will have conveyed, there are many different causes of web service feature interactions. These include the causes shown in Fig. 21. We introduce two causes of interactions that we consider characteristic of web services, and that

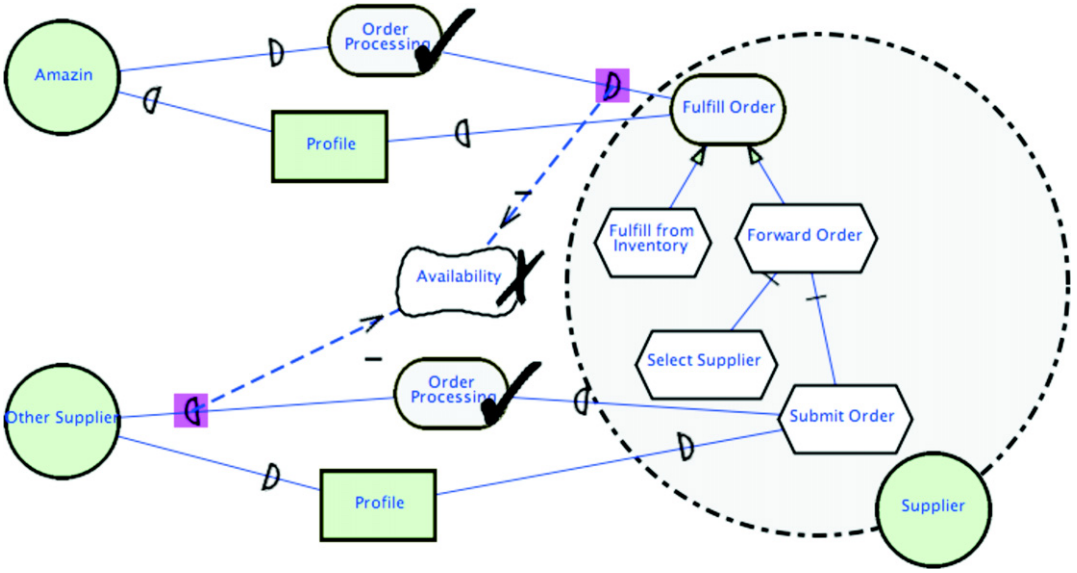


Fig. 20. GRL model of OrderProcessing with OrderProcessing.

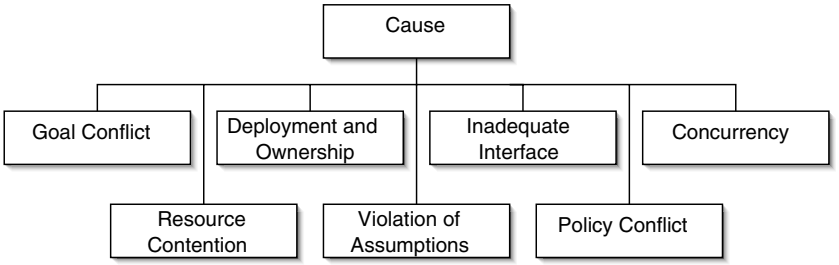


Fig. 21. Classification by causes of feature interactions.

are not encountered in such a pronounced way in closed, centralized telecommunications systems: deployment and ownership, and inadequate interfaces (also referred to as information hiding in [21]). The other causes have equivalents in telephony (although goal conflicts also appear to play a more significant role in the context of web services).

5.1. Goal conflict

Each feature has specific tasks or goals it is trying to achieve. When there is only one web service, there is also only one set of (conflict-free) goals. However, when services are combined into higher-level services, each with its own goals, it may be that the goals of those services (or, more precisely, the means for achieving those goals) are in conflict with one another, and we cannot guarantee to achieve them all. We consider a goal to be the expression of an intent of the service, as discussed in Section 2.3.1.

Goal conflicts are often the result of a *side effect*. In those cases, achieving one goal negatively affects another goal. Example 5 illustrates this type of interaction. The composition of the Authenticate User and Access Profile features results in an undesirable violation of Privacy as a side effect of how the intent of iPassport (Convenience) is achieved. (We like to distinguish between the intent of a service towards whose achievement it contributes, and side effects on other goals that it affects, as it achieves the primary goal.)

5.2. Resource contention

Features may be competing with each other through access to shared resources of finite capacity on a service provider. Examples of such resources are: disk space, memory, CPU, network bandwidth, database access, etc. The correct operation of one feature may be compromised by the *interference* of

another feature that is using more than its allotted share of resources.

Examples 1 and 8 illustrate resource contentions. If it is possible for a circular chain of suppliers linked through their Order Processing dependencies to form, a deadlock or buffer overflow may occur, which then decreases the Availability of the suppliers in the chain (which amounts to a denial of service).

5.3. Deployment and ownership

Decisions as to *who* provides the features needed (ownership), and *where* they should be provided (deployment) lead to performance, scalability and quality assurance, as well as to conflict of interest (separation of duty) issues:

1. Physical decoupling of features (by making dependent features run on different hosts, and sometimes under different ownership) helps solve resource contention (by avoiding that a single host becomes a bottleneck).
2. Grouping of related features gives their owner more control over the resulting system, and allows for performance optimizations. However, physical decoupling under the same ownership can lead to better scalability.
3. Conversely, delegating a feature to a third party removes the need for local management features to assure its quality. Of course, we then have to trust the third party that such features are properly supported.
4. Ownership of some features by the same owner can also lead to a conflict of interest, and a loss of trust in the owner due to (perceived) bias. This provides further incentives for delegation and physical decoupling.

Interactions of this type are shown in Examples 5 and 7. In both cases, one and the same entity (iPassport) authenticates the Customer, and controls access to its profile, respectively stores the profiles. Profile information is shared between Amazin and its Suppliers without involvement of the Customer (as the UCM model in Fig. 7 clearly shows). This causes the following conflict of interest: if acting in the interest of Customers, protecting their Privacy should be the foremost goal of iPassport; however, if acting on behalf of Suppliers, it would be in the best interest of the organization providing the iPassport service to share profiles with Suppliers, and thus violating the Privacy of Customers.

Due to the decentralized nature of control in a service-oriented system, deployment and ownership issues are faced more frequently and on a wider scale than in a traditional telecommunications system. Whereas such issues arise in telecommunication systems, there is usually an operator that oversees the service integration and can coordinate feature deployment and resolve ownership issues. Of course, with increasing deregulation in telecommunications, the problem will look very much like that facing web services.

5.4. Violation of assumptions

Feature developers need to make some assumptions about how other features (the users of the feature, and features provided to the feature) work. They can make incorrect assumptions, for example, due to *semantic ambiguity* (such as use of the same concepts in different ways), or the presence of different *versions* of the same feature. Similarly, feature implementations may be based on incorrect assumptions about their *context of use*. A characteristic of an assumption violation is that a change in one feature breaks a formerly correct assumption that another feature relies on.

Examples 1, 2, and 4 illustrate this type of interaction. Example 1 illustrates how assumptions are often violated as a result of service evolution. As suppliers decide to evolve their order processing processes to forward unfulfilled order to other suppliers, an assumption may have been made that the supplier which finally fulfills the order will be a different supplier. However, the presence of a loop in the chain of suppliers breaks that assumption. In Example 4, the same price element is used with two different interpretations.

5.5. Inadequate interface

Decoupling service interfaces (i.e., feature specifications) from their implementation is sometimes a dubious benefit. On one hand, decoupling insulates feature users from changes in the feature implementation. All they (apparently) need to know about the feature is the interface. On the other, the information conveyed through current web service interfaces (i.e., operation signatures) is not sufficient to properly use a feature. Feature users must make too many assumptions about how a feature is implemented. Similarly, service providers implementing

an interface may find that the interface does not provide sufficient guidance for the implementation. In such cases, they may make design decisions that constrain the use of the feature in unexpected ways.

A result of inadequate interface specification is that service users cannot control how a service performs. For example, they may not be able to pass on required parameters to a service implementation. This can lead to a duplication of effort, inconsistencies, and incorrect execution. Conflicts of this type may concur with a policy conflict, where the policies of feature users are not in agreement with the (unexpressed) policies of the service provider.

Examples 5 and 6 demonstrate inadequate interface interactions. In Example 5, the iPassport service does not declare through any of its interfaces that profiles will be shared transparently with parties other than Amazin. In Example 6, the Manage Profile feature does not allow a Customer to impose restrictions on what profile information service providers can access. However, the Customer may not want all of these parties to access their full profile.

While the general issue of inadequate specification of features is certainly also a significant issue in the telephony domain, we feel that the issue is both more specific and more pervasive in the web services context. The notion of a web service interface is very specific in its focus on service operations, and current proposals to include non-functional aspects in the interface definition have been limited. Also, there is not one well-known set of interfaces that service providers implement corresponding to the standard set of telephony features (not including multimedia and other emerging services). Rather, services may evolve from a specific *context of use*, and, therefore, impose limitations on service interfaces (e.g., parameters that cannot be passed across the interface) that may lead to feature interactions once the context of use is widened.

5.6. Policy conflict

Policies provide the means for specifying and modulating the behavior of a feature to align its capabilities and constraints with the requirements of its users [10]. A policy conflict occurs, if there are policies (e.g., authentication or privacy policies) specified on two features that refer to their corresponding operations, and the policies are not compatible [16].

Policy conflicts are particularly prone to cause user confusion, as policies are often specified by the users as part of customizing a feature. Example 5 shows such an interaction. Suppliers are not bound to the same privacy policy as Amazin. For example, a Supplier may decide to keep profile information beyond the extent of the order, whereas Amazin's privacy policy forbids this.

5.7. Concurrency

The correct operation of a composite web service may also depend on properly handling concurrency. Incorrect invocation order, timing glitches, and transaction failures cause this type of interaction. A feature may expect events to take place in a certain order. If a feature user breaks this order, the correctness of the feature's results is no longer guaranteed. Another situation is that certain operations must either all complete, or not have any effect at all. Example 3 illustrates this type of interaction. An order cancellation should not only terminate either Payment Processing or Delivery, but both. A better solution, in this case, would have been to only ship an order after payment has been received. However, as the execution of a business process is distributed over several service providers, each performing part of the task, identifying the boundaries of a transaction, and furthermore enforcing them, is generally difficult in a service-oriented system, as it lacks a central authority.

6. Conclusion

This paper makes three contributions:

1. It discusses characteristics of web services that suggest the separate treatment of web service feature interactions.
2. It develops a case study that we hope can serve as benchmark for comparing approaches to web service feature interaction management.
3. It proposes a classification of feature interactions among web services by their nature and causes, using the case study for illustration.

Throughout the paper, we have made many observations about the nature of web services that suggest that we should treat web service feature interactions separately from other types of feature interactions. These include:

- Systems using web services are largely built from third-party services, over whose implementation service users have little control.
- There are a large number of service users and small- and medium-sized service providers, and all are potential service integration points.
- New services can be cheaply created by assembling existing ones. Specifically, this means services will be created by end users.
- There are no operators or "central authorities" (unlike in the telecom world) which oversee the integration of services from different vendors.
- Web services must evolve rapidly to adapt to changes in the business environment, and to meet the needs of loosely coupled business models.
- Heterogeneity must be accepted as a defining characteristic of web services. We cannot assume the various actors to use a common ontology.
- There is an increasing trend towards automated service discovery and composition, and towards making service context-aware.
- Furthermore, the concept of an ACID properties from the database world cannot be easily mapped into the web services context.
- Establishing trust will also become much harder in a world, in which we cannot rely on pre-existing, proven relationships.

In light of these observations, we have proposed a classification of feature interactions among web services by their nature and causes. This classification is summarized in Table 2. Numbers refer to examples that illustrate a given type of interaction. Some examples appear more than once, because they exemplify multiple interactions. It should be pointed out that the table is likely not complete, and we invite readers to add to it.

We position our classification in the tradition of existing classifications of feature interactions for the telecommuncations domain, while emphasizing web service-specific aspects. A classification of web service feature interactions is beneficial, as it helps us understand the scope of the feature interaction problem in the web services domain, and provides a benchmark against which to assess the coverage of solutions to this problem. Solutions for specific feature interactions can then be generalized to other interactions of the same type.

Table 2
Summary of the classification of web service feature interactions

Cause of interactions	Nature of interactions	
	Functional	Non-functional
Goal conflict		5
Resource contention	1	8
Deployment and ownership		5, 7
Violation of assumptions	1, 2, 4	
Inadequate interface		5, 6
Policy conflict		5
Concurrency	3	

Specifically, our classification builds on the work by [6] for the telecommunications domain. That work was based on the premise, even more important now with web services, that service creation is no longer governed by a single organization. It also discusses a categorization by *nature* of the interactions, which depended on the nature of the features involved, the number of users, and the number of components in the network, and by *causes*. However, some causes of feature interactions do not carry over to the web services domain. For instance, we do not list limitations on network support as a cause, since explicit service invocation in web services avoids signaling ambiguity.

In our own earlier work [20,21], we have provided additional examples of interactions, as well as approaches for resolving them. By contrast this paper does not consider resolution. The goal conflict scenario is based on our work on assessing privacy technologies [24]. An earlier version of the classification was presented in [22]. In this paper, we have reworked the classification, added examples of interactions, and put it into a more accessible format.

Acknowledgements

Parts of this research were sponsored by an NSERC Discovery Grant. We also want to thank the students who participated in the implementation of the prototype of the Amazin virtual bookstore, and in related case studies. In particular, we want to recognize Yi Lin and Zhiyong Lu for their efforts in coding much of the Amazin prototype, and Alex Oreshkin for his contribution to exploring the use of FSP in various feature interaction scenarios.

References

[1] G. Alonso, F. Casati, et al., Web Services: Concepts, Architectures, and Applications, Springer, 2004.
[2] D. Amyot, Introduction to the user requirements notation: Learning by example, Computer Networks 42 (3) (2003) 285–301.

[3] V. Arago, A. Fernandes, Conflict resolution in web service federations. In: Proceedings of International Conference on Web Services (Europe) LNCS, vol. 2853, Springer, 2003, pp. 109–122.
[4] F. Cabrera, et al, Web Services Transaction, IBM, Microsoft, and BEA, 2002.
[5] M. Calder, M. Kolberg, E. Magill, S. Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Computer Networks 41 (1) (2003) 115–141.
[6] J. Cameron, N. Griffeth, et al., A feature interaction benchmark for IN and Beyond, Feature Interactions in Telecommunications Systems (1994) 1–23.
[7] F. Chen, S. Li, W.C. Cheng-Chung, Feature analysis for service-oriented reengineering. In: Proceedings of Asia–Pacific Software Engineering Conference, IEEE, 2005, pp. 201–208.
[8] E. Gamma, R. Helm, et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 2004.
[9] GRL. Available from: <http://www.cs.toronto.edu/km/GRL>, last accessed in April 2006.
[10] H. Kamoda, et al, Policy Conflict Analysis Using Free Variable Tableaux for Access Control in Web Services Environments. In: Proceedings of World Wide Web Conference, 2005.
[11] D. Keck, P. Kuehn, The feature and service interaction problem in telecommunications systems, IEEE Transactions on Software Engineering (1998) 779–796.
[12] J. Magee, J. Kramer, Concurrency: State Models and Java Programs, Wiley, 1999.
[13] E. Magill, Feature Interactions: Old Hat or Deadly New Menace? in: K. Turner, E. Magill, D. Marples (Eds.), Service Provision: Technologies for Next Generation Communications, Wiley, 2004, pp. 235–252.
[14] E. Pulvermüller, A. Speck, 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.
[15] A. Ryman, Understanding Web Services. Available from: <http://www.software.ibm.com/wsdd/library/techarticles/0307_ryman/ryman.html>, 2003.
[16] A. Sahai, C. Thompson, W. Vambenepe, Specifying and Constraining Web Service Behavior through Policies, Workshop on Constraints and Capabilities for Web Services, W3C, 2004.
[17] UCM. Available from: <http://www.usecasemaps.org>, last accessed in April 2006.
[18] webMethods, Glue User Guide. Available from: <http://www.webmethods.com/docs/glue/guide>, last accessed in April 2006.
[19] M. Weiss, Feature Interactions in Web Services, Feature Interactions in Telecommunications and Software Systems, IOS, 2003, pp. 149–156.
[20] M. Weiss, B. Esfandiari, On Feature Interactions among Web Services. In: Proceedings of International Conference on Web Services, IEEE, 2004, pp. 88–95.
[21] M. Weiss, B. Esfandiari, On feature interactions among web services, International Journal on Web Services Research 2 (4) (2005) 21–45.
[22] M. Weiss, B. Esfandiari, Towards a Classification of Web Service Feature Interactions. In: Proceedings of International Conference on Service-Oriented Computing LNCS, vol. 3826, Springer, 2005, pp. 101–114.
[23] M. Weiss, D. Amyot, Business process modeling with URN, International Journal of E-Business Research 1 (3) (2005) 63–90.
[24] M. Weiss, B. Esfandiari, Modeling Method for Assessing Privacy Technologies, in: G. Yee, Privacy in e-Services, Idea Books, 2006, pp. 265–280.
[25] M. Weiss, A. Oreshkin, B. Esfandiari, Method for detecting functional feature interactions of web services, Journal of Computer Systems Science and Engineering 21 (4) (2006) 273–284.

Michael Weiss (Ph.D., University of Mannheim, 1993) is an associate professor of Computer Science at Carleton University. Before joining Carleton in 2000, he worked four five years in the telecommunications industry (Mitel Corporation) on agent-based service creation environments. His research interests include software architecture and patterns, service-oriented architectures, and open source.

Babak Esfandiari is an associate professor at the department of Systems and Computer Engineering at Carleton University, Ottawa, Canada. He obtained his Ph.D. in computer science at the University of Montpellier, France in 1997. His research interests include agent-based systems and network computing.

Yun Luo was a master's student at the School of Computer Science, Carleton University. He is presently a software engineer at Nortel Networks.