Interoperability among Independently Evolving Web Services

Shankar R. Ponnekanti and Armando Fox

Stanford University

Abstract. The increasing popularity of XML Web services motivates us to examine if it is feasible to substitute one vendor service for another when using a Web-based application, assuming that these services are "derived from" a common base. If such substitution were possible, end users could use the same application with a variety of back-end vendor services, and the vendors themselves could compete on price, quality, availability, etc. Interoperability with substituted services is non-trivial, however, and four types of incompatibilities may arise during such interoperation – *structural*, *value*, *encoding* and *semantic*. We address these incompatibilities three-fold: (1) static and dynamic analysis tools to infer whether an application is compatible with a substituted service, (2) semi-automatically generated middleware components called *cross-stubs* that actually resolve incompatibilities and enable interoperation with substituted services, and (3) a lightweight mechanism called *multi-option types* to enable applications to be written from the ground up in an interoperation-friendly manner. Using real applications and services as examples, we both demonstrate and evaluate our tools and techniques for enabling interoperation with substituted services.

1 Introduction

Maintaining interoperability among independently evolving components is an important but challenging problem in distributed systems (e.g., Vinoski [1]). We address a variant of this problem in the context of SOAP/WSDL based XML Web services. Multiple Web service vendors (e.g., Google, Hotbot and Altavista) may offer services with substantially similar functionality, and this leads to two scenarios: (1) *autonomous services*, where each vendor independently defines his WSDL service, or (2) *derived services*, where once a vendor becomes popular, other vendors adopt and extend the popular vendor's service. This paper shows that if vendors cooperate and espouse the derived services model, cross-vendor interoperability can be achieved. (While our techniques also apply to the autonomous services case, it is much more difficult, as explained in section 8.)

The derived services case, illustrated in figure 1, is a variant of the independent evolution problem, since each of the derived vendor services represents an independent extension of the base service. Our goal is to facilitate interoperation with *non-native services*, i.e., services other than the one the application was originally written to. (See the figure for more details.) E.g., a Google application

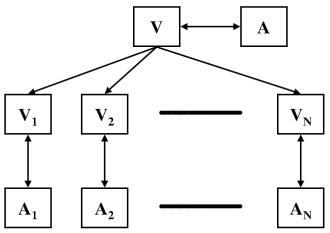

Fig. 1. Vendors $V_1, V_2, \ldots, V_N$ derive their services from base service V. Applications $A_1, A_2, \ldots, A_N$ are written to services $V_1, V_2, \ldots, V_N$ respectively. We refer to the service V_i as the *native* or *source* service of A_i, while all other services $V_j, j \neq i$ and V are considered *non-native* services for A_i. For an application A written to base service V, all of the derived services $V_1, V_2, \ldots, V_N$ are non-native services. Only a sibling, parent or a child of the source service – and not any unrelated service – qualifies as a non-native service. We only consider one level of derivation in this paper.

should ideally interoperate with other search engines also, a driving directions application written to an existing provider (e.g., Arcweb) should be switchable to a future provider that offers traffic-sensitive driving directions. Much like other services in the economy such as credit-cards and insurance services, the choice of a Web service depends on many factors (cost, reputation, quality, speed, etc.), all of which vary over time, apart from the fact that some vendors may shut down and others may come online. E.g., Inktomi was once the dominant search provider, today it is Google, and it might be Microsoft or Yahoo in the future. Lack of interoperability with non-native vendors can significantly undermine the "revolution" promised by Web service proponents. An analogy is being unable to switch credit card, cellphone, insurance or mortgage providers.

Non-native service interoperability is challenging however – in section 2, we identify four types of incompatibilities it may entail: *structural*, *value*, *encoding* and *semantic*. We address this problem with static/dynamic analysis and middleware based tools and techniques, and the results obtained with existing Web services and applications are summarized below:

- Static analysis reveals that many Web service applications only use a fraction of a service's functionality, making the proposition of interoperation with non-native services a reasonable one.
- For existing applications and services we have studied, a combination of static and dynamic analysis is sufficient to make automated determinations of compatibility with non-native services.
- Actual interoperation with the non-native service can be realized through semi-automatically generated *cross-stubs*. Cross-stub generation involves incompatibility resolution when the application is not fully compatible with the non-native service, and is done using a GUI-based tool.
- A number of incompatible application-service pairs we studied indicates that incompatibilities often arise due to *non-critical* reasons – reasons that do not hamper the basic functioning of the application.

– If applications are authored using our proposed *multi-option types*, it is possible to automatically determine whether incompatibilities with a non-native service are non-critical, and if so, automatically resolve the incompatibilities.

To summarize, an application owner (i.e., a developer or a user/administrator) can (1) auto-determine which non-native services are fully compatible, (2) set up interoperation even with incompatible services, provided the incompatibilities are (manually) determined to be non-critical, and (3) for applications written with multi-option types, even auto-determine whether the incompatibilities with a non-native service are non-critical, and if so, auto-resolve them.

In the context of XML applications and services, HydroJ [2] has recently addressed the evolution problem by building upon "XML pattern languages" [3]. While HydroJ proposes new kinds of language types and paradigms, we address the problem using the *static host types* approach, in which WSDL specifications are statically mapped to language types in the client application. The static host types approach is already widely adopted through Web service toolkits such as AXIS, Glue, Wasp and .NET, making our approach easier to deploy and also applicable to legacy applications. Furthermore, we go beyond HydroJ by providing a mechanism for resolving incompatibilities if a desired non-native service is not fully compatible, and we also propose multi-option types to further enhance interoperability. Vinoski [1] observes that the static host types approach is convenient but sacrifices XML flexibility, a limitation we address in this paper.

Although our work is implemented in the context of Web services due to the current interest in these technologies, our techniques are more generally applicable. Also, while our focus is on *cross-vendor* interoperability, our techniques can also be applied to the (simpler) *cross-version* interoperability problem, where an application needs to interoperate with multiple versions of the same vendor service. The rest of the paper proceeds as follows. Section 2 describes service derivations and the types of incompatibilities that may arise with non-native services. Section 3 describes the static and dynamic analysis algorithms and tools we use to identify compatibility between applications and non-native services. Section 4 describes incompatibility resolution and cross-stubs. Section 5 reports experiments with existing Web services (Google, Mappoint and Arcweb) and existing client applications. Section 6 describes our proposal for authoring future client applications in an interoperation-friendly manner. We discuss semantic issues in section 7, related work in section 8 and conclude in section 9.

2 Service Derivations and Incompatibilities

2.1 Background

A WSDL *service* (see figure 2 for an example) is a collection of *porttypes* (which roughly correspond to RPC interfaces, although Web services are *not* equivalent to traditional distributed objects [4,5,6]), each of which supports a set of *operations*. An operation has input and output *messages*. The types of input and output messages are specified using XML schema. (We sometimes use the terms parameters and returns to denote inputs and outputs respectively.) XML

schema defines built-in *simple types* (int, float, string, etc.) and allows *complex types* to be built up from them. Unlike most traditional type systems, type derivation in XML schema allows for both *extension* and *restriction.* Thus, a complex type can be extended by adding new mandatory or optional elements, and also restricted by removing optional (but not mandatory) elements. In addition, the value space of simple types can be restricted (but not extended) by applying one or more *facets* such as enumeration, regular expression pattern, minimum/maximum value, etc.

Limitations: For the reader familiar with XML schema, we only consider complex types with element-only content, and do not consider identity constraints, wildcards or substitution groups. For a reader familiar with WSDL, we deal with document style and RPC-style operations in this paper, but not the other lesser known and used operation styles. Finally, we only consider one level of service derivation. (Service derivation is explained below.) Some of these excluded features may not raise any new issues, and none are used by the services we have experimented with so far, but we plan to consider them in future work when services using them become available.

2.2 WSDL Service Derivation

The WSDL specification does not yet define service derivation. One might imagine two types of derivations: add/remove an operation or extend/restrict the input and output message types of an operation. As an example, two vendors etailer1 and etailer2 could independently derive from vendor etailer's service of figure 2 as follows:

- etailer1 adds customer ratings. Accordingly, etailer1 extends `KSRequest` to `KSRequest1` by adding an optional `minRating` field. Similarly, `Product` is extended to `Product1` by adding an optional `rating` field.
- etailer2 does not sell books, and does not maintain sales ranks. So, etailer2 restricts `Category` to `Category2`, which only allows "All", "Music" and "Movies". Also, `Product` is restricted to derived type `Product2`, which removes the optional `salesrank` field.

Independent extensions (but not restrictions) can lead to semantic conflicts if two vendors each add a field with the same name but different "meanings". This problem is avoided in XML schema if each vendor defines his extensions in his own namespace. Thus, the `minRating` field added by etailer1 belongs to namespace "etailer1", and our techniques (conservatively) consider it as different from a `minRating` field added by another vendor. If, on the other hand, a vendor wishes to reuse a field with the same semantics added by another vendor, there is a mechanism to import an element from another namespace in XML schema.

While XML schema allows both extension/restriction for complex types by adding/removing fields, it only allows restriction of the value spaces of simple types. Value space extension, while useful for deriving services, raises the same semantic issues as adding fields. Unlike fields, values have no namespaces, but we can address this problem with facets, as discussed in section 7.

```
<definitions targetNamespace="etailer">
  <portType name="EShop">
    <operation name="keywordSearch">
      <inputpart name="request" type="KSRequest"/>
      <outputpart name="product" type="Product"/>
    </operation>
  </portType>
  <complexType name="KSRequest">
    <element name="keyword" type="string"/>
    <element name="category" type="Category" minOccurs="0"/>
  </complexType>
  <complexType name="Product">
    <element name="id" type="string"/>
    <element name="category" type="Category"/>
    <element name="salesrank" type="int" minOccurs="0"/>
  </complexType>
  <simpleType name="Category">
    <restriction base="string">
      <enumeration values="All, Books, Music, Movies"/>
    </restriction>
  </simpleType>
</definitions>
```

Fig. 2. A vendor named etailer defines a WSDL service in the "etailer" namespace containing a porttype EShop with a single operation keywordSearch with input type KSRequest and output type Product. KSRequest has two fields: a keyword and an optional category that can take one of four values "All", "Books", "Music" or "Movies". The output type Product has three fields: id, category and salesrank. (We use some notational shortcuts here, e.g., inputpart and outputpart are not legal WSDL elements.)

2.3 Types of Incompatibility

Given the service derivation scheme of the previous section, four types of incompatibilities may arise between applications and non-native services: *structural*, *value*, *encoding* and *semantic*. A structural incompatibility is a mismatch in the structure of the (XML) message sent by the sender and expected by the receiver, while a value incompatibility arises when the structure is as expected, but the filled-in values are unexpected. The bulk of the paper deals with structural and value incompatibilities (together referred to as *SV-incompatibilities*), and they are explained in more detail below. Encoding incompatibilities, illustrated and addressed in section 4, arise because instances belonging to different schema types are not identical even if they have the same structure and identical values. As explained in the previous section, semantic incompatibilities arise when different vendors introduce extensions with identical syntax (i.e., same structure and value) but differing meanings, and the problem can be alleviated by namespaces and facets as further discussed in section 7.

Considering applications written to a given source service S, a non-native target service T may differ from S in the following ways, each of which represents a kind of SV-incompatibility:

- Missing methods: If T removes a method from the base service, or if S adds a method, S-applications may use this method and hence be incompatible with T. The converse case of extra methods in T does not cause an incompatibility.
- Extra fields: If T adds a field f to the base service, or if S removes f, S-applications do not use f, but T expects a value for f if it is mandatory (`minOccurs > 0`). No incompatibility occurs for optional extra fields, or for extra T fields in method outputs (as opposed to method inputs).
- Missing fields: If T removes (or S adds) a field f, S-applications using this field are incompatible with T. Incompatibility occurs even if f is optional (`minOccurs=0`) – if an S-application uses an optional field, it may be important to the application, and cannot be ignored.
- Facet mismatches: If S and T have different facets for an input field f, where the value space for $S.f \not\subseteq$ the value space for $T.f$, values passed by S-applications may be disallowed for T resulting in incompatibility. For output fields, incompatibility occurs when value space of $T.f \not\subseteq$ value space of $S.f$.
- Cardinality mismatches: If S and T have different cardinality requirement for a field f, as when T declares `minOccurs=3` and S declares `minOccurs=2`, incompatibility may result.

In the first case, we refer to the offending method m as the *causal method*, while in other cases the offending field is called the *causal field*. In the last four cases, we further distinguish whether the causal field is an input/output field. These incompatibility categories are summarized in figure 3. In the examples we have experimented with so far, $I_1 - I_5$ (missing methods, missing/extra fields and input facet mismatches) are prevalent, while the others are rare. Finally, note that all the SV-incompatibilities can be detected automatically given the source and target service WSDL specifications.

3 Application Usage Behavior and Compatibility

While the incompatibilities of figure 3 represent *all* the potential SV-incompatibilities between S and T, many of them are irrelevant for a given S-application A, e.g., an I_1-category incompatibility is irrelevant for A if A never calls the causal method. In this section, we discuss how we gather the method and field usage information from the application.

3.1 Usage Inference with Static Analysis

As noted earlier, our approach is primarily intended for applications written using static host types. In the static host types approach, the WSDL specification and XML schema types are mapped to Java classes, and applications are written using these classes. The Java classes generated for etailer1's service, and sample application code using this service is shown in figure 4.

Our static analysis tool, called UAT-S (usage analysis tool-static) identifies which fields of the inputs are filled in by the application, and for leaf-level fields UAT-S also determines which values they are set to. Similarly, UAT-S also identifies which output fields of the service result are actually consumed by the

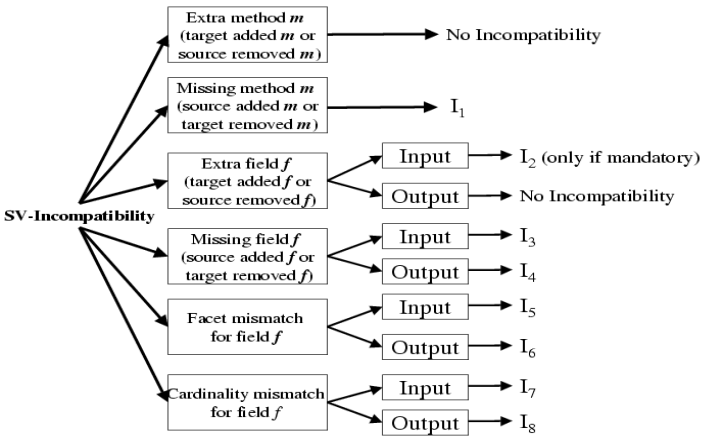

Fig. 3. The possible SV-incompatibilities (i.e., structural and value incompatibilities) when applications written to service S interoperate with a non-native service T. T may differ from S in having extra/missing methods, extra/missing fields or cardinality/facet mismatches. The I_2 incompatibility occurs only if the causal field is mandatory i.e., minOccurs>0. (Other incompatibilities can occur even with optional fields.)

```
package etailer1;
public interface JEShop1 {
  public JProduct1 keywordSearch(JKSRequest1 x);
  public JProduct1[] alsoBought(String id, String category);
}
public class JKSRequest1  {
  public String keyword;
  public String category;
  public Integer minRating;
}
public class JProduct1  {
  public String id;
  public String category;
  public Integer salesrank;
  public Integer rating;
}
public class JEShop1Stub implements JEShop1 { .. }
// Sample application code
L1   JEShop1Stub stub = new JEShop1Stub();
L2   JKSRequest1 req = new JKSRequest1();
L3   req.category = "Music";
L4   req.keyword = infile.read(..);
L5   JProduct1 prod = stub.keywordSearch(req);
L6   outfile.write(prod.id, prod.salesrank);
L7   JProduct1[] others = stub.alsoBought(prod.id, prod.category);
```

Fig. 4. Java classes generated for the etailer1 service of section 2. We add an extra method "alsoBought" here, which returns other products purchased by customers who bought a given product. Notice how the code "pipes back" the id and category fields of the keywordSearch return from line L5 to the alsoBought method on line L7.

application. To do this, UAT-S performs an interprocedural points-to analysis [7] on the application code. For each call site of every method in the service interface, UAT-S determines which allocation sites the parameters and returns (and their subfields) can point to. Based on these points-to sets, UAT-S generates *usage tuples*, such as shown below for the application code in figure 4:

```
T1  <method, keywordSearch>
T2  <method, alsoBought>
T3  <input, keywordSearch@param1.category, known, "Music", appCode>
T4  <input, keywordSearch@param1.keyword, unknown>
T5  <input, alsoBought@param1, return, keywordSearch@return.id>
T6  <input, alsoBought@param2, return, keywordSearch@return.category>
T7  <output, keywordSearch@return.id>
T8  <output, keywordSearch@return.salesrank>
    .....
```

The first component indicates whether the tuple refers to a method or an input/output field. The second component identifies the method or the input/output field. A method tuple identifies that the specified method is invoked at some call site in the application. For input tuples, the third component can be known, unknown or return as explained below:

- known: This means at some service callsite, the application passes a known value for this field. For example, T3 was derived from line L3 of the sample code. For known input tuples, the final component indicates the source of this value, i.e., application or library code.
- unknown: This means at some service callsite, the value passed to this field cannot be statically determined. T4 and line L4 illustrate this case.
- return: This means that at some callsite, the value passed to this field was obtained as a return from another callsite. For example, T5 indicates that `prod.id` obtained as return from `keywordSearch` on line L5 is piped back to `alsoBought` on line L7.

The output tuples record which output fields are consumed by the application. For example, T7,T8 indicate that output fields id and salesrank were consumed (line L6). The input field `minRating` is never used by the application, and the output field `rating` is never consumed, so no tuples are generated for them. Input tuples of type "known" and "unknown" are only generated for leaf level input fields. Values are not recorded for outputs (unlike for inputs), and we assume that services can generate any legal output value (i.e., any value allowed by the facets declared in the specification).

3.2 Compatibility Based on Statically Inferred Usage Behavior

We provide a tool CAT-S (compatibility analysis tool-static) that uses the usage tuples to eliminate irrelevant SV-incompatibilities between the source and target interfaces. Each incompatibility i is processed by CAT-S as follows depending on which category (among $I_1 - I_8$ from figure 3) it belongs to:

- I_1: If the causal missing method m is never invoked, i.e., there is no method usage tuple for m, i is irrelevant.

- I_2: If the causal extra field belongs to method m, and there is no usage tuple for m, i is irrelevant, because the method to which the extra field belongs to is never invoked.
- I_3-I_4: If there is no usage tuple for the causal missing field f, i is irrelevant because the missing field is never used.
- I_5: If there is no input tuple for the causal field f, i is irrelevant. Otherwise, set RELEVANT to false and iterate through each input usage tuple t involving f:
  - If t's type is unknown, set RELEVANT to true. This is because if the value for f at some call site is unknown, we assume (conservatively) that it can be the worst case value – one that causes the incompatibility.
  - If t's type is known, and the known value is disallowed by the target service's facets for f, set RELEVANT to true, else leave RELEVANT unchanged.
  - If t's type is return, and if the piped return field is r (for example, in tuple T6, $r =$ `keywordSearch@return.category` and $f =$ `alsoBought@param2`), if the target service's value space for $r \not\subseteq$ the target service's value space for f, we set RELEVANT to true, else leave RELEVANT unchanged. (This is because the value returned by the target service for r is being sent back to the target service as f, so there is no incompatibility if the value space for $r \subseteq$ the value space for f.)

  Once we have iterated through all tuples involving f, if RELEVANT is false, i cannot occur and is irrelevant.
- I_6-I_8: If there is no usage tuple for the causal field f, i is irrelevant.

First, the asymmetry in handling inputs vs. outputs in rules I_5 and I_6 arises because, as noted earlier, we assume that services can generate any legal value allowed by the declared facets. Second, the above rules are correct, but not optimal as elaborated below:

Correctness: The points-to analysis we perform as part of UAT-S can escape assignments when the application code uses native methods, dynamically loaded code (using URLClassLoader) or reflection. In these cases, the above rules only declare incompatibilities as "likely irrelevant" rather than irrelevant. In other cases, the above rules are always conservative.

Optimality: The above rules for $I_5 - I_8$ are not optimal – unlike missing method and missing/extra field incompatibilities, exact constraint and facet compatibility cannot always be determined statically for applications written in general-purpose languages such as Java, so we err on the conservative side. For example, it is possible that an application always passes values allowed by the target service facets, but if the values passed cannot be statically determined (i.e., there is an unknown input usage tuple for this field), the above rule for I_5 (conservatively) does *not* rule the incompatibility as irrelevant. Similarly, fields that can occur multiple times are typically modeled as arrays in the static host types approach, and optimal detection of cardinality incompatibilities requires that the array lengths be determined statically, which is not always possible.

We have rarely encountered $I_6 - I_8$ incompatibilities in the examples we have dealt with so far, although I_5 incompatibilities are quite frequent. Dynamic analysis (discussed in the next section) partially addresses the sub-optimality of the rules in this case.

3.3 Dynamic Analysis

Dynamic analysis complements static analysis, especially when an application can theoretically generate an incompatible message, but never does so in practice. To enable dynamic analysis, we track the input messages sent by the application to the service at runtime in normal operation. Message interception can be performed at the stubs if they are suitably instrumented, or using a network-level proxy otherwise. Since we control stub generation, we use the former approach. CAT-D (compatibility analysis tool-dynamic) simply checks if all the past input messages sent by the application are compatible for the target service. Clearly, the effectiveness of CAT-D increases as time of capture increases. Dynamic message tracking and compatibility checking is technically straightforward, so we do not discuss further mechanical details.

Summary of compatibility determination: When it is desired to switch an S-application A to one of several non-native target services, say available at a registry, we do the following for each target service T:

- Generate all SV-incompatibilities between S and T from their specifications.
- Using the usage tuples for A and the CAT-S rules, filter out irrelevant incompatibilities.
- Among the remaining incompatibilities (if any), mark the ones that do not occur in practice (as determined by CAT-D) as likely irrelevant.

The target services are ranked based on the remaining incompatibilities, referred to as *relevant incompatibilities*. While fully compatible target services (if any) are best from an interoperability standpoint, the application owner may still choose a not fully compatible target service for other reasons (cost, reputation, quality, etc), and the next section explains incompatibility resolution.

4 Cross-stubs

Instead of a regular stub, the interaction between an application and a non-native target service is mediated at runtime by a semi-automatically generated middleware component called *cross-stub*. A cross-stub is link-compatible with a regular-stub, allowing the unmodified application to be run against the non-native service. When generating a cross-stub, the application owner must resolve the relevant incompatibilities (if any). The available resolution choices are:

1. Runtime exception: Here, the cross-stub throws a runtime exception when this incompatibility is detected at runtime. As indicated by JEShop1Stub, a network service application is already expected to handle an exception, for other reasons such as service and network failures. This choice is applicable for all incompatibility categories.

2. Ignore: For missing fields and for cardinality/facet mismatches involving optional fields, the owner may choose this option if it is acceptable for the causal field to be dropped.
3. Supply a value: For extra fields (category I_2), the owner can supply a value that should be used for this field.
4. Alternative value: For facet mismatch incompatibilities, the owner can provide a substitute value to be used when the value supplied by the application is disallowed by the target service facets. The substitutes can be tailored to the target service facets, as the following examples indicate:
   - Enumeration facet: A fixed substitute value is specified.
   - Range facet: The owner may choose to use the closest legal value.
   - Pattern facet: A search and replace substitute pattern (similar to the "s/../../" expressions of `sed` and `perl`) may be specified, and is applied when the original value is disallowed by the target service facets.

As an example for the pattern facet case, in both their Web UI and WSDL API, the Google query parameter allows special operators in queries, such as "foo file:pdf" and "foo site:cnn.com". If the target service specifies a pattern facet disallowing "file:" queries, the application owner can provide a substitute pattern that removes the "file:" terms from the query.

In all the above four options, the owner can optionally specify a *notify message*, to be displayed to the user (with a pop-up window) at runtime when the incompatibility occurs and the resolution action is taken. Once the owner resolves all the relevant incompatibilities, the cross-stub is auto-generated from the source and target service specifications and incorporates the resolution code.

For some incompatibilities, resolution may require more complex handling than selecting one of the standard resolution choices. (E.g., a missing method incompatibility may be better handled by invoking a third-party service.) Modifying the application sources (if at all available) is not desirable from a maintenance standpoint, because the core application logic should ideally be kept separate from the incompatibility handling code. For this purpose, we allow the application developer to optionally provide a *custom handler* (written to well-defined conventions) that is interposed between the application and the cross-stub. The custom handler may use arbitrary code to handle the incompatibility. Another advantage of custom handlers is that they can be used to handle new types of incompatibilities. For example, extending the service derivation scheme of section 2 to allow derived services to add new fault types (i.e., exceptions) to existing operations results in a new type of incompatibility - mismatch in thrown fault types. Additional fault types introduced by the target non-native service can be handled in the custom handler.

During the resolution process, an application owner may determine that an incompatibility can not be handled and critically affects the functioning of the application, and thus abort the cross-stub generation process. In section 6, we examine how such aborts can be avoided by automatically determining beforehand whether the incompatibilities with a non-native service are critical.

In addition to implementing SV-incompatibility resolution, cross-stubs also handle *encoding incompatibilities* that arise because, as shown below, instances

of different XML schema types (such as etailer1's KSRequest1 and etailer2's KSRequest2) derived from a common base type are not identical even if they contain the same (non-null) elements and values.

```
<KSRequest xsi:type="KSRequest1">
  <keyword xsi:type="string">foo</keyword>
  <category xsi:type="Category1">Music</category>
  <minRating xsi:type="int" xsi:nil="true"/>
</KSRequest>
<KSRequest xsi:type="KSRequest2">
  <keyword xsi:type="string">foo</keyword>
  <category xsi:type="Category2">Music</category>
</KSRequest>
```

Depending on how etailer2 service is implemented, it may expect the type attribute to be set to "KSRequest2", and hence not recognize even an SV-compatible KSRequest1 instance. To address this, a cross-stub from services S to T is link-compatible with an S stub, but generates messages on the wire identical to those generated by a T stub.

5 Experimental Results

Application	Service	#meths	#used	#i/p	#filled	#o/p	#consumed
ORG #60	Google SearchService	3	1	10	10	27	6
ORG #69	Google SearchService	3	1	10	10	27	6
ORG #78	Google SearchService	3	1	10	10	27	6
ComparePop	Google SearchService	3	1	10	10	27	2
JDLS	Mappoint FindService	5	1	17	9	51	16
JDLS	Mappoint RenderService	4	2	327	39	48	17
JDLS	Mappoint RouteService	2	1	102	50	281	25
JDLS	All Mappoint Services	11	4	446	98	380	58
StoreLocator	All Mappoint Services	11	5	414	62	446	44
AW-apps	All Arcweb Services	45	13	446	134	210	83

Table 1. Usage statistics extracted by UAT-S for several applications. These results indicate that the applications only exercise a fraction of the service functionality. For example, JDLS only accessed 4 out of 11 methods, and even among these 4 methods, filled in only 98/446 input fields and consumed only 58/380 output fields. (The 446 inputs/380 outputs belong to the 4 accessed methods only, not all the 11 methods.) The Google applications filled in all inputs because they were forced to – the 10 inputs are simple type parameters rather than fields of a single structure – but they exercise only a small fraction of the input value space for these parameters. E.g., the language and country restricts parameters permit search customized to hundreds of language and countries, but these applications only use a few of these options.

Our implementation was primarily done within AXIS, an open source toolkit from Apache that implements static Java host types for WSDL services. Dynamic analysis and cross-stub generation was implemented by modifying the AXIS stub generators. UAT-S was implemented using joeq [8], which allows us to perform context-sensitive (but flow-insensitive) inter-procedural points-to analysis. We implemented an optional extension that allows the analysis to include primitives, i.e. an instruction that assigns the value 10 to a primitive int is considered as an allocation site that creates a `java.lang.Integer` object with value 10. In

some cases, UAT-S declares a primitive value "unknown" even if a more detailed analysis could have ascertained a known value, e.g. static evaluation (or partial evaluation) of arithmetic expressions; but we have found that this does not significantly reduce the effectiveness of the analysis.

We perform three sets of experiments. (1) Application usage behavior experiments, where we study what fraction of service functionality typical applications exercise. Non-native service interoperability is more likely to be successful if applications typically exercise only a small subset of the service functionality, because this small subset is likely also supported by non-native services. (2) Compatibility and integration experiments, where we demonstrate our tools and techniques in action. (3) Nature of incompatibilities experiments, where we study how often incompatibilities occur due to non-critical reasons. If incompatibilities frequently arise due to non-critical reasons, then non-native service interoperability can be further automated by employing *multi-option types* (to be described in section 6).

5.1 Application Usage Behavior

We examine several existing applications. The first three are taken from the O'Reilly book "Google Hacks", although we reimplemented them in Java. ComparePop is a Google application we wrote that compares relative popularity of search phrases across different languages (English, French, etc) based on the number of matches and where the words occur in the matching Web pages (title, URL or link). JDLS (Java Desktop Location Suite, a desktop application that provides maps, routes, panning/zooming, etc), StoreLocator (a JSP-based application created by a third-party vendor SpatialPoint) and AW-apps (collection of JSP applications accessing Arcweb location services) are available from the Mappoint and Arcweb web sites. Mappoint is a collection of three document style services (Find, Route and Render), while Arcweb, provided by the leading GIS software vendor ESRI, is RPC-style and consists of six different services. Both are quite complicated – for example, AXIS generates over 125 classes for Mappoint.

Table 1 shows the results produced by UAT-S on these applications. The results indicate that applications often only exercise a subset of features of the backend service they are written to, implying they can be compatible with non-native services even if the latter do not support all the features of the source service. (Also, the results of the next section show that most inputs/outputs used by these applications are supported by other services.) Thus, interoperation with non-native services is a reasonable proposition.

5.2 Compatibility and Integration

We first consider the scenario where an application owner needs to choose a suitable target service from among several non-native services available at a registry. In particular, we study compatibility of several search applications with five different search vendors: Google, AltaVista, AllTheWeb, Hotbot and Teoma. The Google WSDL service is considered the base service; we created WSDL services

Application	Google	AltaVista	AllTheWeb	Hotbot	Teoma
ORG#60-Google	TC				
ORG#60-AltaVista	√	TC			
ORG#60-AllTheWeb	√		TC		
ORG#60-Hotbot	√	√	√	TC	
ORG#60-Teoma	√	√	√		TC
ORG#69-Google	TC				
ORG#69-AltaVista	√	TC			
ORG#69-AllTheWeb	√		TC		
ORG#69-Hotbot	√	√	√	TC	
ORG#78-Google	TC	√	√		√
ORG#78-AltaVista	√	TC	√		√
ORG#78-AllTheWeb	√	√	TC		√
ORG#78-Teoma	√	√	√		TC
ComparePop-Google	TC		√	√	
ComparePop-AllTheWeb	√		TC	√	
ComparePop-Hotbot	√		√	TC	

Table 2. Compatibility of search applications with different vendors. (TC indicates trivially compatible, √ means compatible, i.e., all incompatibilities removed by CAT-S, while a blank indicates one or more relevant incompatibilities.) The row ComparePop-AltaVista is missing because ComparePop cannot be implemented using AltaVista, since one or more features needed by ComparePop is not provided by AltaVista. The same applies to other missing rows. These results show that when an application owner has a choice of several non-native services to use, (s)he can automatically determine which among them are compatible using our techniques.

for the other vendors based on the features supported in each vendor's "advanced search" Web pages. We paired each of four search-based applications—ORG#60, ORG#69, ORG#78 and ComparePop—with each of the five search services. Each application-service pair only makes calls that are legal for its native service. For example, ORG#60-AltaVista, an interactive application, allows a user to enter only queries containing operators supported by AltaVista. As shown in table 2, the owner can use CAT-S to automatically determine which among the potential target non-native services are compatible with the application at hand. For a compatible service, no resolution step is needed, and the owner can auto-generate the needed cross-stub.

We next consider a scenario where an application owner has pre-decided on a specific (possibly incompatible) target service, and the goal is to enable rapid integration. We consider two Mappoint applications and four Google applications and target services Arcweb and AltaVista respectively. Table 3 shows the effectiveness of automatic application usage inference during the integration process. Using the usage tuples inferred by UAT-S, CAT-S eliminates most of the incompatibilities, so the owner is only left with a few incompatibilities to resolve using the GUI tool.

5.3 Nature of Incompatibilities

Here, we study a number of incompatible application-service pairs to determine how often incompatibilities occur due to non-critical vs critical reasons. To maximize incompatibility cases, we study compatibility of Google applications with other vendors, since Google supports more features than others. Figure 5 shows that in as many as 33 out of the 56 incompatible pairs, the incompatibilities

Application	Service	#Total	#Relevant	Application	Service	#Total	#Relevant
JDLS	Find	25	1	ORG#60	Google	13	1
JDLS	Render	44	10	ORG#69	Google	13	1
JDLS	Route	50	2	ORG#78	Google	13	1
StoreLocator	Find	25	4	ComparePop	Google	13	2
StoreLocator	Render	44	10	-	-	-	-
StoreLocator	Route	50	0	-	-	-	-

Table 3. Number of incompatibilities (total and relevant) for two Mappoint and four Google applications with the non-native Arcweb and AltaVista services respectively. These results are based on CAT-S alone (i.e, CAT-D was not used here.) and demonstrate the effectiveness of application usage behavior inference in the integration process. Using the usage behavior data gathered by UAT-S, CAT-S is able to weed out most incompatibilities. Without CAT-S/UAT-S, the owner has to manually determine the relevance of each incompatibility.

do not affect the basic functioning of the application. This result is significant for two reasons. (1) the results of section 5.1 show that applications typically exercise only a fraction of the service functionality. Figure 5 suggests that even among the features exercised, many are not critically needed for the functioning of the application, further suggesting that interoperability with non-native services is practical. (2) the figure indicates that a mechanism for automated "white vs grey" determination is desirable, since this allows the "white" cases (i.e., critically incompatible cases) to be auto-filtered out without manual effort.

App	AltaVista	AllTheWeb	Hotbot	Teoma
60	SO	SO, F	D	SO, F
61	SC	SC	SC	SC
62		F	F	F
63	SO	SO, F	D	SO, F
65		R	R	R
67	SO	SO, F	SO, F	SO, F
68	SO	SO, F	D	SO, F
69	SO	SO, F	SO, F	SO, U, F
70		F	F	F
73	SO	SO, F	SO, F	SO, F
74-77	SO	SO, F	SO, F	SO, F
78		F	D, F	F
81	SS	SS	F	SS
83	T	T	T	T
84	CS	CS	CS	CS

Fig. 5. Compatibility of various O'Reilly Google applications with non-native vendors. Black indicates compatible, white indicates incompatibility due to a critical feature, and grey indicates incompatibility due to a non-critical feature. Each label identifies an incompatibility, e.g., SO = special operators (i.e., "file:pdf", "site:cnn.com", etc) F = filter (indicates if multiple results from same host should be filtered out). For legacy applications, the white vs. grey determination requires manual effort, whereas it can be automated for multi-option types.

6 Interoperation-friendly Applications

In this section, we examine how applications can be written ground-up for better interoperability. Specifically, if an application were explicitly authored to indicate *which* features are critical, then our algorithms could use this information to automatically resolve non-critical incompatibilities. Non-critical should not be confused with optional. (Recall that the service specification can declare a certain field as optional, e.g., `minRating` was declared optional in the schema definition of `KSRequest1`.) Whether a field is non-critical depends on application semantics. Much like WSDL, RPC systems in the past have often allowed *services* to specify (in their interface specifications) whether certain parameters are optional, but few systems allow a *client* to specify that a particular parameter or field that it supplies is non-critical.

Our goal is to specify the critical/non-critical information manually only once at application authoring time, and to use it automatically every time integration with a non-native service is desired. (An application is written once, but integrated multiple times during its lifetime.) To enable this goal, we propose *multi-option host types*.

6.1 Multi-Option Host Types

Example multi-option types for the KSRequest1 input message are shown below. (A different design based on generic types is possible in Java 1.5.)

```
public class M_JKSRequest1 {
  boolean non_critical = false; // ok to ignore?
  boolean ignored = false; // was it ignored?
  M_String category;
  M_String keyword;
  M_String minRating;
}
public class M_String {
  boolean non_critical = false;  // ok to ignore?
  boolean ignored = false; // was it ignored?
  String suppliedValue; // value supplied by app
  Object substitutes;  // other values to use if suppliedValue is disallowed
  String usedValue; // value that was actually sent to the service
  public M_String(String val) {
    suppliedValue = val;
  }
}
```

A multi-option simple type (such as `M_String`) is similar to the corresponding plain simple type (such as `java.lang.String`) except that in addition to supplying a value, a multi-option type provides a boolean `non_critical` indicating whether it is OK for the application if this field were altogether ignored, and an optional `substitutes` object providing a set of alternative values to use if the supplied value is not allowed by the service. A multi-option complex type (`M_JKSRequest1`) is similar to the corresponding ordinary complex type, except each of its fields is replaced by the corresponding multi-option type, and the boolean `non_critical` indicates if the whole subtree corresponding to this complex type can be ignored. Multi-option types are used as shown below:

```
M_String m_cat = new M_String("Books");
m_cat.substitutes = {``All''};

M_Integer m_rating = new M_Integer(4);
m_rating.non_critical = true;

M_JKSRequest1 req = new M_JKSRequest1();
req.category = m_cat;
req.rating = m_rating;
```

Instead of specifying values, the substitutes object can also specify *substitute patterns* (like the "s/../.." patterns in `sed` and `perl`) or *substitute ranges*. The former specifies that the value can be transformed using the specified pattern if need be. A substitute range indicates that any value in the range is acceptable. For applications using multi-option types, the cross-stub generator generates *best-effort cross-stubs* automatically from the source and target non-native service specifications. A best-effort cross-stub behaves similarly to a regular cross-stub, except (1) if it detects a missing field incompatibility, but the causal field is `non_critical`, it simply sets `ignored` to `true` and ignores the incompatibility, and (2) if it detects a facet mismatch incompatibility, but the causal field specifies substitute values one of which is compatible with the target service facets, `usedValue` is set to the compatible substitute value, and the incompatibility is ignored. (Substitute patterns and ranges are similarly handled.) Once the call is made, the application can determine if the value was ignored or substituted by the cross-stub, and (perhaps) notify the user:

```
stub.keywordSearch(req); // see req in previous code snippet
// we've called the service. were the values changed by the stub?
if (m_cat.usedValue != "Books" or m_rating.ignored) {
  // if need be, notify the user or log somewhere
}
```

The philosophical essence of multi-option types is to build substitutability from ground up – fine-grained field-level substitutability is an atomic building block for higher-level service substitutability.

6.2 Multi-Option Types in Use

Here, we discuss real examples of multi-option types in use. First, Mappoint applications such as JDLS and StoreLocator specify fonts and styles, and colors for routes/zones, which could be made `non_critical` and/or substitutable. Second, the O'Reilly Google applications specify `filter` and `safesearch`, but do not critically rely on them, so these can be made `non_critical`. (These parameters respectively indicate if multiple results from the same host and adult content should be excluded.) Finally, many O'Reilly Google applications allow special operators to be entered by the user, but can usefully function with services that disallow some special operators. So, a substitute pattern can be specified that removes the special operators.

For applications written using multi-option types, UAT-S records which fields were indicated as `non_critical`, and the provided substitute values/patterns/ranges (if any). The CAT-S rules of section 3.2 are updated as follows. If a missing field incompatibility i is considered relevant by the rules of section 3.2, but the field

is always declared `non_critical`, declare i as non-critical. If a facet mismatch incompatibility i is considered relevant by the rules of section 3.2, but substitute values or ranges are available, we check if any of the specified substitute values or any value in the substitute range is legal for the target service facets. If so, i is declared non-critical. The case when substitute patterns are available is a bit involved, and not implemented yet, but is theoretically tractable. Substitute patterns are finite state transducers, and since regular languages are closed under FST's, we can check if the regular language obtained by applying a substitute pattern to the source value space is a subset of the target value space.

If the Google applications of figure 5 were written using multi-option types, the new CAT-S rules would automate the "white vs. grey" classification, and non-critical incompatibilities will be automatically resolved by the cross-stubs.

7 Semantics and SV-Compatibility

The bigger picture is that we have facilitated automatic detection of SV-compatibility, but this does not assure semantic compatibility, i.e. it does not guarantee the service will "do what you want." Ideally, we want an application and service to be SV-compatible if and only if they are semantically compatible. We can approach this ideal if service creators follow these guidelines:

1. If a vendor restricts some base interface construct (method, type, type field or valuespace) or keeps it the same, he should keep its semantics the same as those of the corresponding base interface construct.
2. If a vendor extends by adding a method, type or type field, he should do so in his own namespace, unless he wants to reuse an extension already done by someone else, in which case he should import the construct from that party's namespace.
3. If a vendor extends the value space of some type/field, and uses facets to specify this, the system would know which values carried over from the base and which are added by each vendor. So, even if two vendors add the same value with different semantics, the system can infer that the value is not part of the base type and hence can conservatively assume that this value might have different semantics for each vendor.

The above conventions are similar to the *re-use namespace names* and the *new namespaces to break* rules of Orchard [9], and if adhered to will ensure that:

1. For methods, types and type fields, (a) SV-compatibility implies semantic compatibility, and (b) semantic compatibility implies SV-compatibility to the extent vendors reuse existing extensions from other namespaces;
2. For values, SV-compatibility will imply semantic compatibility, though not necessarily the converse.

8 Related work

We urge that vendors cooperate and derive their services from existing WSDL services. This may make economic sense even for competing vendors, since a derived service may help them draw traffic. However, we recognize that a competing

vendor may (e.g., for political reasons) choose to define a completely different autonomous WSDL specification. Building upon [10,11], we have partially addressed autonomous services interoperability before [12] by employing *adapters*. In future work, we combine UAT-S/CAT-S/CAT-D with adapters to better address the autonomous services case. However, unlike the semi-automatically generated cross-stubs, adapters must be completely handwritten, and thus autonomous services interoperability requires much more effort.

Referring back to the derived services case, service interface subtyping has been used for evolution, but with a base service V and two derivatives V_1 and V_2, traditional subtyping only handles two (namely $V \rightarrow V_1$ and $V \rightarrow V_2$) out of the six non-native service interoperation scenarios ($V \rightarrow V_1$, $V \rightarrow V_2$, $V_2 \rightarrow V$, $V_1 \rightarrow V$, $V_1 \rightarrow V_2$ and $V_2 \rightarrow V_1$). Also, traditional subtyping does not consider restriction. Compared to systems that address cross version/vendor substitutability without relying on subtyping [13,14,15,16], we leverage application usage behavior and multi-option types to customize the integration behavior to the application with minimal effort from the application developer/owner.

Schmidt et al. [17] explain the lack of practical versioning support in CORBA, while Vinoski [1] and Vogels [4] point out some of the interoperability/versioning issues with XML Web services. XDuce [3] provides static type checking for programs processing XML data, but does not target compatibility of applications exchanging XML messages over the network. Debugging tools such as SOAPscope [18] can check differences in WSDL documents, but UAT-S/CAT-S/cross-stubs go much further. David Orchard's guidelines [9] for XML schema versioning emphasize the need to provide *processing models* to deal with unrecognized schema components/extensions (i.e., incompatibilities). Our contribution is in allowing automated detection of which unrecognized components/extensions can break a *given application* and to specify a processing model (i.e., resolution choice) in an application-specific manner.

Multi-option types are most similar to Spreitzer's flexible types [19]), Orchard's "mustUnderstand" model [9] and the HTTP extension framework's non-ignorable bits [20]. These mechanisms at best only provide the equivalent of the non-critical flag, but not the substitute patterns, ranges or values. Furthermore, they rely on both client and server side support, while our multi-option types are entirely client-side and require no changes to wire protocols or server-side implementations. Schema evolution [21] and data integration [22] are well-researched, but do not leverage application usage behavior or provide multi-option types. Schema matching systems [23] attempt to auto-derive semantic relationships between different schemas; we rely on simple conventions, and perform no non-trivial semantics inference. Unlike semantic Web services [24], our approach does not require development of or global agreement on semantic markup languages.

9 Conclusions

Interoperability with non-native services is an enabling technology, because it assures application developers that their applications will stay relevant and usable across vendors, and hence promotes Web services adoption and application development. Our foray into this problem space reveals that many applications

only access a fraction of the service functionality, and even among the features accessed only a subset is critically needed. Leveraging these observations, we address the problem threefold: (1) static and dynamic analysis tools that automatically infer application behavior and determine compatibility with non-native services, (2) a GUI tool for resolving incompatibilities and generating cross-stubs, middleware components that actually enable the interoperation to occur, and (3) a lightweight mechanism called multi-option types that enables applications to be authored from ground up for better interoperability. We presented a set of experiments that, although by no means exhaustive, provide initial support for our claims and demonstrate the workings of our techniques.

Acknowledgements. We thank Godmar Back and John Whaley for implementation tips, and George Candea, Andy Huang, Emre Kiciman, John Mitchell, Terry Winograd and the anonymous reviewers for their feedback on prior drafts.

References

1. Vinoski, S.: The More Things Change... IEEE Distributed Systems Online **5** (2004)
2. Lee, K., et al.: Hydroj: object-oriented pattern matching for evolvable distributed systems. In: OOPSLA 2003, ACM Press (2003) 205–223
3. Hosoya, H., Pierce, B.C.: Xduce: A statically typed xml processing language. ACM Transactions on Internet Technology **3** (2003) 117–148
4. Vogels, W.: Web Services Are Not Distributed Objects. IEEE Internet Computing **7** (2003) 59–66
5. Stal, M.: Web services: beyond component-based computing. Commun. ACM **45** (2002) 71–76
6. Mikalsen, T., et al.: Reliability of Composed Web Services–From Object Transactions to Web Transactions. In: Workshop on Object-oriented Web Services, OOPSLA. (2001)
7. Berndl, M., et al.: Points-to Analysis using BDDs. In: PLDI, San Diego, CA (2003)
8. Whaley, J.: The joeq project pages (1999) `http://joeq.sourceforge.net`.
9. Orchard, D.: Versioning XML Vocabularies (2003) `http://www.xml.com/pub/a/2003/12/03/versioning.html`.
10. Gribble, S.D., et al.: The ninja architecture for robust internet-scale systems and services. Special Issue of Computer Networks on Pervasive Computing **35** (2001)
11. Vayssiere, J.: Transparent Dissemination of Adapters in Jini. In: DOA 2001, (Italy)
12. Ponnekanti, S.R., Fox, A.: Application Service Interoperation without Standardized Interfaces. In: IEEE Percom '03, Dallas-Fort Worth, TX (2003)
13. Senivongse, T.: Enabling flexible cross-version interoperability from distributed services. In: DOA 1999, IEEE (1999) 201–210
14. Evans, H., Dickman, P.: Drastic: A run-time architecture for evolving, distributed, persistent systems. In: ECOOP'97. Volume 1241 of LNCS., Springer (1997)
15. Ajmani, S., Liskov, B., Shrira, L.: Scheduling and simulation: How to upgrade distributed systems. In: Proc. of Ninth HotOS, Lihue, HI (2003)
16. Melloul, L., et al.: Reusable Functional Composition Patterns for Web Services. In: To Appear, IEEE ICWS. (2004)
17. Schmidt, D.C., Vinoski, S.: CORBA and XML, Part 1: Versioning (2001) `http://www.cuj.com/documents/s=7995/cujcexp1905vinoski/`.
18. Mindreef: (SOAPscope) `http://www.mindreef.com`.
19. Spreitzer, M., et al.: More Flexible Data Types. (In: Eighth IEEE WET-ICE 1999)
20. Nielsen, H.F., et al.: HTTP Extension Framework (1999) Internet draft.
21. Skarra, A.H., Zdonik, S.B.: The management of changing types in an object-oriented database. In: OOPSLA '86, ACM Press (1986) 483–495
22. Levy, A.Y.: Answering Queries using Views: A Survey. `www.cs.washington.edu/homes/alon/views.ps` (1999)
23. Rahm, E., Bernstein, P.: On Matching Schemas Automatically. Technical Report MSR-TR-2001-17, Microsoft Research, Redmond, WA (2001)
24. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems (Special Issue on Semantic Web) **16** (2001) 46–53