Ensuring Interoperable Service-oriented Systems through Engineered Self-Healing*

Giovanni Denaro
University of Milano-Bicocca
viale Sarca 336, Milano, Italy
denaro@disco.unimib.it

Mauro Pezzè
University of Milano-Bicocca
and University of Lugano
mauro.pezze@unisi.ch

Davide Tosi
University of Milano-Bicocca
viale Sarca 336, Milano, Italy
tosi@disco.unimib.it

ABSTRACT

Many modern software systems dynamically discover and integrate third party libraries, components and services that comply with standard APIs. Compliance with standard APIs facilitates dynamic binding, but does not always guarantee full behavioral compatibility. For instance, problems that derive from behavior incompatibility are quite frequent in service-oriented applications that dynamically bind service implementations that match API specifications.

This paper proposes a technique to engineer applications with a self-healing layer that dynamically reveals and fixes interoperability problems. The core elements of the technique are catalogs and a runtime infrastructure. Catalogs support developers in configuring the self-healing layers. The runtime infrastructure enacts the configured self-healing strategies. This paper discusses both the effectiveness of our solution and the relevance of the problem in the context of service-oriented applications, referring to a case study of Web2.0 social applications that integrate the standard APIs *del.icio.us* and *OpenSocial*. As an outcome of this experience we present an inconsistency catalog that supports the configuration of self-healing layers for Web2.0 applications.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Design

Keywords

Self-healing software, interoperable service-oriented applications, integration mismatches, integration faults

*This work has been supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC-FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.
Copyright 2009 ACM 978-1-60558-001-2/09/08 ...$5.00.

1. INTRODUCTION

The spreading of component-based and service-oriented technologies favors the emergence of libraries, components and services that offer the same or similar behaviors, and promote applications that dynamically discover and integrate alternative implementations of third-party software. The typical scenario exploits standard APIs that facilitate late binding between applications and third-party libraries, components and services. Compliance to standard APIs supports integration, but does not guarantee full behaviour compatibility. Standard APIs generalize the possible implementations, and abstract from several implementation-dependent details. Generalization and lack of details can cause interoperability problems and can lead to runtime failures.

We investigate this problem in the domain of service-oriented applications that rely on third-party services. We focus on scenarios where multiple providers offer services that implement standard APIs and provide compatible functionality. For these applications, interoperability problems are frequent, as we discuss in details in the next section.

In this paper, we propose a technique to engineer service-oriented applications with a self-healing layer that dynamically reveals and fixes mismatches between different implementations of services that comply with standard APIs. The approach relies on inconsistency catalogs to capture common interoperability problems that escape API compatibility checks. Client developers exploit the catalogs to identify potential mismatches between different implementations of the same APIs, and configure the self-healing layer accordingly. They configure self-healing layers by generating compatibility test suites and corresponding adaptors. Compatibility test suites and adaptors are paired: Compatibility test cases reveal possible mismatches, and trigger adaptors that solve them. We refer to the pairs of compatibility test cases and adaptors as *test-and-adapt plans*. The self-healing layer is a runtime infrastructure that executes the compatibility test suites when new implementations are bound at runtime, automatically detects the occurrence of mismatches, and dynamically triggers the suitable adaptors.

Interoperability problems have been investigated in the context of design for change, service composition, semantic web engineering and service interchangeability.

Design for change approaches support the design of evolving APIs. Enabling interoperability with multiple implementations of a standard API shares some aspects with supporting evolving APIs, but entails additional challenges that call for novel approaches, as the one proposed in this paper.

For instance, services from different providers are competing rather than evolving API implementations, and in this scenario we cannot count on references to previous versions, as assumed by most approaches to evolving APIs ([5], [29]).

Assertion-based dynamic analysis and formal verification of service composition focus on contract violations and can check for compatibility of services when formally specified, thus complementing our approach [1, 9, 2].

The work on semantic web and service interchangeability is closely related to our research. Semantic web technology and ontologies can solve semantic ambiguities, but provide little guidance to identify and diagnose application mismatches [24, 14]. Approaches to interchangeable services support interoperability of applications with services other than the ones the application was originally written for, but the work done so far considers mostly structural mapping between non-standard service interfaces, and dismisses semantic aspects of different implementations of standard interfaces [22, 21, 12, 16]. We provide additional details about these approaches in Section 7.

This paper contributes to the scientific and technical knowledge in many ways. It provides empirical evidence of interoperability problems between applications and implementations of standard APIs by reporting experience in the emerging domain of social networking applications (Section 2). It proposes an original self-healing approach for detecting and solving interoperability problems at runtime, referring to the experience acquired by testers during field testing and maintenance, and formalized by developers at design time (Section 3). It reports experience results that indicate the effectiveness of the proposed approach (Section 4). It suggests a first generalization of the results by introducing an inconsistency catalog to identify common mismatches and healing strategies (Section 5), and discusses the impact of the approach on performance (Section 6). It discusses the novelty of the approach by surveying related work (Section 7). Finally, it indicates research directions related to the results presented in the paper (Section 8).

2. SERVICE INTEROPERABILITY

Problems of interoperability due to dynamically bound implementations of standard APIs affect applications built with several modern software engineering technologies and paradigms, such as dynamic-linking libraries, component-based middlewares and service-oriented architectures. To make our discussion more concrete, this section exemplifies this type of interoperability problems with reference to service-oriented applications that integrate competing implementations of standard service APIs.

We report results of two studies in which we investigate the interoperability between applications that integrate third-party services through a standard API, and different implementations of the APIs. In the first study, we consider applications for bookmark handling that integrate social bookmarking services offered through the standard *del.icio.us* API, to store and retrieve bookmarks online. We executed four applications each integrated with four compatible implementations of the *del.icio.us* API. In the second study, we consider social networking applications, where network containers that manage social networks of users host social network gadgets that provide user-oriented functionality across a social network. For example, *facebook.com* is a popular social network container, and *BizX* (from *toostep.com*) is a social network gadget that lets users create virtual business cards and distribute them across the network of friends. We executed five social network containers that use the *OpenSocial* API, the most popular standard for the interactions between gadgets and containers, each with eight social network gadgets.

App	**#tests**	**#failures with del.icio.us-based web**			
		D	M	F	L
DEL1.14	35	0	9	-	3
GAD	15	0	4	3	2
BtoD	7	0	0	1	0
SABROS.us	4	0	1	2	1

Legend:
D: `delicious` [`del.icio.us`] F: `faves` [`faves.com`]
M: `magnolia` [`ma.gnolia.com`] L: `link-wieza` [`link.wieza.net`]
GAD: `GUI+del.icio.us` DEL1.14: `del.icio.us java API`
BtoD: `Bookmarks To Delicious`
-: not-tested because of incompatible authentication mechanisms between `DEL1.14` and `faves`

(a)

App(gadget)	**#tests**	**#failures with OpenSocial container**				
		O	H	M	F	I v0.1
BizX	16	0	4	-	1	3
BuboMe	9	3	2	-	4	3
BuddyPoke	12	1	2	-	-	-
Emote	7	0	0	-	-	1
LastFM	6	0	0	-	0	0
RateMyFriends	6	0	0	2	2	2
Unype	11	0	1	1	0	1
Zorap	3	0	0	0	0	0

Legend:
O: `orkut` [`www.orkut.com`] M: `myspace` [`www.myspace.com`]
H: `hi5` [`www.hi5.com`] F: `facebook` [`www.facebook.com`]
I: `imeem` (v0.1 released in mid-May) [`www.imeem.com`]
-: not-tested because installation of the gadget failed

(b)

Table 1: Integration failures with different API implementations

Table 1 reports the results of the studies. The tables report the number of test cases that have been executed for each application and each implementation of the standard service APIs, and the number of experienced integration failures. The test cases revealed 26 integration failures for the implementations of the *del.icio.us* API, and 33 integration failures for the implementations of the *OpenSocial* API. Notice that all *del.icio.us* clients in the experiment were originally designed and tested using *del.icio.us* as target web service, as confirmed by the results of our tests that did not reveal failures for *del.icio.us* (column $\langle D \rangle$ of Table 1 (a)). Additional details on the case studies can be found in [26].

In-depth analysis confirms that the definitions of the two standard APIs focus on syntactic and type issues, but leave several semantic details underspecified. This leads to inconsistent implementations of the APIs and consequent failures when applications integrate different API implementations.

We discuss the issue referring to gadget *BizX* that was originally developed for *orkut*, and executes correctly within this container, but fails when integrated in other containers (row $\langle BizX \rangle$ of Table 1 (b)). Table 2 shows the details of the implementations of the *OpenSocial* API that are respon-

BizX assumptions	Container Implementation choices orkut	hi5	facebook	imeem v0.1	Failure
Thumbnails < 70x70	64x49	100x100	50x37	100x100	flawed vCard layout
getDisplayName retrieves two strings	two strings	one string (name)	two strings	two strings	null pointer exception
In URL: Userid as URL?uid=ID	URL?uid=ID	URL?userid=ID	URL?id=ID	URL/people/ID	TypeError: User has no properties
Activities supported	supported	supported	supported	unsupported	Activities never created
Activity body supported	supported	unsupported	supported	N/A (see above)	flawed activity view

Table 2: Analysis of failures of the gadget BizX when integrated with different OpenSocial containers

sible for the failures of *BizX* in our study. Semantic details that escape the definitions in the *OpenSocial* API results in inconsistent implementation choices in the containers.

For example, row $\langle Thumbnails \ldots \rangle$ of Table 2 indicates that different containers return thumbnail images of different sizes. The implementation choices of containers *hi5* and *imeem* are incompatible with *BizX* assumptions and cause failures. Row $\langle In\ URL \ldots \rangle$ of Table 2 indicates different format conventions for the userid within a URL, that lead to failures when *BizX* is integrated in all containers other than *orkut*. Row $\langle Activity\ body \ldots \rangle$ of Table 2 indicates that in some cases the API is only partially implemented by the containers, leading to failures when invoking the unsupported operations.

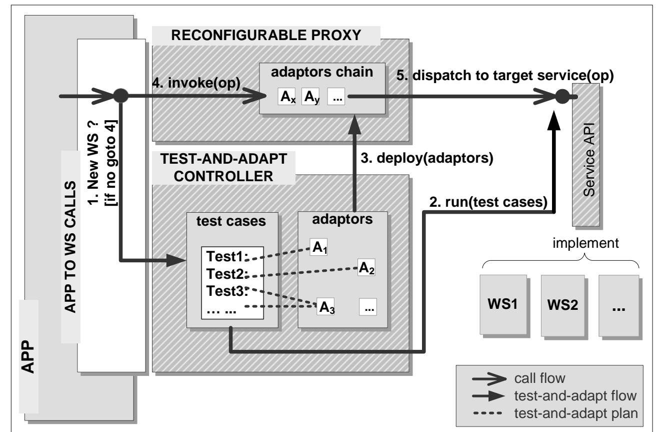

Figure 1: Test-and-adapt: runtime aspects

3. THE TEST-AND-ADAPT PARADIGM

Most problems that derive from inconsistent implementations of standard APIs do not preclude the interoperability of the applications with different implementations of the same API. In all our experiments, the different implementations of the service APIs, albeit inconsistent, preserve the main service functionality.

Many inconsistencies can be predicted by analyzing the standard APIs on the basis of domain expertise and previous experience, identified by executing few simple test cases on the target implementations, and solved by means of simple adaptors. Following this observation, we propose a mechanism that augments applications with test cases and adaptors that provide self-healing capabilities to dynamically solve interoperability problems across inconsistent implementations of standard APIs.

Our mechanism, hereafter *test-and-adapt*, includes design and runtime aspects. At runtime, applications dynamically execute test cases to verify the consistency of the current implementation of a standard API, trigger suitable adaptors if needed, and use the selected adaptors to mediate the next interactions through the API. At design time, applications must be engineered with *test-and-adapt plans*. We define a test-and-adapt plan as a relation between test cases that check for inconsistency, and adaptors that fix the inconsistency at runtime. Each test case is paired with an adaptor that is activated depending on the test outcome.

Figure 1 illustrates the runtime mechanism for the case of standard API web services. The mechanism includes a *Reconfigurable proxy* that dynamically binds adaptors to service invocations, and a *Test-and-adapt controller* that manages test-and-adapt plans. The figure illustrates the two invocation flows for the cases of a newly-bound and in-use service implementation, respectively. If the runtime mechanism identifies that a new implementation has been bound to the service API (for instance, because of a new provider or because of the notification of a service update), it first activates the *Test-and-adapt controller* that executes the test cases associated with the API at design time, and then deploys suitable adaptors according to the test results (steps 1, 2, and 3 in Figure 1). When a service implementation is in-use, the application invokes the service through the *Reconfigurable proxy* that is responsible for executing the adaptors that may have been formerly selected for the current implementation (steps 1, 4 and 5 in Figure 1).

Step 2 requires the execution of test cases, causing the service to be invoked out of the application flow. To avoid loss of integrity, we assume either no side effects on the services or the availability of sandbox execution environments, as increasingly offered by service provides, like in the case of our experiments with social containers. We share this requirement with other approaches based on on-line testing [28], like the in-vivo testing [6] and the metamorphic testing approaches [3].

Devising test-and-adapt plans is the design-time core of the approach. As when designing modern software systems, engineers shall identify unexpected executions and design exceptions handlers, when dealing with dynamically bound implementations of standard APIs, software engineers shall identify potential inconsistencies that may arise from different implementations of the same API, produce test cases to reveal inconsistent implementations, and design corresponding adaptors. Pairing adaptors with test cases makes the approach suitable to cope with incompatibilities that may arise when referring to implementations not known at design

time. Engineers can identify potential inconsistencies either from their previous experience or through inconsistency catalogs that capture experience of software designers and domain experts. The inconsistency catalogs can be specialized on the application domains and the execution environments, and may evolve over time to optimize the process.

Test-and-adapt plans cannot cope with all possible inconsistencies, but can take care of some classes of problems, in particular problems that derive from identifiable weaknesses of APIs. As all test based approaches and most self-healing solutions, we aim to solve some, but not all problems. Solving all problems is not a realistic goal, but being able to automatically solve some problems at runtime represents a big improvement, since it reduces system malfunction and downtime. Richer service specifications, if available, can disambiguate API inconsistencies in some cases; in the future, we aim at integrating specification checking to complement test-based diagnosis in our approach.

4. EXPLORATIVE STUDY

We experimented the test-and-adapt approach with two sets of service-oriented applications that use the *del.icio.us* social bookmarking and the *OpenSocial* APIs, respectively, and we studied the interactions of the applications with different implementations of the considered APIs. In Section 2, we already discussed the empirical data that witness the relevance of the problems of interchanging applications and API implementations. In this Section we evaluate the test-and-adapt paradigm: We investigate the possibility of generating test-and-adapt plans, and study the efficacy of test-and-adapt plans to identify and solve inconsistencies between applications and service implementations. In the next sections, we generalize data from experiences to derive an initial inconsistency catalog, and we discuss performance issues.

We illustrate the results of evaluating the test-and-adapt approach incrementally, by first discussing the data collected on applications that integrate *del.icio.us* services, and then showing how these data generalize to applications that integrate *OpenSocial* services.

Del.icio.us.

Del.icio.us is a social bookmarking web service for storing, sharing, and discovering web bookmarks. The original implementation of the service (acquired by Yahoo in 2005) has become very popular over the last years, and at the time of writing it serves a community of millions of users, and hosts more than 100 million bookmarked URLs. As a consequence of this increasing trend of popularity, other vendors started providing web services compliant with the *del.icio.us* API, to facilitate both reuse of know-how and integration of existing applications. Today, the *del.icio.us* API[1] is a *defacto* standard API with multivendor implementations.

Listing 1 shows, in java-like notation, an excerpt of the *del.icio.us* API, as relevant to the writing of this section. The API defines three classes of operations: *posts* operations allow for retrieving (specific, all, or recently modified), adding, updating and deleting bookmarks; *tags* operations allow for retrieving and renaming tag keywords that can be associated with the bookmarks for classification purpose; *tags.bundles* operations provide an auxiliary functionality to define, retrieve and delete semantically coherent sets of tags.

[1]http://delicious.com/help/api

```
posts.get(String filterByTag, String filterByDate,
     String filterByUrl)
posts.all(String filterByTag)
posts.recent(String filterByTag, int maxItems)
posts.dates(String filterByTag)
posts.add(String url, String description, String
    tags, String date)
posts.update()
posts.delete(String url)
tags.get()
tags.rename(String old, String new)
tags.bundles.set(String bundle, String tags)
tags.bundles.all()
tags.bundles.delete(String bundle)
```

Listing 1: Excerpt of the del.icio.us API

Integration Fault	Description
Inconsistent interpretation of parameters or values	Each module interpretation is reasonable, but they are incompatible
Violations of value domains or capacity/size limits	Implicit assumptions on ranges of values or sizes
Side-effects on parameters or resources	Implicit effects of a module on resources that are not explicitly mentioned in its interface
Missing or misunderstood functionality	Incorrect assumptions about expected results due to underspecification of functionality

Table 3: Taxonomy of integration faults

We applied the test-and-adapt approach to the applications of Table 1 (a), focusing on how they use the *del.icio.us* API. The test-and-adapt approach looks for ambiguity and incompleteness of the API specifications that can lead to inconsistent implementations, and can thus cause failures when integrating different implementations. We generated a first set of inconsistencies by referring to a taxonomy of classic integration faults as outlined in Table 3 (borrowed from [19]). Since the considered applications use subsets of the same standard API, we derived a general set of test-and-adapt plans for the standard API, and we tailored the plans to the different applications.

We used the taxonomy as a checklist, we scanned it sequentially, and we applied each entry to all items in the *del.icio.us* API. We identified 15 sources of inconsistency that may originate mismatched service implementations, and we designed corresponding test-and-adapt plans. We then deployed the test-and-adapt plans as modified JUnit test cases, which can be launched by the applications at runtime and whose failure activates the adaptors required by the plans. Adaptors are implemented as proxy components that mediate calls to the API. We modified all four client applications in the experiment to run the test cases when connecting to a new provider, and to delegate service calls to the adaptors activated as a result of executing the test cases.

In Appendix A, we show the sources of inconsistencies, mismatches and test-and-adapt plans that are discussed in this section. The complete set of inconsistencies, mismatches and test-and-adapt plans is available in [26].

Here we illustrate the process of devising the test-and-adapt plans by referring to the first entry of the taxonomy in Table 3: *Inconsistent interpretation of parameters or val-*

	Ma.gnolia	Faves	Link-wieza
DEL1.14	A1, A2, A5, A8, A13, A15	n.a.	A2, A8
GAD	A1, A2	A1, A2, A14	A2
BtoD	-	A14	-
SABROS.us	A2	A2, A14	A2

Adaptors are listed by giving the label presented in Appendix A.
n.a indicates blocking dependencies on other APIs or libraries.
- stands for no adaptor deployed.

Table 4: Adaptors deployed for *del.icio.us*

ues. We scanned the API, and we identified several parameters that are underspecified and thus can cause integration failures when interpreted differently in different implementations. For instance, item $S1$ in Appendix A refers to parameters of type *string* that indicate lists of tags (used by several operations, for instance */posts/add*): These parameters are underspecified because the type *string* does not indicate the separators between tags, which can be implemented in several inconsistent ways. Having identified this class of potential mismatches ($M1$ in Appendix A), we designed a test-and-adapt plan to reveal and solve them (T1 and A1 in Appendix A): The test cases execute the target service with different separators to identify the separator used by the current implementation, and the adaptors rewrite the strings according to the identified separator, thus assuring the consistency of the interactions. Test cases and adaptors refer to a set of common separators, thus are not universal, but they are likely to work in many common cases.

To validate our work, we re-executed the 12 combinations of the clients and the service implementations alternative to the original *del.icio.us* implementation indicated in Table 1. In this experience, the adaptors automatically solved all the mismatches that cause the failures listed in Table 1. Table 4 reports the adaptors that have been deployed at runtime by the test-and-adapt infrastructure when executing the benchmark applications.

OpenSocial. To increase the confidence in the results obtained from the experience with *del.icio.us*, we replicated the experiment with another set of service-oriented applications that use services through a standard API.

OpenSocial is an API for web-based social network applications, released by Google in 2007[2]. A web-based social network is a thematic web-portal (or *container*) that lets users define mutual connections in the form of friendship relationships, and that exploits these relationships to elicit information that can be shared among users. A typical functionality of a social network container is to enable users to declare other users as friends, and notify updates of people profiles to friends. Containers host *gadgets*, applications that provide user-oriented functionality across a social network. The *OpenSocial* API is currently supported by several social network containers and gadgets. The *OpenSocial* API is based on HTML and JavaScript, and defines the support for gadgets to access data and core functions of a social network on a container that implements the API.

We proceeded as in the previous experiment. We analyzed

[2]http://code.google.com/apis/opensocial/docs/0.7/reference

	containers as in Table 1					
gadget	O	H	M	F	I v0.1	I v0.2
BizX	-	A2, A14, A29, A30	n.a.	A14	A2, A5, A14, A30	A2, A14, A30
BuboMe	A2, A7, A17, A27	A7, A17, A26	n.a.	A2, A7, A17, A26	A2, A5, A7, A17, A26	A2, A7, A17, A26
Buddy Poke	A11	A2, A11, A19	n.a.	n.a.	n.a.	n.a.
Emote	-	-	n.a.	n.a.	A5	-
LastFM	-	-	-	-	-	-
RateMy Friends	-	-	A14	A14	A5, A14	A14
Unype	-	A29	A29	-	A5	-
Zorap	-	-	-	-	-	

Column headers follow the conventions of Table 1 (b).
Adaptors are listed by giving the label presented in Appendix A.
n.a indicates blocking dependencies on other APIs or libraries.
- stands for no adaptor deployed.

Table 5: Adaptors deployed for *OpenSocial*

all items in the API, we identified a total of 30 sources of inconsistency that may originate mismatching service implementations, and we designed corresponding test-and-adapt plans. Appendix A samples the sources of inconsistencies, mismatches and test-and-adapt plans. Table 5 reports the adaptors that have been deployed at runtime by the test-and-adapt infrastructure when executing the benchmark applications, and that solved all failures indicated in Table 1.

The data in the last two columns of Table 5 refer to two subsequent versions of the container *imeem*: versions *v0.1* and *v0.2*. We ran the test-and-adapt plans developed for previous containers also for these two versions of *imeem*. They successfully identified and fixed incompatibilities, thus guaranteeing the interoperability of the new containers with the gadgets considered in our study. The only exception is the gadget *BuddyPoke*, whose incompatibility problems depend on APIs that we did not consider in our study.

5. INCONSISTENCY CATALOG

The explorative studies reported in the previous sections provide consistent data that confirm our hypotheses: API compatibility does not guarantee complete service interoperability; The identified problems can be revealed with simple test cases, and can be fixed with simple adaptors; Test cases and adaptors comprise test-and-adapt plans that serve different cases. The experience with the *del.icio.us* and the *OpenSocial* APIs and the analysis of other services common in Web2.0 social applications indicate classes of inconsistencies that both recur in standard APIs and map to common inconsistency patterns, in this domain.

In this section, we generalize the experimental data collected so far into an initial *inconsistency catalog* tailored to Web2.0 social applications. The Catalog, reported in Appendix B, is organized in sections that group classes of inconsistencies that refer to the same aspects: parameters, state data, functionality and semantics. The catalog entries capture sources of possible inconsistencies between Web ap-

plications and implementations of the service APIs (item Si of the catalog entries), describe the related possible misinterpretations (Mi in the catalog), indicate the test cases that can reveal the inconsistency (Ti), and suggest possible adaptors (Ai).

The first section of the catalog (*parameters*) describes classes of inconsistencies that derive from inconsistent interpretations of service parameter types that do not specify completely the nature of the values to be assigned to the actual parameters. Parameters with these characteristics show up frequently in APIs for Web2.0 social applications, which manipulate many textual data, URLs, item lists, and that easily dismiss strongly typed systems for the sake of flexibility.

The second section of the catalog (*state data*) describes classes of inconsistencies that derive from service-side data underspecified in the APIs. Both inconsistent bounds on the capacity of stored collections and missing support for arbitrary sets of non-mandatory data fields occur in several API for Web2.0 social applications, which entail implicit server-side collections and data structures with non-mandatory fields.

The third section of the catalog (*functionality*) describes classes of inconsistencies that derive from partial implementations of APIs. This is a fairly common choice of providers who may not see the convenience of implementing rare or unessential operations. Standard APIs pursue completeness, and thus include also operations that implement side functionality, sometimes perceived as non-essential and ignored in some implementations. For instance, *ma.gnolia* and *link-wieza* do not implements handling of bundles functionality specified in the *del.icio.us* API. Standard APIs often include operations that can be derived by specializing or combining other operations. Derived operations may be considered redundant and ignored in some implementations. For instance, *ma.gnolia* does not implement the /tags/rename functionality specified in the *del.icio.us* API, since it can be obtained by combining /posts/all, /posts/delete, /posts/add. Standard APIs sometimes include operations that work in different modes, that is, serve classes of requests distinct by means of parameter options. Some implementations do not provide the operations for options that are not considered particularly relevant. For instance, *orkut* and *hi5* do not implement all modes of the *OpenSocial* send message API that defines public, private, email, and acknowledge modes. Standard APIs include operations that serve only usability issues, and are sometimes ignored in implementations. For instance, operations to display the results in different orders may not be always implemented. Standard APIs may define asynchronous notifications through callbacks. Some implementations ignore the callbacks but serve the related operations.

The last section of the catalog (*semantics*) describes classes of inconsistencies that derive from inconsistent handling of service semantics. Sometimes, standard APIs do not completely specify error/success codes, and different implementations may code them referring to inconsistent conventions. For instance, the *OpenSocial* API does not fully specify error codes returned by operations for fetching people, when the containers do not implement all required fields. The implementations that we considered return inconsistent error codes. Several standard APIs include operations that remove items from collections, but only some implementations return an error code when the item to be removed does not belong to the collection. Similarly, operations for inserting key-value pairs in maps entail the special case of items whose key is already in the map, which admits either overriding or non-overriding semantics.

API	Impl.	#test cases	Test run
Del.icio.us	del.icio.us	10	43.88 sec.
	ma.gnolia	10	64.68 sec.
	link.wieza	10	23.71 sec.
OpenSocial	orkut	16	3.10 sec.
	hi5	16	3.58 sec.
	mySpace	16	4.32 sec.
	facebook	16	4.12 sec.
	imeem	16	4.47 sec.

Table 6: Mean execution time of test suites from test-and-adapt plans

6. PERFORMANCE CONSIDERATIONS

Checking for large amount of potential mismatches can have relevant performance implications. Table 6 reports the mean execution time of the test suites of the test-and-adapt plans defined in our experiments against different implementations of the *del.icio.us* and *OpenSocial* APIs. The data confirm that executing test cases when applications are distributed over a network (as in the case of applications that use *del.icio.us*, *mag.nolia*, and *link-wieza* web service implementations) is significantly slower than when applications are executed on the same machine (as in the case of *OpenSocial* applications, where gadgets and containers execute at server-side).

The test overhead has a perceivable impact especially when the services are executed over the network. However, we observe that the test cases are executed only when binding new service implementations. This does not happen frequently, and entails high risks of runtime failures. We believe that trading performance for dependability is acceptable, especially if performance is only seldom affected, and the dependability threats may prevent the access to the desired functionality.

We foresee scenarios in which test cases are executed before the actual interactions between the applications and the service implementations, thus avoiding performance impacts. If the binding between the services and the applications is statically configured at deployment-time, test-and-adapt plans can be executed for each new deployment, with no impact on the runtime performance. If an infrastructure is extended to enable applications to notify providers or brokers of their test-and-adapt plans, these can be executed off-line before the applications access the services.

7. RELATED WORK

In this section we discuss previous work on service interchangeability, compare our approach to semantic web technologies, and acknowledge influential research in autonomic and self-managed systems.

Service interchangeability. Service interchangeability focuses on interoperability between applications and services that fulfil equivalent goals, but are designed independently

by different vendors and are not always fully compatible. Ponnekanti and Fox describe a technique that retrieves service interface adaptors from a repository, and chains them if the target interface of an adaptor matches the source interface of another adaptor, thus deriving adaptors for unsupported interface pairs [21]. Motahari Nezhad et al. study how adaptors derived from WSDL specifications can map between the interaction protocols of compatible services [16]. These adaptors are limited to interface incompatibilities that can be solved by implementing mappings among method names, parameters and protocols of different APIs. Ponnekanti and Fox propose a static analysis technique to select, out of a pre-computed set of service mismatches, the mismatches that are relevant for different clients [22]. For these mismatches, they provide adaptors (cross-stubs) that intercept the mismatches during service invocations and throw a runtime exception, supply pre-defined values, or ignore the problem and continue. Kaminski et al. propose to deploy chains of adaptors at service-side to guarantee backward compatibility for clients of previous versions of evolving services, moving the responsibility of maintaining compatibility from client to service providers [12].

Our approach differs from previous work on service interchangeability in several ways. We consider different implementations of standard APIs. In this context, the API uniquely defines messages and types for all complying service implementations. Thus we do not need to adapt interfaces, but we cope with integration mismatches that are impossible to detect statically for services that comply with the same standard specification. None of the above approaches describes how to cope with this problem. We propose inconsistency catalogs that survey the weaknesses of a standard API to discover potential integration mismatches, and use dynamic testing to detect at runtime the occurrence of such mismatches.

The cross-stubs proposed by Ponnekanti and Fox are configured and deployed by developers on a per-case basis, and can only skip or propagate failures, or mask them by supplying predefined values. Our adaptors are automatically configured and deployed at runtime based on policies that depend on the results of the dynamic testing stage, and can provide suitable recovery strategies specialized from libraries of common adaptors. Furthermore, while previous approaches deal with syntactic and structural problems only, our inconsistency catalog includes also some semantic inconsistencies and describes the related adaptation plans.

Our experiments confirm the observation that clients of standard APIs use subsets of the API functionality, thus a static analysis technique like the one defined by Ponnekanti and Fox is in principle applicable to select, among the potential mismatches identified for an API specification, the subset that is relevant for a client. This does not eliminate the need for runtime testing, but would allow for filtering out irrelevant test cases, thus limiting the runtime overhead. We plan to integrate this static analysis as future work.

Semantic web technology. Semantic web technology and ontologies provide reference frameworks of concepts that describe relevant semantics of web resources, and support automatic reasoning to solve semantic ambiguities ([24, 14].) The main hinder to using these approaches in practical applications is the difficulty of defining generally agreed domain ontologies. To overcome this limitation, several technologies for *semantic web services* have emerged in the last years as an attempt to enrich web service languages with ontological annotations, aimed to facilitate discovery and lower interoperability barriers [15, 23, 11, 25, 18]. Here we discuss the WSMO/WSMX stack of technologies that explicitly address web services adaptation.

The web service modeling ontology (WSMO, [23]) enables semantic descriptions of both web services and client goals. These descriptions can be matched to facilitate service discovery and interaction. The authors of WSMO recognize that heterogeneity naturally arises in open and distributed environments, and propose *mediators* to address mismatches that can derive from heterogeneity of resources that ought to be interoperable. Typically foreseen tasks of mediators include translating reference ontologies, adapting communication protocols, and reconfiguring service workflows. WSMO tools include the web service execution environment (WSMX, [11]), a software system that execute WSMO-based web services and the related mediators.

Technologies such as WSMO and WSMX albeit promising are still in their infancy. We acknowledge them the merit of pointing out the necessity of coping with interoperability mismatches. However, they provide no guidance to identify likely mismatches and automatically diagnose their occurrence when applications are connected to new implementations of a web service. Current mediators are mostly limited to well standardized adaptations, such as applying predefined mappings between ontologies and protocols. It remains still unclear if general purpose adaptations can be designed with these technologies.

Our approach proposes guidelines to identify mismatches, and devise diagnosis test cases accordingly. Our adaptors exploit the full power of a programming language, and can realize general purpose adaptations including partial recovery and failure masking. We are currently investigating synergies between semantic web technologies and our ideas, aiming to use ontological reasoning to complement testing-based diagnoses, and mediators to serve as part of the adaptation tasks.

Autonomic and self-managed systems. Autonomic computing and self-managed systems propose feedback loops to monitor software executions and keep application parameters under control [13]. In the last few years, researchers have investigated this notion in many application domains with different goals and approaches, including self-configuring software architectures ([4, 17, 27]), self-adaptive use and allocation of computing resources ([7, 20]), self-healing mechanisms ([8]), and programming models inspired from biology ([10]). Our work exploits the ideas of autonomic computing and self-managed systems to guarantee the interoperability of clients with remote web services with many independent implementations.

8. CONCLUSIONS

This paper discusses the problem of service interoperability that derives from incompatible implementations of standard APIs, and proposes a self-healing solution. It illustrates the growing relevance of the problem by providing data from experience in the important domain of Web2.0 social networking applications, and proposes a technique that incorporates experience into test cases and adaptors to automatically heal problems in deployed applications. It illus-

trates the applicability of the approach by showing experiences with some relevant applications, and proposes a first generalization by defining an inconsistency catalog for the domain of Web2.0 social applications. The catalog captures experience and supports self-improving.

We believe that the self-healing approach proposed in this paper can be applied to a wide range of applicative domains and technology frameworks. The approach does not have strict technology constraints, and can be generalized beyond the domain of service-based applications. Further controlled experiments with implementations of standard APIs can refine our empirical data about the effectiveness of the approach, identify new relevant applicative domains, and generalize interface requirements and common inconsistencies. Moreover, we are currently investigating how to make adaptations fully compositional, independently of the order in which the test-and-adapt plans are executed.

9. REFERENCES

[1] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC)*, pages 269–282, 2005.

[2] D. Beyer, A. Chakrabarti, and T. A. Henzinger. An interface formalism for web services. In *Proceedings of the 1st International Workshop on Foundations of Interface Technologies (FIT)*, ENTCS. Elsevier, 2005.

[3] W. Chan, S. Cheung, and K. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2006.

[4] S. Cheng, A. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, pages 276–277, 2004.

[5] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, Nov. 1996.

[6] M. Chu, C. Murphy, and G. E. Kaiser. Distributed in vivo testing of software applications. In *Proceeding of the International Conference on Software Testing, Verification, Validation (ICST)*, pages 509–512, 2008.

[7] J. Cobleigh, L. Osterweil, A. Wise, and B. S. Lerner. Containment units: a hierarchically composable architecture for adaptive systems. In *Proceedings of the tenth Symposium on Foundations of Software Engineering (FSE)*, pages 159–165. ACM Press, 2002.

[8] C. Dabrowski and K. Mills. Understanding self-healing in service-discovery systems. In *Proceedings of the 1st Workshop on Self-healing Systems (WOSS)*, 2002.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 738–741, 2004.

[10] S. George, D. Evans, and L. Davidson. A biologically inspired programming model for self-healing systems. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS)*, pages 102–104, Nov. 2002.

[11] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. WSMX - a semantic service-oriented architecture. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 321–328, 2005.

[12] P. Kaminski, H. Müller, and M. Litoiu. A design for adaptive web service evolution. In *Proceedings of the International Workshop on Self-adaptation and self-managing systems (SEAMS)*, pages 86–92, 2006.

[13] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[14] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax, W3C recommendation 10 Feb. 2004. http://www.w3.org/TR/rdf-concepts.

[15] D. Martin, M. Burstein, O. Lassila, M. Paolucci, T. Payne, and S. McIlraith. Describing web services using OWL-S and WSDL. In *DAML-S Coalition working document*, 2003.

[16] H. R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 993–1002, 2007.

[17] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[18] M. Paolucci and M. Wagner. Grounding OWL-S in WSDL-S. In *Proceedings of the International Conference on Web Services (ICWS)*, Sept. 2006.

[19] M. Pezzè and M. Young. *Software testing and analysis*. John Wiley & Sons, 2008.

[20] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw. Dynamic configuration of resource-aware services. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 604–613, 2004.

[21] S. Ponnekanti and A. Fox. Application-service interoperation without standardized service interfaces. In *Proceedings of the Pervasive Computing and Communications Conference (PerCom)*, 2003.

[22] S. Ponnekanti and A. Fox. Interoperability among independently evolving web services. In *Proceedings of the International Middleware Conference*, pages 331–351, 2004.

[23] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A.Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontologies*, 1(1):77–106, 2005.

[24] M. K. Smith, D. L. McGuinness, and C. Welty. OWL Web Ontology Language guide, W3C recommendation 10 Feb. 2004. http://www.w3.org/TR/owl-guide.

[25] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service descriptions with compositional specification. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 544–552. ACM Press, May 2004.

[26] D. Tosi, G. Denaro, and M. Pezzè. Experimental data on service interchangeability. Technical Report LTA:2008:01, University of Milano-Bicocca, Sep. 2008.

[27] G. Valetto and G. Kaiser. Using process technology to control and coordinate software adaptation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 262–272. IEEE Computer Society, May 2003.

[28] Q. Wang, L. Quan, and F. Ying. Online testing of web-based applications. In *Proceeding of the International Computer Software and Applications Conference (COMPSAC)*, pages 166–169, 2004.

[29] Z. Xing and E. Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007.

APPENDIX

A. API ANALYSIS RESULTS

This appendix reports the results of our analysis of the *del.icio.us* and *OpenSocial* APIs relevant for the presentation in the paper. Refer to [26] for the complete results; below we use the same numbering as in [26] for consistency reasons. Each result item consists of the source of inconsistency (S) identified in the API, the mismatch (M) that can be induced in different implementations, and the corresponding test-and-adapt plans (T and A). Result items are numbered for reference purposes. Test cases and adaptors are indicated informally.

Del.icio.us.

S1: List of tags passed as type *string* M1: Inconsistent separators between tags in the string T1: Different separators to detect the separator used by the service A1: Rewrite the strings accordingly
S2: Tag names passed or returned as type *string* M2: Inconsistent character sets (e.g., lower/upper-case) T2: Different character sets to detect unsupported characters A2: Escape/restore unsupported characters in tag names
S5: XML responses that admit omission of non-mandatory fields M5: Inconsistent sets of field in XML responses T5: Known XML responses to detect fields omitted by the service A5: Add omitted fields using default values
S8: Functionality that handles tag bundles may be considered not essential M8: Operations for handling bundles are not implemented T8: Use of bundles to detect missing implementation A8: Insert a prefix in tag names to identify bundled tags
S13: */tags/rename* can be obtained by combining */posts/all*, */posts/delete* and */posts/add* M13: */tags/rename* is not implemented T13: Invoke */tags/rename* to detect missing implementation A13: Implement */tags/rename* with */posts/all*, */posts/delete* and */posts/add*
S14: */posts/add* does not specify how it handles multiple requests that refer to the same URL M14: Inconsistent implementations of */posts/add* that may (or not) override the previous bookmark of an URL T14: Invoke */posts/add* of an URL many times to detect non-overriding semantics A14: Delete entries of referred URLs before invoking */posts/add*
S15: Underspecified notification of attempts to delete URLs that cannot be found at service side M15: Implementations of */posts/delete* that return different notification messages when URLs cannot be found T15: Invoke */posts/delete* with URLs that do not exist at service side to detect notification messages A15: Log and mask notification messages

OpenSocial.

S2: The API only mandates handling of either *TITLE* or *TITLE_ID* fields of activities, while all other fields (e.g., *BODY* and *URL*) can be potentially ignored by containers M2: Some non-mandatory fields are ignored T2: Use fields of activities to detect ignored fields A2: Use a valid field (e.g., *TITLE*) to handle data of ignored fields, and restore the correct contents of the fields when retrieving the activities
S5: Allowing gadgets to fetch activities may be considered not essential (for containers that provide UI to access the activities) M5: Op. *newFetchActivityRequest* is not implemented T5: Invoke the op. to detect missing implementation A5: Record gadget's activities within the gadget space
S7: Embedding media items into activities may be obtained by using HTML *href* in the activity body M7: Operation *newActivityMediaItem* is not implemented T7: Invoke the op. to detect missing implementation A7: Implement *newActivityMediaItem* through a suitable HTML *href* in the activity body
S11: Parameter *Type* of *newMessage*selects among four working modes (*PUBLIC_MESSAGE*, *PRIVATE_MESSAGE*, *EMAIL*, *NOTIFICATION*) M11: Some modes is not supported T11: Try each mode to detect missing support A11: Tunnel unsupported message types through a compatible supported one; if tunneling may violate privacy, block requests for the unsupported message types;
S14: The person's profile URL, *PROFILE_URL*, embeds parameters (e.g., the user ID) M14: Inconsistent schemas to embed params within URL T14: Retrieve and parse a known profile URL to detect the parameter embedding schema A14: Convert between URL formats accordingly
S17: In a people record, the API mandates handling of field *ID*, while all other fields (e.g., *ABOUT_ME*) can be ignored by the containers M17: Some fields are ignored T17: Fetch fields of a known person to detect ignored fields A17: Filter requests for ignored fields and return a default placeholder value
S19: In fetching people operations, the additional parameter *FILTER* allows for restricting the fetched people to the subset of them that have installed the current gadget (*FILTER* equals to *HAS_APP*) M19: Filtering is not supported T19: Use the parameter *FILTER* to detect missing support A19: Return an empty set of people when *FILTER* is equal to *HAS_APP*
S26: Underspecified error handling in fetch people operations when some person fields in the request are not implemented by the container M26: Implementations that point the error in different ways, e.g., returning a *null* reference or a *BAD_REQ* T26: Fetch a person with known data to detect fields with errors A26: Intercept and homogenize the errors
S29: User display name returned as string M29: Inconsistent implementations that do not return a blank separated name-surname pair T29: Invoke *getDisplayName* to detect the problem A29: Rewrite the string to satisfy client requirement
S30: Size of the user thumbnail is not specified M30: Inconsistent implementations that use non-fitting thumbnails T30: Invoke *newFetchPersonRequest* for a known user with a non-fitting thumbnail to detect the problem A30: Locally resize the thumbnail to satisfy the client requirement

B. INCONSISTENCY CATALOG

This appendix presents the inconsistency catalog as result of our explorative studies. The catalog indicates possible sources of inconsistency in Web2.0 applications (S), mismatches that may derive from such sources of inconsistency (M), and suggests test-and-adapt plans (T and A) accordingly. Some entries list more than one possible adaptation strategy. The choice depends on specific application conditions.

Parameter-related sources of inconsistency are characterized according to whether they apply to input (In) or output (Out) parameters. For each source of inconsistency, we also indicate examples with reference to our del.icio.us ($D.M\langle i\rangle$) and OpenSocial ($OS.M\langle i\rangle$) studies [26].

Inconsistent interpretation of parameter types.

S: [In/Out] Character strings that represent lists of items (D.M1)
M: Inconsistent separators between items in the string
T: Test possible separators to detect the one used by the server
A: Rewrite the string according to the used separator

S: [In] Character strings that represent textual data (D.M2, OS.M1)
M: Inconsistent character sets
T: Test possible char sets to detect unsupported characters
A: Escape/restore unsupported characters

S: [In/Out] Character strings that represent structured data (D.M3)
M: Inconsistent interpretation of the strings in the domain of the structured data
T: Test different values to detect the applied mapping between strings and the domain of structured data
A: Rewrite the strings to guarantee a valid mapping

S: [Out] URLs that embed data values (OS.M14)
M: Inconsistent schemas to embed data values within URLs
T: Retrieve a known URL to detect the applied schema
A: Convert between URL formats after the detected schema

S: [In/Out] Parameters that allow values of multiple types (OS.M15)
M: Inconsistent implementations that refer to either of the types
T: Use the parameter to detect the type referred in the implementation
A: Convert between types as required

S: [In/Out] Integer numbers that indicate positions in a list (OS.M16)
M: Inconsistent implementations that number the first position as either 0 or 1
T: Test positions for a known list to detect the applied reference value
A: Increment/decrement the number to match the reference value

Inconsistent support of service state data.

S: Inconsistent capacity bounds for collections stored at server-side (D.M6, D.M7)
M: Insufficient capacity bounds
T: Test known collections of decreasing size to detect guaranteed bounds
A: Implement priority policies to remove exceeding items from collections to avoid exceeding capacity bounds

S: Non-mandatory fields in data structures that are stored at server-side (OS.M2, OS.M17)
M: Some non-mandatory fields are ignored by the service
T: Test non-mandatory fields to detect ignored fields
A: Use a valid field to handle data of ignored fields
A: Filter the requests for unsupported fields and return default placeholder values

Inconsistent support of service functionality.

S: Set of operations, identifiable from the API naming schema, that implement a side functionality (D.M8, OS.M4-5, OS.M12)
M: Operations that belong to the set are not implemented
T: Invoke the operations to detect missing implementation
A: Specialize the available operations to (partially) achieve the missing behavior
A: Provide local support to (partially) achieve the missing behavior
A: Filter missing operations

S: Operations that can be obtained as specialization of other (sets of) operations (D.M9-13, OS.M7, OS.M23)
M: Operation is not implemented
T: Invoke the operation to detect missing implementation
A: Implement the operation by exploiting the specialization

S: Operations that work in different modes, selected by parameters, to address semantically different sets of data (OS.M3, OS.M11, OS.M18)
M: Some working modes is not supported
T: Invoke the operation for each working mode to detect missing support
A: Specialize the available modes to achieve the missing behavior
A: Filter missing modes

S: Operations that implement usability issues, i.e., to select alternative presentations of the results (OS.M19-20)
M: Some usability issue is not supported
T: Invoke the operation for usability issue to detect missing support
A: Locally tune the presentation of results
A: Return either an unchanged or a default result

S: Operations that notify results asynchronously (OS.M6, OS.M13)
M: Callback functions are never invoked
T: Invoke the operation and check the callback to detect missing support
A: Activate local polling for completion, extract the results, and invoke the client callback
A: Either deliver a default or no notification

Inconsistent handling of service semantics.

S: Underspecified error/success codes of operations, for error/-success cases handled at client-side (OS.M8-10, OS.M26-28)
M: Implementations that return different error/success codes or no code for the same error/success case
T: Invoke the operation for a known error/success case to detect the error/success code
A: Intercept and homogenize the error/success code

S: Operations that delete items from collections (D.M15)
M: Implementations that may or may not notify an error code for items that do not belong to the collection
T: Invoke the operation for an item that does not belong to the collection to detect the possible error code
A: Log and filter the error code

S: Operations that insert/update key-value pairs (D.M14, OS.M24)
M: Overriding/non-overriding semantics in case of subsequent invocations for pairs with the same key
T: Invoke the operation twice to detect the applied semantics
A: Use delete-insert policy to force overriding semantics