Università degli Studi di Milano – Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Laboratorio di Test e Analisi del Software

Rapporto Interno – Technical Report

Experimental data on service interchangeability

D. Tosi, G. Denaro, M. Pezzè

LTA:2008:01

Viale Sarca 336,
20126 Milan, Italy

Experimental data on service interchangeability

Giovanni Denaro and Mauro Pezzè and Davide Tosi
University of Milano Bicocca
{denaro, pezze, tosi}@disco.unimib.it

1 Introduction

The popularity of many web services favors the proliferation of implementations of common APIs that are quickly becoming de facto standard. Compliance with standard APIs facilitates service interchangeability and dynamic discovery of alternative implementations, but does not guarantee correct interoperability under all possible circumstances.

Most problems that derive from inconsistent implementations of service 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 APIs, albeit inconsistent, preserve the main service functionality. Many inconsistencies can be predicted by analyzing the service APIs on the basis of domain expertise and previous experience, identified by executing few simple test cases on the target services, and solved by means of simple adaptors. As all test based approaches and most self-healing solutions, we aim to solve some, but not all problems. Solving all problems would be fantastic, but is not a realistic goal. Being able to automatically solve some problems at run time is a big improvement, since it reduces system malfunction and downtime.

Following this observation, we propose a mechanism that augments service-oriented applications with test cases and adaptors that provide self-adaptive capabilities to dynamically solve incompatibility problems across inconsistent implementations of service APIs. Our mechanism, hereafter *test-and-adapt*, includes design and runtime aspects. At run-time, applications dynamically execute test cases to verify the consistency of the current implementations of service APIs, trigger suitable adaptors when needed, and use the selected adaptors to interact with the current service. 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 associated with an adaptor that is activated depending on the test outcome.

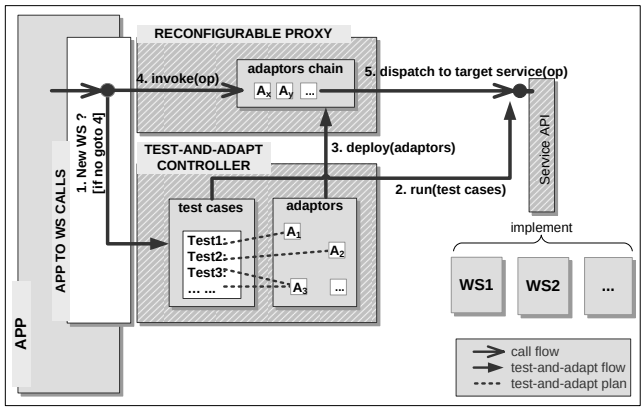


Figure 1. The test-and-adapt approach

Figure 1 illustrates runtime mechanism. The mechanism includes a *Reconfigurable proxy*, which dynamically binds adaptors to service invocations, and a *Test-and-adapt controller*, which manages test-and-adapt plans. The figure illustrates the two service activation flows that handle the cases of service implementations that have and have not been changed after the last access, respectively. If the service implementation has not changed, the application invokes the service through the *Reconfigurable proxy*, which is responsible for activating the adaptors that may have been formerly selected for the current implementation (steps 1, 4 and 5 in Figure 1). If the application identifies a new implementation of 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*, which executes the test cases associated to the API at design-time, deploys suitable adaptors according to the test results (steps 1, 2, and 3 in Figure 1), and then activates the *Reconfigurable proxy* that executed the deployed adaptors. Step 2 requires the execution of test cases. To avoid loose of integrity, we assume either no side effects on the services or the availability of sandbox execution environments, as increasingly provided be service provides, like in the case of

our experiments with social containers.

Devising test-and-adapt plans is the core of the approach. As when designing modern software, software engineers shall identify unexpected executions and design exceptions handlers, when designing service-oriented applications, service engineers shall identify potential inconsistencies that may arise from different implementations of the same API, generate test cases to reveal inconsistent implementations, and design service adaptors. 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.

Our current experience on devising test-and-adapt plans and using the test-and-adapt runtime approach encompasses two explorative case studies that we discuss in the next section. The lessons learned from these experiences generalize to a first sets of inconsistency catalog and validation data for the approach.

In this report, we provide detailed data about problems of interoperability with different implementations of service APIs, referring to our experience with Web2.0 social applications that use the *del.icio.us* and the *OpenSocial* APIs.

2 Experimental data for *del.icio.us*

In the first study, we considered applications for bookmark handling that integrate social bookmarking services offered through the standard *del.icio.us* API, to store and retrieve bookmarks online.

Listing 1 shows, in java-like notation, an excerpt of the *del.icio.us* APIs, 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 (lines 1-7); *tags* operations allow for retrieving and renaming tag keywords that can be associated with the bookmarks for classification purpose (lines 8-9); *tags.bundles* operations provide an auxiliary functionality to define, retrieve and delete semantically coherent sets of tags (lines 10-12).

In our experiments, we executed four applications each integrated with four compatible implementations of the *del.icio.us* API. Table 1 and Table 2 report the results of this study. Table 1 summarizes the number of integration test cases generated with category partition for the operations offered by the *del.icio.us* API (Column *# of TCs*), and subsets of test cases executed for the considered applications (*DEL1.14*, *GAD*, *BtoD*, and *SABROS.us*). Differences of the number of executed test cases depend on the operations used by the different applications. Table 2 reports the number of test cases that have been executed for each application and each implementation of the standard service API, and the number of experienced integration failures.

```
1  posts.get(String filterByTag, String filterByDate
       , String filterByUrl)
2  posts.all(String filterByTag)
3  posts.recent(String filterByTag, int maxItems)
4  posts.dates(String filterByTag)
5  posts.add(String url, String description, String
       tags, String date)
6  posts.update()
7  posts.delete(String url)
8  tags.get()
9  tags.rename(String old, String new)
10 tags.bundles.set(String bundle, String tags)
11 tags.bundles.all()
12 tags.bundles.delete(String bundle)
```

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

Operation	No. TC	Del 1.14	Gad	BtoD	Sabros.us
Authentication	2	2	2	2	2
/posts/get	6	6	-	-	-
/posts/all	4	4	4	-	2
/posts/add	5	5	5	5	-
/posts/delete	2	2	2	-	-
/posts/recent	2	2	-	-	-
/posts/dates	2	2	-	-	-
/posts/update	1	1	-	-	-
/tags/get	2	2	2	-	-
/tags/rename	4	4	-	-	-
/tags/bundles/set	2	2	-	-	-
/tags/bundles/all	1	1	-	-	-
/tags/bundles/delete	2	2	-	-	-
	35	35	15	7	4

All considered applications are open source projects at sourceforge.net. DEL1.14 *is an open source Java API for interacting with the* del.icio.us *social bookmarks service;* GAD = GUI+del.icio.us *is a Python desktop manager of a* del.icio.us *account;* BtoB = BookmarksToDelicious *is a standalone Python client for posting bookmarks to* del.icio.us*;* SABROS.us *is a PHP CMS to put bookmarks online.*

Table 1. Integration test cases for the *del.icio.us* social bookmarking API

The tests revealed 26 integration failures for the implementations of the *del.icio.us* 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 2). Table 3 discusses the assumptions we made when we tested the integration of the four applications in combination with the four API implementations. Failures reported in Table 2 are a superset of the failures that derive from the different assumptions made by the *del.icio.us* API providers (i.e., each assumption violation may generate several failures).

Assumptions	Del.icio.us Implementation choices				Affected App
	del.icio.us	ma.gnolia	faves	link.wieza	
Tags are comma separated	yes	no (tags are space separated)	no (tags are space separated)	yes	DEL1.14, GAD
Tags with uppercase characters are supported	yes	no	no	no	DEL.14, GAD, SABROS.us
/posts/get returns XML elements that contain the *post* field	yes	no	-	yes	DEL1.14
*/tags/boundles/** operations are supported	yes	no	-	no	DEL1.14
/tags/rename operation is supported	yes	no	-	yes	DEL1.14
/posts/add overrides bookmarks of the same URL	yes	yes	no (bookmark is duplicated)	yes	GAD, BtoD, SABROS.us
/posts/delete is silent when deleting non-existing bookmarks	yes	no (returned the error: "*something went wrong*")	-	yes	DEL1.14

Table 3. Analysis of failures

App	#tests	#failures with del.icio.us service			
		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

D: *delicious* [del.icio.us] / M: *magnolia* [ma.gnolia.com]
F: *faves* [faves.com] / L: *link-wieza* [link.wieza.net]
-: not-tested because of incompatible authentication mechanisms

Table 2. Integration failures with different *del.icio.us* implementations

Once we demonstrated that the integration of different applications with different implementations of the *del.icio.us* service API is a real problem, we applied the test-and-adapt approach to the applications listed in Table 2, 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 borrowed from[1]. 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 then 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.

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

Hereafter we report the complete results of our analysis of the *del.icio.us* APIs. 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. Adaptors are characterized as *Full* (if they guarantee the service functionality to full extent), *Partial* (if they provide reasonable albeit degraded behavior), or *Failure masking* (if they only return default results to avoid runtime failures).

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 [Full]

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 [Full]

S3: Date values passed or returned as type *string*
M3: Inconsistent format of date values
T3: Different date formats to detect the format used by the service
A3: Convert date values accordingly [Full]

S4: URL values passed or returned as type *string*
M4: Inconsistent format of URL values
T4: Different URL formats to detect the format used by the service
A4: Convert URL formats accordingly [Full]

S5: XML responses (in */posts/get*) 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 [Partial]

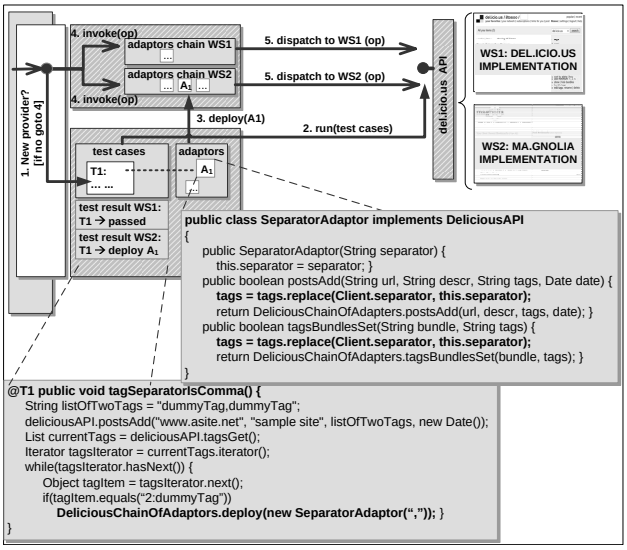
S6: Unspecified number of bookmarks that can be stored by the service
M6: Insufficient capacity bounds of the collection of bookmarks
T6: Decreasing sets of bookmarks to detect applicable capacity bounds
A6: Implement priority policies to cut exceeding bookmarks [Partial]

S7: Unspecified number of tags that can be stored by the service
M7: Insufficient capacity bounds of the collection of tags
T7: Decreasing sets of tags to detect applicable capacity bounds
A7: Implement priority policies to remove exceeding tags [Partial]

S8: */tags/bundles/** functionality 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 [Full]
S9: */posts/dates* can be obtained by specializing */posts/all* M9: */posts/dates* is not implemented T9: Invoke */posts/dates* to detect missing implementation A9: Implement */posts/dates* with */posts/all* [Full]
S10: */posts/recent* can be obtained by specializing */posts/get* M10: */posts/recent* is not implemented T10: Invoke */posts/recent* to detect missing implementation A10: Implement */posts/recent* with */posts/get* [Full]
S11: */posts/update* can be obtained by specializing */posts/get* M11: */posts/update* is not implemented T11: Invoke */posts/update* to detect missing implementation A11: Implement */posts/update* with `/posts/get` [Full]
S12: */tags/get* can be obtained by specializing */posts/all* M12: */tags/get* is not implemented T12: Invoke */tags/get* to detect missing implementation A12: Implement */tags/get* with */posts/all* [Full]
S13: */tags/rename* can be obtained by combining */posts/all*, */posts/delete*, */posts/add* M13: */tags/rename* is not implemented T13: Invoke */tags/rename* to detect missing implementation A13: Implement */tags/rename* combining the three operations [Full]
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* [Full]
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 [Full]

Let us discuss in detail a test-and-adapt plan: For instance, item $S1$ refers to parameters of type *string* that indicate lists of tags (used by several operations (e.g., */posts/add*): These parameters are underspecified because the type *string* does not indicate the separators between tags, which can thus be implemented in several inconsistent ways. Having identified this class of potential mismatches ($M1$), we designed a test-and-adapt plan to reveal and solve them (T1 and A1): 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.

We coded all the 15 test-and-adapt plans as JUnit test cases, which activate the deployment of an adaptor when required. 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 associated with each test-and-adapt plan when connecting to a new provider, and to delegate service calls to the adaptors deployed as a result of executing the test cases. The detailed



Test1 *checks if the service implementation uses commas to separate tags in lists. It calls operation* /posts/add *with a comma separated string as parameter, and then calls operation* /tags/get *to verify the number of tags actually processed by the service. If the check succeeds, it activates an adaptor of type* SeparatorAdaptor *configured for comma separated strings. The adaptors of type* SeparatorAdaptor *filter calls to the operations that use list of tags passed as type* string*, and rewrite the strings according to the specified separator, before forwarding the method calls.*

Figure 2. Instantiated runtime infrastructure

results of the test cases execution, and the set of adaptors automatically deployed by the test-and-adapt infrastructure are shown in Table 3 and in Table 4, respectively. In this first experience, the adaptors automatically solved all the mismatches that cause the failures listed in Table 2.

	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

n.a indicates blocking dependencies on other APIs or libraries.
- stands for no adaptor deployed.

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

Figure 2 illustrates the resulting runtime infrastructure: The figure zooms on a test-and-adapt plan for the tag-separator mismatch, and illustrates that this adaptor is deployed when connecting the service implementation provided by *ma.gnolia* and not for the one provided by *del.icio.us* (alternative paths of the 4-5 flow in the figure).

2.1 Performance data

Here, we report performance data referring to the *del.icio.us* case study. Table 5 summarizes the line of code and the effort required to write the 15 test cases and the adaptors needful by the DEL1.14 application.

	loc	effort
Test suite	675 loc (15 test cases)	10 hours
Adaptors	678 loc (6 adaptors)	4 hours
A1	124 loc	60 mins
A2	130 loc	30 mins
A5	75 loc	30 mins
A8	82 loc	30 mins
A13	90 loc	30 mins
A15	87 loc	30 mins
Infrastructure	420 loc	autogenerated

Table 5. Static Measures for the DEL1.14 application

Table 13 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* APIs.

API	Impl.	No. of 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.

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

Execution of adaptors is less time consuming than test cases. Table 14 reports results measured during our experiments for the application *DEL1.14* in connection with the services from *ma.gnolia*, which requires the indicated adaptors, and *del.icio.us*, which does not require any adaptor. The Table testifies that the worsening of performance caused by the adaptors keeps within acceptable bounds.

Adaptor	If deployed	If not deployed
A1	2634ms	2116ms
A2	2751ms	2230ms
A5	2409ms	897ms
A8	2128ms	1010ms
A13	3011ms	1573ms
A15	1930ms	968ms

Table 7. Mean execution time for running DEL1.14 with or without adaptors

3 Experimental data for *OpenSocial*

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.

Listing 2 shows, in java-like notation, an excerpt of the *OpenSocial* API that serves several purposes: fetching data associated with people and friends (lines 1-3), publishing and accessing user activity information (lines 4-8), sending messages (lines 9-10), and persisting key-value pair data for server-free stateful applications (lines 11-12). Field items (lines 1, 4 and 9) denote the data structures used by the service.

```
1  opensocial.Person.Field {ABOUT_ME: string, AGE:
       number, GENDER: opensocial.Enum.Gender,
       PROFILE_URL: string, ... other fields }
2  opensocial.DataRequest.newFetchPersonRequest(id,
       opt_params)
3  opensocial.DataRequest.newFetchPeopleRequest(
       idSpec, opt_params)
4  opensocial.Activity.Field {BODY: string, TITLE:
       string, TITLE_ID: string, MEDIA_ITEMS: photo,
        video, image, ... other fields }
5  opensocial.newActivity(params)
6  opensocial.newActivityMediaItem(mimeType, url,
       opt_params)
7  opensocial.requestCreateActivity(activity,
       priority, opt_callback)
8  opensocial.DataRequest.newFetchActivitiesRequest(
       idSpec, opt_params)
9  opensocial.Message.Field {BODY: string, TITLE:
       string, TYPE: {EMAIL, NOTIFICATION,
       PRIVATE_MESSAGE, PUBLIC_MESSAGE} }
10 opensocial.requestSendMessage(recipients, message
       , opt_callback)
11 opensocial.DataRequest.
       newUpdatePersonAppDataRequest(id, key, value)
12 opensocial.DataRequest.
       newFetchPersonAppDataRequest(idSpec, keys)
```

Listing 2. Excerpt of the OpenSocial API

In our experiments, 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. Table 8 reports the number of test cases that have been executed for each application and each implementation of the standard service API, and the number of experienced integration failures. The tests revealed 33 integration failures for the implementations of the *OpenSocial* API. Notice that some of the social network gadgets were originally designed for containers that do not provide sandbox environment for testing

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

All considered applications are downloadable from: *http://opensocialdirectory.org*
O: *orkut* [`www.orkut.com`]
H: *hi5* [`www.hi5.com`]
M: *myspace* [`www.myspace.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

Table 8. Integration failures with different *OpenSocial* implementations

(e.g., Ning [*www.ning.com*]), and thus were not included in our experiments.

Once we provided empirical evidence that the integration of different applications with different implementations of the *OpenSocial* service API is a real problem, we applied the test-and-adapt approach to the applications of Table 8, focusing on how they use the *OpenSocial* API.

We used the taxonomy borrowed from[2] as a checklist, we scanned it sequentially, and we applied each entry to all items in the *OpenSocial* API. We identified 30 sources of inconsistency that may originate mismatched service implementations, and we designed corresponding test-and-adapt plans. Hereafter we report the complete results of our analysis of the *OpenSocial* APIs. 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.

S1: Attributes of activities, messages and people of type *string*
M1: Inconsistent character sets (e.g., lower/upper-case)
T1: Different character sets to detect unsupported characters
A1: Escape/restore unsupported characters [Full]

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 the containers
M2: Some 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 [Full]

S3: In operation *newFetchActivityRequest*, parameter *idSpec* selects among four working modes (*VIEWER*, *VIEWER_FRIENDS*, *OWNER*, *OWNER_FRIENDS*)
M3: Some working modes is not supported
T3: Try each modes to detect missing support
A3: Return empty result for unsupported modes [mask the failure]

S4: The functionality that handles activities may be considered not essential
M4: Operations for handling activities are not implemented
T4: Use activities to detect missing implementation
A4: Record gadget's activities within the gadget space [Partial: the gadget will provide access to its own activities only]

S5: Allowing gadgets to fetch activities may be considered not essential (for containers that provide UI to access the activities)
M5: Operation *newFetchActivityRequest* is not implemented
T5: Invoke *newFetchActivityRequest* to detect missing implementation
A5: Record gadget's activities within the gadget space [Partial: the gadget will provide access to its own activities only]

S6: Asynchronous notifications of results of the creation of an activity can be considered as not essential
M6: The callback function, optional parameter of operation *requestCreateActivity*, is not invoked by the container
T6: Invoke *requestCreateActivity* with the callback parameter to detect missing support
A6: Activate local polling for completion of the activity creation, extract the results, and invoke the client callback [Full]

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 *newActivityMediaItem* to detect missing impl.
A7: Implement *newActivityMediaItem* through a suitable HTML *href* in the activity body [Full]

S8: Underspecified error code when creating activities based on malformed activity objects
M8: Inconsistent implementations that return different error codes (operation *requestCreateActivity*)
T8: Create activities with malformed activity objects to detect the error code
A8: Intercept and homogenize the error code [Full]

S9: Underspecified error code when missing the permissions for accessing activities
M9: Implementations that return different error codes
T9: Access activities without permissions to detect the error code
A9: Intercept and homogenize the error code [Full]

S10: Underspecified notification of the success to create an activity
M10: Implementations of *requestCreateActivity* that notify success in different ways (e.g., *null* response item or error message *false*)
T10: Invoke *requestCreateActivity* for a valid activity to detect the notification
A10: Intercept and homogenize the notification [Full]

S11: Parameter *Type* 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 [Partial]; if tunneling may violate privacy, block requests for the unsupported message types [failure masking];

S12: The functionality that handles messages may be considered not essential
M12: Operations for handling messages are not implemented
T12: Use messages to detect missing implementation
A12: Implement message handling within the gadget [Partial: messages will be available in the gadget only]

S13: Asynchronous notifications of success/failure to send a message may be considered not essential
M13: The callback function, optional parameter of operation *requestSendMessage*, is not invoked by the container
T13: Invoke *requestSendMessage* with the callback parameter to detect missing support
A13: Always notify success [Partial]

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

S14: The person's profile URL, *PROFILE_URL*, embeds parameters (e.g., the user ID)
M14: Inconsistent schemas to embed the parameters within the URL
T14: Retrieve and parse a known profile URL to detect the parameter embedding schema
A14: Convert between URL formats accordingly [Full]

S15: The API specifies that Person's *ADDRESSES* field can be implemented as either type *string* or array of items of *Address*
M15: Inconsistent implementations of this field that refer to either of two types
T15: Retrieve the field and check for the type
A15: Convert to a homogeneous type [Full]

S16: When using parameter *FIRST* to indicate the first relevant item in a list of fetched people, the API does not specify the reference value for the first item (0 or 1)
M16: Inconsistent interpretations of the value of the parameter *FIRST*
T16: Specify parameter *FIRST* for a known list of people to detect the applied reference value
A16: Increment/decrement the value of *FIRST* to match the reference value [Full]

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 record to detect ignored fields
A17: Filter requests for ignored fields and return a default placeholder value [failure masking]

S18: In fetching people operations, parameter *idSpec* selects among four working modes (*VIEWER*, *VIEWER_FRIENDS*, *OWNER*, *OWNER_FRIENDS*)
M18: Some working modes is not supported
T18: Try each modes to detect missing support
A18: Return empty set of people for unsupported classes of requests [mask the failure]

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* [failure masking]

S20: In fetching people operations, the additional parameter *SORT_ORDER* allows for ordering the fetched people by either *NAME* or *TOP_FRIENDS*
M20: Either of the orderings is not supported
T20: Try each ordering to detect missing support
A20: Locally sort the list of people after fetching [Full]

S21: In fetching people operations, the pair of additional parameters *FIRST* and *MAX* allow for paginated access to the fetched people (i.e., only *MAX* people starting from the *FIRST*th are returned)
M21: Paginated access is not supported
T21: Try paginated access to a known list of people to detect missing support
A21: Retrieve a list of people up to (*FIRST* + *MAX*) items and locally do pagination as requested [Full]

S22: Unspecified maximum number of items that can be fetched by using the *MAX* parameter
M22: Inconsistent container bounds for the maximum number of items that can be returned in a single invocation
T22: Starting from the client maximum value, use decreasing values of *MAX* to fetch known sets of people, to detect a bound that is guaranteed by the container
A22: Aggregate the result of subsequent invocations to satisfy the value of *MAX* as specified in the client request [Full]

S23: Operation *newFetchPersonRequest* can be obtained by specializing *newFetchPeopleRequest*
M23: Operation *newFetchPersonRequest* is not implemented
T23: Invoke the *newFetchPersonRequest* to detect missing implementation
A23: Implement *newFetchPersonRequest* with *newFetchPeopleRequest* [Full]

S24: Operation *newUpdatePersonAppData* does not specify the handling of subsequent requests for application data with same key
M24: Implementations that override the value of a single item or store multiple items with same key
T24: Invoke *newUpdatePersonAppData* twice with application data that share the same key to detect (non-)overriding semantics
A24: Use delete-insert policy to force overriding semantics [Full]

S25: When using parameter *SORT_ORDER* to sort a fetched list of people, the API does not specify any conventional ordering
M25: Implementations that apply decreasing (or increasing) ordering
T25: Fetch a known list of people using parameter *SORT_ORDER* to detect the applied ordering
A25: Locally reorder the list if needed [Full]

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* error
T26: Fetch a person with known data to detect fields with errors
A26: Intercept and homogenize the errors

S27: Underspecified error code in fetch people operations when people cannot be fetched due to missing permissions
M27: Implementations that return different error codes, e.g., *UNAUTHORIZED* or *FORBIDDEN* error codes
T27: Fetch a person without permissions to detect the error code
A27: Intercept and homogenize the error code [Full]

S28: Underspecified error code in fetch people operations when the maximum number of items (parameter `MAX`) in the request exceeds the limit guaranteed by the container
M28: Implementations that return different error codes
T28: Use increasing values of `MAX` to fetch known sets of people to detect the error code
A28: Intercept and homogenize the error code [Full]

S29: User display name returned as string
M29: Inconsistent impl. that do not return a blank separated name-surname pair
T29: Invoke *getDisplayName* to detect the problem
A29: Rewrite the string to satisfy the 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 client requirement

For instance, item $S30$ refers to users' thumbnails that are retrieved invoking the *newFetchPersonRequest()* operation: The size of the thumbnails is underspecified, thus different containers may return thumbnail images of different sizes. Having identified this class of potential mismatches ($M30$), we designed a test-and-adapt plan to reveal and solve them (T30 and A30): The test cases execute the container requesting a known user with a non-fitting thumbnail to detect the problem and the adaptors locally resize the thumbnail according to the application assumptions, thus assuring the consistency of the interactions.

We coded all the 30 test-and-adapt plans as JUnit test cases, which activate the deployment of an adaptor when required and we proceed as follow: 1) we execute all the test cases of the 30 test-and-adapt plans in combination with the five containers. Table 9 provides data of the execution of the gadget in combination with the five containers. 2) then we modified all eight applications in the experiment to run the test cases when connecting to a new provider, and

to delegate service calls to the adaptors deployed as a result of executing the test cases. Adaptors are implemented as proxy components that mediate calls to the API. Table 11 reports the set of adaptors automatically deployed by the test-and-adapt infrastructure; 3) finally, we present the detailed results of the test cases execution when the eight considered applications are combined with the five containers in Table 10. In this experience, the adaptors automatically solved all the mismatches that cause failures listed in Table 8.

	containers as in Table 8					
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, A26	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	-	-	-	-	-	

n.a indicates blocking dependencies on other APIs or libraries.

- stands for no adaptor deployed.

Table 11. Adaptors deployed for *OpenSocial*

3.1 Performance data

Here, we report performance data referring to the *OpenSocial* case study. Table 12 summarizes the line of code and the effort required to write the 30 test cases and the adaptors needful by the BizX application.

	loc	**effort**
Test suite	1201 loc (30 test cases)	12 hours
Adaptors	XXX loc (5 adaptors)	XXX hours
A2	43 loc	20 mins
A5	XXX loc	25 mins
A14	XXX loc	15 mins
A29	17 loc	10 mins
A30	30 loc	15 mins
Infrastructure	-	-

Table 12. Static Measures for the BizX application

Table 13 reports the mean execution time of the test suites of the test-and-adapt plans defined in our experiments against different implementations of the *OpenSocial* APIs.

API	**Impl.**	**No. of test cases**	**Test run**
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 13. Mean execution time of test suites from test-and-adapt plans

Execution of adaptors is less time consuming than test cases. Table 14 reports results measured during our experiments for the application *BizX* in connection with the two containers *Hi5* (with four deployed adaptors) and *Orkut* (without deployed adaptors). The Table testifies that the worsening of performance caused by the adaptors keeps within acceptable bounds.

Adaptor	**If deployed**	**If not deployed**
A2	XXXms	XXXms
A14	XXXms	XXXms
A29	XXXms	XXXms
A30	XXXms	XXXms

Table 14. Mean execution time for running BizX with or without adaptors

ORKUT	**TEST OUTCOME**
T1	skipped. Dependency problem with T5
T2	failed. Unsupported fields: *BODY_ID*, *STREAM_FAVICON_URL*, *TEMPLATE_PARAMS*
T3	passed. All working modes are supported
T4	passed. The handling of activities is supported
T5	failed. Fetching activities is not allowed. Error=UNAUTHORIZED
T6	passed. Asynchronous notifications are supported
T7	failed. *newActivityMediaItem* is not implemented
T8	passed. As expected, the returned error code is: *BAD_REQ*
T9	passed. As expected, the returned error code is: *UNAUTHORIZED*
T10	passed. As expected, *HAD_ERROR false*
T11	failed. Unsupported working modes: *PUBLIC_MSG*, *PRIVATE_MSG*, *NOTIFICATION*, *EMAIL*
T12	passed. The handling of messages is supported
T13	failed. Asynchronous notifications are not supported. Error code: *BAD_REQ*
T14	passed. As expected, the user ID is embedded as *URL?uid*
T15	passed. The field *ADDRESSES* is implemented as array of items *Address*
T16	passed. FIRST0 points to the first item of the list of fetched people
T17	failed. Ignored fields: *EMAILS*
T18	passed. All the four working modes are supported
T19	passed. *HAS_APP* filtering is supported
T20	failed. Ordering by *NAME* is unsupported
T21	passed. Paginated access is supported
T22	failed. Maximum numbers of items is 20
T23	passed. *newFetchPersonRequest* is implemented
T24	passed. Keys are properly updated
T25	skipped. Dependency with T20: *SORT_ORDER* by name is unsupported
T26	failed. Error code: *UNDEFINED*
T27	failed. Error code: *UNDEFINED*
T28	passed. Error code: *BAD_REQ*
T29	passed. Returned user's name and surname
T30	passed. Returned a user's thumbnail $<$ 70x70 pixels (64x49)

(Orkut)

HI5	**TEST OUTCOME**
T1	failed. Some characters sets are not supported (i.e., special chars)
T2	failed. Unsupported fields: *APP_ID*, *BODY*, *BODY_ID*, *EXTERNAL_ID*, *POSTED_TIME*, *PRIORITY*, *STREAM_SOURCE_URL*, *STREAM_TITLE*, *STREAM_URL*, *TEMPLATE_PARAMS*, *USER_ID*
T3	passed. All working modes are supported
T4	passed. The handling of activities is supported
T5	passed. Fetching activities is allowed
T6	passed. Asynchronous notifications are supported
T7	failed. *newActivityMediaItem* is not implemented
T8	passed. As expected, the returned error code is: *BAD_REQ*
T9	failed. Returned error code: *UNDEFINED*
T10	passed. As expected, *HAD_ERROR false*
T11	failed. Unsupported working modes: *PUBLIC_MSG*, *PRIVATE_MSG*, *NOTIFICATION*, *EMAIL*
T12	passed. The handling of messages is supported
T13	failed. Asynchronous notifications are not supported. Error code: *BAD_REQ*
T14	failed. The user ID is embedded as *URL?userid*
T15	skipped. Dependency with T17 (*ADDRESSES* is unsupported)
T16	passed. FIRST0 points to the first item of the list of fetched people
T17	failed. Ignored fields: *ABOUT_ME*, *ADDRESSES*, *EMAILS*
T18	passed. All the four working modes are supported
T19	failed. *HAS_APP* filtering is unsupported
T20	failed. Ordering by *NAME* and *TOP_FRIENDS* is unsupported
T21	passed. Paginated access is supported
T22	failed. Maximum numbers of items is 20
T23	passed. *newFetchPersonRequest* is implemented
T24	passed. Keys are properly updated
T25	skipped. Dependency with T20: *SORT_ORDER* is unsupported
T26	failed. Error code: *UNDEFINED*
T27	failed. Error code: *UNDEFINED*
T28	passed. Error code: *BAD_REQ*
T29	failed. Returned user's name only
T30	failed. Returned a user's thumbnail $>$ 70x70 pixels (100x100)

(Hi5)

MYSPACE	TEST OUTCOME
T1	skipped. Dependency problem with T5
T2	failed. Only the mandatory *TITLE* and *TITLE_ID* fields are supported
T3	skipped. Dependency with T5
T4	passed. The handling of activities is supported
T5	failed. Fetching activities is not allowed. Error=UNDEFINED
T6	failed. Asynchronous notifications are unsupported
T7	failed. *newActivityMediaItem* is not implemented
T8	skipped. Dependency with T6
T9	skipped. Dependency with T5
T10	skipped. Dependency with T6
T11	failed. Unsupported working modes: *PUBLIC_MSG*, *PRIVATE_MSG*, *NOTIFICATION*, *EMAIL*
T12	failed. The handling of messages is unsupported
T13	failed. Asynchronous notifications are not supported. Error code: *no answer*
T14	failed. The user ID is embedded as *URL?1234567*
T15	skipped. Dependency with T17 (*ADDRESSES* is unsupported)
T16	failed. FIRST1 points to the first item of the list of fetched people
T17	failed. Ignored fields: *ADDRESSES*, *EMAILS*
T18	skipped. Unexpected exception: *"TypeError: personId undefined"*
T19	failed. *HAS_APP* filtering is unsupported
T20	failed. Ordering by *NAME* and *TOPS_FRIENDS* are unsupported
T21	failed. Paginated access is unsupported
T22	failed. Maximum numbers of items is 20
T23	passed. *newFetchPersonRequest* is implemented
T24	failed. Keys are duplicated instead of updated
T25	skipped. Dependency with T20: *SORT_ORDER* is unsupported
T26	failed. Error code: *no answer*
T27	failed. Error code: *no answer*
T28	passed. Error code: *no answer*
T29	failed. Returned user's name only
T30	skipped. The user's thumbnail is not retrievable

(MySpace)

FACEBOOK	TEST OUTCOME
T1	passed. All the character sets are supported
T2	failed. Unsupported fields: Unsupported fields: *APP_ID*, *BODY_ID*, *EXTERNAL_ID*, *MEDIA_ITEMS*, *POSTED_TIME*, *PRIORITY*, *STREAM_FAVICON_URL*, *STREAM_SOURCE_URL*, *STREAM_TITLE*, *STREAM_URL*, *TEMPLATE_PARAMS*, *URL*, *USER_ID*
T3	passed. All working modes are supported
T4	passed. The handling of activities is supported
T5	passed. Fetching activities is allowed
T6	passed. Asynchronous notifications are supported
T7	failed. *newActivityMediaItem* is not implemented
T8	passed. As expected, the returned error code is: *BAD_REQ*
T9	failed. Returned error code: *Problem initializing QueryRequest*
T10	failed. Returned *MSG=[object Object]*
T11	failed. Unsupported working modes: *PUBLIC_MSG*, *PRIVATE_MSG*, *NOTIFICATION*, *EMAIL*
T12	failed. The handling of messages is unsupported
T13	failed. Asynchronous notifications are not supported. Error code: *no answer*
T14	failed. The user ID is embedded as *URL?id*
T15	skipped. Dependency with T17 (*ADDRESSES* is unsupported)
T16	passed. FIRST0 points to the first item of the list of fetched people
T17	failed. Ignored fields: *ABOUT_ME*, *ADDRESSES*, *EMAILS*, *GENDER*
T18	passed. All the four working modes are supported
T19	passed. *HAS_APP* filtering is supported
T20	failed. Ordering by *NAME* and *TOP_FRIENDS* are unsupported
T21	passed. Paginated access is supported
T22	failed. Maximum numbers of items is 20
T23	passed. *newFetchPersonRequest* is implemented
T24	passed. Keys are properly updated
T25	skipped. Dependency with T20: *SORT_ORDER* is unsupported
T26	failed. Error code: *UNDEFINED*
T27	failed. Error code: *UNDEFINED*
T28	passed. Error code: *BAD_REQ*
T29	passed. Returned user's name and surname
T30	passed. Returned a user's thumbnail $<$ 70x70 pixels (50x37)

(Facebook)

IMEEM v0.1	TEST OUTCOME
T1	skipped. Dependency problem with T5
T2	failed. Only the mandatory *TITLE* and *TITLE_ID* fields are supported
T3	skipped. Dependency problem with T5
T4	passed. The handling of activities is supported
T5	failed. Fetching activities is not allowed
T6	failed. Asynchronous notifications are unsupported
T7	failed. *newActivityMediaItem* is not implemented
T8	skipped. Dependency with T6
T9	failed. Returned error code: *no answer*
T10	skipped. Dependency with T6
T11	failed. Unsupported working modes: *PUBLIC_MSG*, *PRIVATE_MSG*, *NOTIFICATION*, *EMAIL*
T12	failed. The handling of messages is unsupported
T13	failed. Asynchronous notifications are not supported. Error code: *no answer*
T14	failed. The user ID is embedded as */people/id*
T15	skipped. Unexpected exception: "*TypeError: address.getField is not a function*"
T16	failed. FIRST is unsupported
T17	failed. Ignored fields: *EMAILS*
T18	passed. All the four working modes are supported
T19	failed. *HAS_APP* filtering is unsupported
T20	failed. Ordering by *NAME* and *TOP_FRIENDS* are unsupported
T21	failed. Paginated access is unsupported
T22	failed. Maximum numbers of items is 20
T23	passed. *newFetchPersonRequest* is implemented
T24	passed. Keys are properly updated
T25	skipped. Dependency with T20: *SORT_ORDER* by name is unsupported
T26	failed. Error code: *no answer*
T27	failed. Error code: *no answer*
T28	failed. Error code: *no answer*
T29	passed. Returned user's name and surname
T30	failed. Returned a user's thumbnail $>$ 70x70 pixels (100x100)

(Imeem v0.1)

IMEEM v0.2	TEST OUTCOME
T1	passed. All the character sets are supported
T2	failed. Only the mandatory *TITLE* and *TITLE_ID* fields are supported
T3	passed. All working modes are supported
T4	passed. The handling of activities is supported
T5	passed. Fetching activities is allowed
T6	failed. Asynchronous notifications are unsupported
T7	failed. *newActivityMediaItem* is not implemented
T8	skipped. Dependency with T6
T9	failed. Returned error code: *no answer*
T10	skipped. Dependency with T6
T11	failed. Unsupported working modes: *PUBLIC_MSG*, *PRIVATE_MSG*, *NOTIFICATION*, *EMAIL*
T12	failed. The handling of messages is unsupported
T13	failed. Asynchronous notifications are not supported. Error code: *no answer*
T14	failed. The user ID is embedded as */people/id*
T15	skipped. Unexpected exception: "*TypeError: address.getField is not a function*"
T16	failed. FIRST is unsupported
T17	failed. Ignored fields: *EMAILS*
T18	passed. All the four working modes are supported
T19	failed. *HAS_APP* filtering is unsupported
T20	failed. Ordering by *NAME* and *TOP_FRIENDS* are unsupported
T21	failed. Paginated access is unsupported
T22	failed. Maximum numbers of items is 20
T23	passed. *newFetchPersonRequest* is implemented
T24	passed. Keys are properly updated
T25	skipped. Dependency with T20: *SORT_ORDER* by name is unsupported
T26	failed. Error code: *no answer*
T27	failed. Error code: *no answer*
T28	failed. Error code: *no answer*
T29	passed. Returned user's name and surname
T30	failed. Returned a user's thumbnail $>$ 70x70 pixels (100x100)

(Imeem v0.2)

Table 9. Test case results for the different containers

BizX	orkut	hi5	myspace	facebook	imeem v0.1	imeem v0.2
T2:	ok	ko (updates of the vCard are missing of a part stored in the *BODY* filed of the activity)	na	ok	ko (see Hi5)	ko (see Hi5)
T5:	ok	ok	na	ok	ko (activities are not retrieved)	ok
T14:	ok	ko (TypeError: sender result has no properties)	na	ko (see Hi5)	ko (see Hi5)	ko (see Hi5)
T29:	ok	ko (null pointer exception)	na	ok	ok	ok
T30:	ok	ko (the graphical layout of the vCard is compromised)	na	ok	ko (see Hi5)	ko (see Hi5)

(a)

BuboMe	orkut	hi5	myspace	facebook	imeem v0.1	imeem v0.2
T2:	ko (the favicon of the gadget is not visualized)	ok	na	ko (see Orkut)	ko (see Orkut)	ko (see Orkut)
T5:	ok	ok	na	ok	ko (activities are not retrieved)	ok
T7:	ko (activities are created without the *any_50.gif* image. Exception: *error 500*)	ko (see Orkut)	na	ko (see Orkut)	ko (see Orkut)	ko (see Orkut)
T17:	ko (the retrieve of the user's email failed)	ko (see Orkut)	na	ko (the retrieve of the user's email and gender failed)	ko (see Orkut)	ko (see Orkut)
T26:	ko (Error code: *undefined* when retrieving the user's email)	ko (see Orkut)	na	ko (Error code: *undefined* when retrieving the user's email and gender)	ko (timeout when retrieving the user's email)	ko (see Imeem v0.1)

(b)

BuddyPoke	orkut	hi5	myspace	facebook	imeem v0.1	imeem v0.2
T2:	ok	ko (only the activity title is rendered)	na	na	na	na
T11:	ko (sending a notification msg to a poked friend failed)	ko (see Orkut)	na	na	na	na
T19:	ok	ko (Retrieving the list of friends using this gadget always returns: *"none of your friends have BuddyPoke installed"*)	na	na	na	na

(c)

Emote	orkut	hi5	myspace	facebook	imeem v0.1	imeem v0.2
T5:	ok	ok	na	na	ko (activities are not retrieved)	na

(d)

RateMyFriends	orkut	hi5	myspace	facebook	imeem v0.1	imeem v0.2
T5:	ok	ok	ok	ok	ko (activities are not retrieved)	ok
T14:	ok	ok	ko (the link to the user's profile is malformed)	ko (see myspace)	ko (see myspace)	ko (see myspace)

(e)

Unype	orkut	hi5	myspace	facebook	imeem v0.1	imeem v0.2
T5:	ok	ok	ok	ok	ko (activities are not retrieved)	ok
T29:	ok	ko (only the user's name is visualized)	ko (see hi5)	ok	ok	ok

(f)

Table 10. Analysis of failures

4 Experimental data for *Gadget*

Here, we provide a third case study (*Gadgets* API) following our previous experiences with the two case studies *del.icio.us* and *OpenSocial*. This evaluation aims to answer two questions: Q1. How big is the problem of integrating third-party services that refer to the same API? Q2. How effective and general is the Inconsistency Catalog (that we derived from the two previous analysis) in practice when applied to a different applicative domain?

To answer to these questions, we replicated our previous experiments with this new API, and we computed the number of integration failures the test-and-adapt approach is able to prevent. *Gadgets* is an API for building mini-applications that run on multiple sites (such as *iGoogle* or *NetVibes*). Gadgets are simple HTML and JavaScript mini-applications served in iFrames that can be embedded in webpages and other applications. The *Gadgets* API is currently supported by several gadgets servers.

Listing 3 shows, in java-like notation, an excerpt of the *Gadgets* API that serves several purposes: managing users' preferences (lines 1-12), managing the retrieve of remote contents (lines 14-19), providing utilities (lines 21-30).

```
1  prefs.getArray(String key)
2  prefs.getBool(String key)
3  prefs.getCountry()
4  prefs.getFloat(String key)
5  prefs.getInt(String key)
6  prefs.getLang()
7  prefs.getModuleId()
8  prefs.getMsg(String key)
9  prefs.getMsgFormatted(String key, String subst)
10 prefs.getString(String key)
11 prefs.set(String key, Object val)
12 prefs.setArray(String key, Array val)
13
14 AuthorizationType = {AUTENTICATED, NONE, SIGNED}
15 ContentType = {DOM, FEED, JSON, TEXT}
16 MethodType = {DELETE, GET, HEAD, POST, PUT}
17 RequestParam = {AUTHORIZATION, METHOD, ...}
18 io.makeRequest(String url, RequestParam params)
19 io.getProxyUrl(String url)
20
21 io.encodeValues(Object fields)
22 json.parse(String text)
23 json.stringify(Object v)
24 util.escapeString(String str)
25 util.getFeatureParameters(String feature)
26 util.getUrlParameters()
27 util.hasFeature(String feature)
28 util.registerOnLoadHandler(Function callback)
29 util.unescapeString(String str)
```

Listing 3. Excerpt of the Gadgets API

We proceeded as in the previous experiments. We used the inconsistency catalog that we derived from our previous experiences, we scanned it sequentially, and we applied each entry to all items in the *Gadgets* API. We identified a total of 10 sources of inconsistencies that may originate mismatching service implementations, and we designed corresponding test-and-adapt plans. Hereafter, we report the complete results of our analysis of the *Gadgets* API. 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.

S1: Attributes of gadgets.* operations of type *string*
M1: Inconsistent character sets (e.g., lower/upper-case)
T1: Different character sets to detect unsupported characters
A1: Escape/restore unsupported characters [full]
S2: Proxied version of the URL passed as type string (op. *getProxyUrl*)
M2: Inconsistent structure of returned values
T2: Invoke *getProxyUrl* to detect the structure of the returned value
A2: Intercept and homogeneously restructure the returned value [full]
S3: The output URLs of op *io.encodeValues* embed parameters
M3: Inconsistent schema to embed the parameters within the URL
T3: Retrieve and parse a known profile URL to detect the parameter embedding schema
A3: Convert between URL format accordingly [full]
S4: The API specifies that the op *prefs.getModuleId* may return a *string*, a *number* or both
M4: Inconsistent interpretation of the type used for the returned value
T4: Invoke *prefs.getModuleId* to detect the type of the returned value
A4: Intercept and homogenize the returned value [full]
S5: In *gadgets.io MethodType*, the API mandates handling of type *GET*, while all other types can be not supported by the servers
M5: Some types are not supported
T5: Make a request with each method type to detect unsupported types
A5: Filter *io* requests for unsupported types and return a warning message [failure masking]
S6: The functionality that manipulate *JSON*, objects may be considered not essential
M6: Operations *gadgets.json.** are not implemented
T6: Use *JSON* objects to detect missing implementations
A6: Implement the manipulation of *JSON* objects client-side [partial]
S7: *RequestParameters* selects among different working modes (*AUTHORIZATION, CONTENT_TYPE,...*)
M7: Some modes is not supported
T7: Try each mode to detect missing support
A7: Tunnel unsupported modes through a compatible supported one [partial]; if tunneling is not possible, block request for the unsupported modes [failure masking]
S8: Asynchronous notifications of *util.makeClosure*, *util.registerOn*, and *io.makeRequest* may be considered not essential
M8: The callback function is not invoked by the gadget server
T8: Invoke an operation with the callback to detect missing support
A8: Always notify success [partial]
S9: Underspecified error code when fetching an unformatted msg for an invalid key (op *prefs.getMsg*)
M9: Inconsistent implementations that return different error codes
T9: Invoke *prefs.getMsg* with an invalid key to detect the error code
A9: Intercept and homogenize the error code [full]
S10: *prefs.set* and *prefs.setArray* do not specify the handling of subsequent requests for preference data with same key
M10: Implementations that override the preference or store multiple preferences with same key
T10: Invoke the operations twice with preferences that share the same key to detect (non)-overriding semantics
A10: Locally store used keys and use versioning mechanisms for subsequent requests with same key [full]

App	#tests	#failures with Gadget service			
		O	I	H	N
Babylon	3 (P1, P9, P10)	0	0	0	0
BizX V1.0	2 (**P7**, P8)	1	1	0	1
BizX V2.0	4 (P3, P5, P7, P8)	0	0	0	0
BuboMe	3 (P5, P7, P8)	0	0	0	0
BuddyPoke V2.018	5 (**P2**, P3, P5, P7, P8)	0	na	2	2
Calc	2 (P1, P10)	0	0	0	0
GoogleMaps	2 (P1, P10)	0	0	0	0
ToDo	2 (P1, P10)	0	0	0	0
Weather	3 (P1, P6, P10)	0	0	0	0

O: *Orkut* [orkut.com] / H: *Hyves* [hyves.nl]
I: *iGoogle* [igoogle.com] / N: *Netlog* [en.netlog.com]
All considered widgets are downlodable from http://opensocialdirectory.org and http://www.labpixies.com/
na: not-tested, the installation of the widget fails
in bold is represented the failed plan

Table 16. Integration failures with different *Gadgets* implementations

We selected a set of widgets available on the web and we analyzed their characteristics focusing on the Gadgets operations they use. We selected the 4 gadget servers: *Orkut*, *iGoogle*, *Hyves*, *Netlog* from the list available at [http://code.google.com/apis/gadgets/].

We coded all the 10 test-and-adapt plans as JUnit test cases, which activate the deployment of an adaptor when required and we proceed as follow: 1) we execute all the test cases of the 10 test-and-adapt plans in combination with the four gadget servers. Table 15 provides data of the execution of the gadget in combination with the four servers. 2) then we modified all eight widgets in the experiment to run the test cases when connecting to a new provider, and to delegate service calls to the adaptors deployed as a result of executing the test cases. Adaptors are implemented as proxy components that mediate calls to the API. Table 16 summarizes the test cases that have been executed for each widgets with each gadget server, and the number of experienced integration failures; It reports also the set of adaptors automatically deployed by the test-and-adapt infrastructure (see the bold plan P). 3) finally, we present the detailed results of the test cases execution and the experienced failures when the *BizX* and *BuddyPoke* widgets are combined with the four gadget servers in Table 17. In this experience, the adaptors automatically solved all the mismatches that cause failures listed in Table 16.

4.1 Performance data

Here, we report performance data referring to the *Gadgets* case study. Table 18 summarizes the line of code and the effort required to write the 10 test cases and the adaptors needful by the *BizX* and *BuddyPoke* widgets. Table 19 reports the mean execution time of the test suites of the test-and-adapt plans defined in our experiments against different implementations of the *Gadgets* APIs.

	loc	effort
Test suite	455 loc (10 test cases)	3 hours
Adaptors	30 loc (2 adaptors)	0.3 hours
A2	13 loc	10 mins
A7	18 loc	10 mins
Adaptor	**If deployed**	**If not deployed**
A2	250ms	220ms
A7	324ms	310ms

Table 18. Static/Dynamic Measures for the BizX/BuddyPoke widgets

```
/* A2: override the getProxyUrl() method. The new
* implementation is transparent wrt the invocation,
* while the returned value is restructured by the
* overridden method */
function getProxyUrl(url) {
  var proxiedUrlAdpt;
  var refresh="?refresh=3600";
  var gadget="&gadget=your_gadget_url";
  var domain=opensocial.getEnvironment().getDomain();
  var container="&container="+domain;
  var url="&url="+url;
  var proxiedUrl=gadgets.io.getProxyUrl(url);
  var strEnd=proxiedUrl.indexOf("/proxy")+5;
  var substr=refresh+gadget+container+url;
  proxiedUrlAdpt=proxiedUrl.substr(0,strEnd)+substr;
  return proxiedUrlAdpt;
}
```

API	Impl.	No. of test cases	Test run
Gadgets	Orkut	10	4 sec.
	iGoogle	10	4 sec.
	Hyves	10	5 sec.
	Netlog	10	5 sec.

Table 19. Mean execution time of test suites from test-and-adapt plans

Execution of adaptors is less time consuming than test cases. Table 18 reports results measured during our experiments for the *BizX* and *BuddyPoke* widgets in connection with the two servers *Orkut* and *Hyves*. The Table testifies that the worsening of performance caused by the adaptors keeps within acceptable bounds.

ORKUT	TEST OUTCOME
T1	passed. Uppercase and lowercase chars are supported: 123UPPERlower
T2	passed. The proxied url is in the form: ?refresh= &gadget= &container= &url=
T3	passed. The schema value1&value2 is satisfied: name=test&cat=1234567
T4	passed. The type of the ret val is the number: 0
T5	passed. PUT, POST, HEAD, and DELETE are supported.
T6	passed. JSON functionalities are supported.
T7	failed. FEED working mode unsupported
T8	passed. Asynchronous notification is supported.
T9	passed. Returned an empty string
T10	passed. Key is overridden. Key1=12

(Orkut)

iGOOGLE	TEST OUTCOME
T1	passed. Uppercase and lowercase chars are supported: 123UPPERlower
T2	failed. The proxied url is in the form refresh= &container= &gadget=
T3	passed. The schema value1&value2 is satisfied: name=test&cat=1234567
T4	passed. The type of the ret val is the number: 0
T5	passed. PUT, POST, HEAD, and DELETE are supported.
T6	passed. JSON functionalities are supported.
T7	failed. FEED working mode unsupported
T8	passed. Asynchronous notification is supported.
T9	passed. Returned an empty string
T10	passed. Key is overridden. Key1=12

(iGoogle)

HYVES	TEST OUTCOME
T1	passed. Uppercase and lowercase chars are supported: 123UPPERlower
T2	failed. The proxied url is in the form ?refresh= &url=
T3	passed. The schema value1&value2 is satisfied: name=test&cat=1234567
T4	passed. The type of the ret val is the number: 0
T5	passed. PUT, POST, HEAD, and DELETE are supported.
T6	passed. JSON functionalities are supported.
T7	passed. All working modes are supported
T8	passed. Asynchronous notification is supported.
T9	passed. Returned an empty string
T10	passed. Key is overridden. Key1=12

(Hyves)

NETLOG	TEST OUTCOME
T1	passed. Uppercase and lowercase chars are supported: 123UPPERlower
T2	failed. The proxied url is in the form ?refresh= &url=
T3	passed. The schema value1&value2 is satisfied: name=test&cat=1234567
T4	passed. The type of the ret val is the number: 0
T5	passed. PUT, POST, HEAD, and DELETE are supported.
T6	passed. JSON functionalities are supported.
T7	failed. FEED working mode unsupported
T8	passed. Asynchronous notification is supported.
T9	passed. Returned an empty string
T10	passed. Key is overridden. Key1=12

(Netlog)

Table 15. Test case results for the different gadgets servers

BizX V1.0	Orkut	iGoogle	Hyves	Netlog
T7:	com.sun.syndication.io.ParsingFeedException: content is not allowed *	(see Orkut)	ok	(see Orkut)

(a)

BuddyPoke	Orkut	iGoogle	Hyves	Netlog
T2a:	ok	na	createMoodActivity(): activity body is wrongly filled **	(see Hyves)
T2b:	ok	na	createPokeActivity(): activity body is wrongly filled ***	(see Hyves)

(b)

∗ when starting the widget, the exception com.sun.syndication.io.ParsingFeedException is thrown in background. This happens because the FEED working mode is unsupported by Orkut and iGoogle

∗∗ whenever a user updates the mood of his/her 3D avatar, an activity is created. BuddyPoke in combination with Hyves and Netlog fails in creating the activity body because the body lacks of the container name. This happens because the createMoodActivity() uses the gadget operation: getProxyUrl() to retrieve the fields useful for creating the activity. In Hyves and Netlog the getProxyUrl() does not return the &container value.

∗ ∗ ∗ whenever a user pokes a buddy, an activity is created. BuddyPoke in combination with Hyves and Netlog fails in creating the activity body because the body lacks of the container name. See ∗∗ for more details.

Table 17. Analysis of failures