Hypermedia-Driven RESTful Service Composition

Rosa Alarcon[1], Erik Wilde[2], and Jesus Bellido[1]

[1] Computer Science Department
Pontificia Universidad Catolica de Chile
ralarcon@ing.puc.cl, jbellido@uc.cl
[2] School of Information
UC Berkeley
dret@berkeley.edu

Abstract. *Representational State Transfer* (REST) services are gaining momentum as a lightweight approach for the provision of services on the Web. Unlike WSDL-based services, in REST the set of operations is reduced, standardized, with well known semantics, and changes the resource's state. Few attempts have been proposed to support composition models for REST, they are mainly operation-centric and fail to acknowledge the hypermedia nature of REST, that is, clients must inspect the served resource state and choose the link to follow from there. We explore RESTful service composition as it is driven by the hypermedia net that is dynamically created while a client interacts with a server resulting in a light-weight approach. We based our proposal on a hypermedia-centric REST service description, the *Resource Linking Language (ReLL)* and Petri Nets as a mechanism for describing the machine-client navigation.

1 Introduction

SOA services provide an endpoint that exposes a set of *operations* on entities that are out of the reach of clients. Operations are described in a standard WSDL document; semantics are not explicit and are usually specified in additional documents so that client designers understand the scope, effects, pre-conditions and assumptions made by service designers and program the clients accordingly. Clients interact with servers following the description (they are *tightly coupled*), if it changes clients fail, clients cannot be notified about changes and failure semantics and its recovery are ad-hoc. Client-server interaction state is kept by the server (*stateful*), which negatively impacts service scalability and increases the complexity of coarse grained operations.

The REST architectural style has been characterized as a restricted subset of SOA [1]. Unlike WSDL operations, in REST the central elements are the *resources*, which are abstract entities identified by URIs that can be manipulated through a *uniform interface*, that is, a reduced set of standard operations whose semantics are well known in advance and are defined by standard transport protocols such as HTTP. Resource's *state* is transfered to/from the clients as a consequence of executing the standard operations. The state is portrayed to the clients by means of *representations*, which are documents serialized according to specific media types (e.g. XML), and contain hyperlinks to related resources, and *controls* that allow clients to perform operations and change resources' state (e.g. `<link=URI rel="service.POST">`, `<form ...>`, etc.). There is no guarantee that the operations, the resources or even the network remain available or unchanged, however, there is a uniform failure interface (i.e. standard protocol error codes) with well known semantics that allow clients to recover accordingly.

A REST service is not an endpoint but a web of interconnected resources, with an underlying hypermedia model that determines not only the relationships among resources but also the possible net of resource state transitions. REST clients discover and decide which links/controls to follow/execute at run-time. This constraint is known as HATEOAS (Hypermedia As The Engine Of Application State). Hence, it is possible to provide a single or a small set of URIs as *entry-points* to the whole service web or a subset of it. REST services consider humans as its principal consumer and they are expected to drive resource discovery and state transition by understanding the representation content. The lack of a machine-readable description forces REST service providers to describe their APIs in natural language which makes difficult to properly design machine-clients RESTful services are becoming a promising technology and recently is being studied as an infrastructure layer for supporting service composition and business processes. Recent proposals, however, heavily rely on the *operation*-based model neglecting the hypermedia characteristics of REST. In this paper, we explore the impact of the hypermedia (HATEOAS) property for supporting machine-clients that implement RESTful service composition. Unlike current approaches, we based our strategy on a service description called ReLL [2], that is also based on the hypermedia property. The approach allow us to implement a machine client that is able to perform dynamic discovery of REST resources.

This paper is organized as follows, Section 2 present related work in both REST composition and REST description; Section 3 briefly presents ReLL, the Resource Linking Language for REST service description; Section 4 introduces an example to illustrate a composition model and language based on Petri Nets; and Section 5 presents conclusions and future work.

2 Related work

In [3] composition requirements specific to REST services are identified, such as, *dynamic late binding*, that is the resource's URI to be consumed is known only in run-time; the composition technique must support the REST *uniform interface*; *dynamic typing*, methods may require parameters with types known only on run-time; *content type-negotiation*, the representations' media type for the component services as well as the composed service can be negotiated; and clients should be able to *inspect the state* of the composition.

JOpera [3] satisfies such requirements, and it is one of the most mature platforms for supporting REST services composition. JOpera provides a visual language for defining a *control flow* and a *data flow transfer* graphs, as well as an execution engine for the resulting workflow. Nodes in the control flow graph represent *tasks* that are dynamically bound to *adapters* such as local UNIX programs, services invocation, etc., and "glue" adapters that perform local computations (e.g. XPath queries, XSLT transformations, etc.). Tasks and adapters have input and output parameters, for instance, HTTP adapter parameters are: `Method`, `URI`, `Body` and headers (`headin`). Adapter invocation order is regulated by *control tasks* that define conditional synchronization points, conditional loops, and forks. The data flow graph makes explicit the data flow between tasks and the resulting composition is written as a BPEL extension for REST [4].

Similarly, Bite [5] proposes a BPEL-inspired workflow composition language describing both control and data flow. Bite partially supports an HTTP-based uniform interface, dynamic typing, state inspection (both `GET` and `POST`). Regarding dynamic late binding, Bite can mint URIs for resources created, but can not inspect representation content and selectively retrieve the URIs served by the service. In both JOpera and Bite, the composition workflow is seen as a unique composed resource. In [6], it is possible to inspect the state of the workflow instance (i.e. the composed resource or *case*) and the tasks that compose it. State transition is modeled as links following a *URI template* that can change dynamically. Tasks progression is driven by humans and no automatic support is provided to retrieve and follow the links embedded in the representation.

Decker [7], presents a formal model for REST process enactment based on Petri Nets, PNML (Petri Net Markup Language), and an execution engine. They partially support dynamic late binding by minting URIs for created resources but do not support the HATEOAS constraint, neither complex guard conditions such as authentication, nor content negotiation (only XML as media type).

On the other hand, it is argued that a service description introduces a contract between clients and servers and hence some degree of coupling. In practice, service developers provide informal documents for its REST APIs describing the resource types, the set of *entry points* (static URIs) and URI patterns for the resources, authentication mechanisms, protocols, operations and media types, and even representation content samples, so that machine-clients can be designed accordingly. Documents may end up outdated, inaccurate or unclear, forcing developers to engage in trial-error phases to accommodate such changes.

A few languages have been proposed to create RESTful services description. For instance, the Web Application Description Language (WADL) [8] describes RESTful services as *resources* identified by URI patterns, media types and the schemas of the expected *request* and *response*. Representations support parameters that can contain links to another resources. WADL does not support link discovery or link generation for new resources, the resulting model is operation-centric and introduces additional complexity with unclear benefits for both human and machine-clients. Other proposals such as WRDL (Web Resource Description Language), and WDL (Web Description Language) [4], introduce less complex descriptions to model resources, but they focus also on the operations allowed for each particular resource and their input and output parameters and do not support the HATEOAS property.

3 Resource Linking Language (ReLL)

In [2] we introduced ReLL, the *Resource Linking Language*, which describes REST services with emphasis on the hypermedia characteristics of the model. Figure 1 presents the metamodel that comprises the main constraints of REST and is the basis for ReLL. A ReLL XML Schema has been also produced [2] and is used to write ReLL XML descriptions. A snippet, written in ReLL, describing a `requestToken` resource is shown in Figure 2. A REST `service` exposes a set of one or more `resources` each with a unique identifier (`xml:id`), names and descriptions (human-readable labels) and optionally constraints for the expected resources, such as a `uri` pattern for the expected (`match`) resource URI (so that a machine client can identify a URI change). A resource may have multiple `representations`, which are the serialization of the resource in some syntax or media `type`. Representations can define schemas for validation of input data.

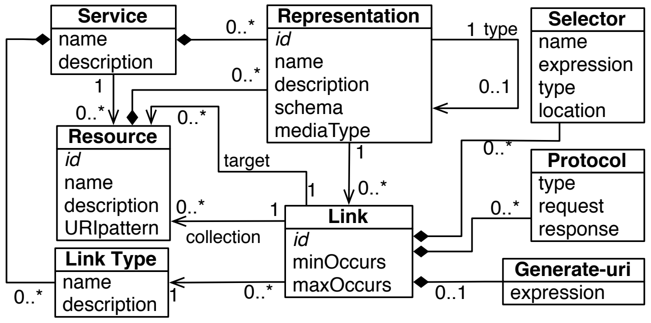

Fig. 1. ReLL Metamodel

Each representation can contain any number of `links` that can be retrieved through `selectors` written in a language (`selector type`) that suits the representation media type. For instance, XPath (*XML Path Language*) expressions for XML-based representations since they allow structured selections within XML document trees. Selectors have a `name` and refer also to a `location` in the representation (e.g. the content or the metadata such as HTTP headers). Links relate resource's representations with other *resource type* (instead of a resource URI) as indicated by the `target`, in order to avoid coupling with the resources' naming scheme. A link can also specify a *protocol* and it is possible to mint or `generate a URI` by executing an expression (e.g. a concatenation written in XPath). ReLL allows also to annotate resources and links with `types`, so that application domain semantics can be explicitly declared without requiring changes in the existent resources.

A machine-client can use the description to automatically and selectively retrieve the underlying web of resources. For instance, in [9], RESTler, a Web crawler uses ReLL descriptions to retrieve resources from a Web site, and REST

```xml
<resource xml:id="requestToken">
    <uri match="https://api.linkedin.com/uas/oauth/requestToken" type="regex"/>
    <representation xml:id="requestToken-text" type="iana:text/plain">
        <name>oauth_token request parameters</name>
        <link xml:id="authorizeRq" type="request" target="authorize" minOccurs="0" maxOccurs="1">
            <selector name="oauthUri" select="oauth_request_auth_url=[a-zA-Z0-9\-\%\.]+" type="regex"/>
            <selector name="oauthToken" select="oauth_token=[a-zA-Z0-9\-]+" type="regex"/>
            <generate-uri xpath="concat($oauth_url,'?',$oauth_token)"/>
            <protocol type="http">
                <request method="get"/>
                <response media="iana:text/plain"/>
            </protocol>
        </link>
    </representation>
</resource>
<resource xml:id="authorize">
     <uri match="https://api.linkedin.com/uas/oauth/authorize\?oauth_token=[a-zA-Z0-9\-]*" type="regex"/>
 </resource>
```

Fig. 2. LinkedIn ReLL snippet

APIs (Twitter, and Google Calendar). Since domain semantics are explicitly supported, it is possible to transform the retrieved resources to its *semantic* counterpart through XSLT, and integrate the services at the semantic level. For instance, in [10] resources from four REST services (Flickr, Twitter, a Web site and a User mapping) are transformed to RDF, and integrated so that SPARQL queries can consider the resulting graph.

4 REST service composition

4.1 OAuth, LinkedIn and Facebook, a composition example

We illustrate these ideas by composing two REST services, LinkedIn and Facebook. A machine-client will retrieve the `SocialNetwork` (contacts and friends) from both sources and will provide an XML representation of the merged data. Entities (top rectangles in each thread) represent REST resources. Both services require user authentication based on the *OAuth* protocol, though with different versions, 1.0 for LinkedIn (Figure 3a) and 2.0 for Facebook (Figure 3b). This difference imply variations in the client-server conversation as well as the passed values. Client-server interaction occurs between the machine-client and the resources implementing the steps of the protocol (e.g. generate a *requestToken*, *authorize*, and generate an *accessToken*). Part of the interaction occurs out of band, as a separate conversation between the service provider and the user. Some parameters are sent to/from the server in the message body, others in the Headers, some messages must be signed, and some parameters are used as inputs for the next client-server interaction step.

The HATEOAS constraint is extensively used in Web content (i.e. it contains embed hyperlinks and controls) and is in great part responsible for the Web popularity since it allows users to dynamically discover material they are interesting in, but unfortunately it is not considered yet by REST API designers, that is,

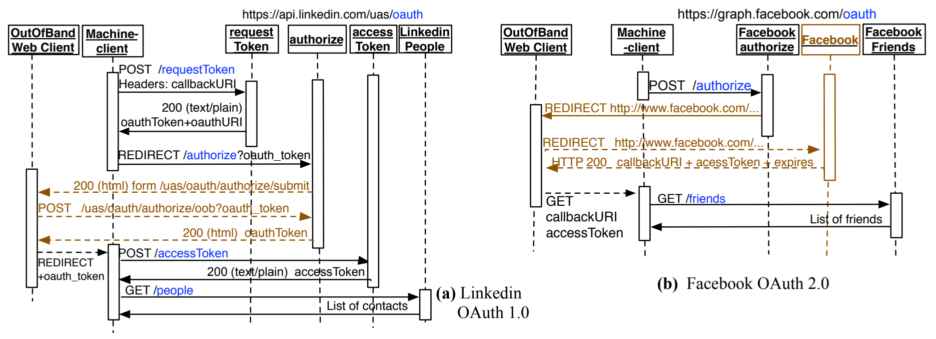

Fig. 3. OAuth sequence diagram for LinkedIn (a) and Facebook (b)

representations typically do not include hyperlinks but data fields that serve to generate or mint the URIs to follow. This practice introduces a strong coupling between clients and servers and forces machine-clients to mint their own URIs

4.2 A Petri Net model

We are interested in the development of machine-clients that enable service composition, so that B2B or mashups involving REST services could be easily developed. The HATEOAS constraint becomes fundamental provided that machine-clients can understand the semantics of the links and controls served in the representations. A ReLL document describes in a declarative way a partial (or whole) view of the web of resources, its domain semantics and the mechanisms required to navigate across such web, and hence, provides machine-clients access to a basic semantic model at a protocol and application level.

In [7], a formalization for *service nets* implementing RESTful processes is introduced. A service net is a colored Petri Net represented by a tuple $S = (\mathcal{P}, \mathcal{T}, \mathcal{F}, \mathcal{F}_{read}, \mathcal{T}_S, \mathcal{T}_R, init, g, uri)$, where $\mathcal{P}$ and $\mathcal{T}$ are disjoint sets of *places* and *transitions*. Places represent states that contain tokens with multiple attributes, and transitions represent activities that can be guarded; transitions are fired when all the tokens in the corresponding input place arrive. Places and transitions are connected through arcs. $\mathcal{F}$ represents the set of flows, and $\mathcal{F}_{read}$ the set of read arcs (GET). $\mathcal{T}_S, \mathcal{T}_R$ correspond to the disjoint sets of *send* and *receive* transitions (also called communication transitions), *init* is the initial marking, that is, the function that initially assigns multiple tokens (with values serialized as XML documents) to places, g is a function that assigns guard conditions to transitions and conditions are combinations of XML serialized input documents, and *uri* is a function that assigns URIs to tuples of communication transitions $(\mathcal{T}_S, \mathcal{T}_R)$ and combinations of XML serialized input documents, this feature allows the model to generate URIs. According to Decker, REST service composition is defined as the merge of *send* and *receive* transitions offered by senders and receivers that belong to separate service nets. One token is assigned to an input place and removed to an output place when a transition fires. In the case of read arcs, tokens are not removed from input places and there is no functional dependency between tokens.

Dynamic late binding in this model, is implemented by receive transitions ($\mathcal{T}_R$) that take input tokens to generate new URIs called here dynamic ports. The rules for uploading information into tokens and for generating new URIs are not presented, but this is not a trivial task since it may require parsing information, encrypting, digital signatures, store information into headers, etc.

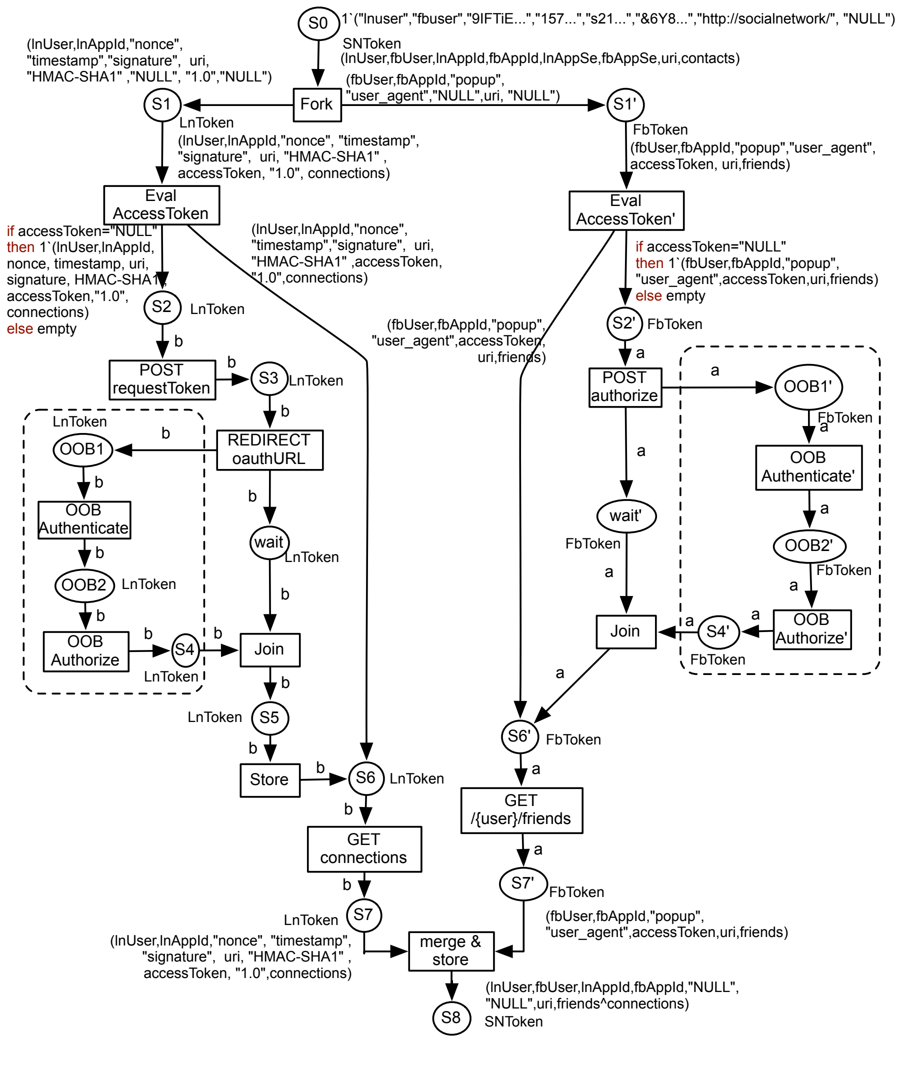
S0
1`("lnuser","fbuser","9lFTiE...","157...","s21...","&6Y8...","http://socialnetwork/", "NULL")
SNToken
(lnUser,fbUser,lnAppId,fbAppId,lnAppSe,fbAppSe,uri,contacts)
Fork
S1
S1'
LnToken
FbToken
Eval AccessToken
Eval AccessToken'
if accessToken="NULL" then 1`(lnUser,lnAppId, nonce, timestamp, uri, signature, HMAC-SHA1, accessToken,"1.0", connections) else empty
if accessToken="NULL" then 1`(fbUser,fbAppId,"popup", "user_agent",accessToken,uri,friends) else empty
S2
S2'
POST requestToken
S3
REDIRECT oauthURL
OOB1
OOB Authenticate
OOB2
OOB Authorize
S4
wait
Join
S5
Store
S6
GET connections
S7
POST authorize
OOB1'
OOB Authenticate'
OOB2'
OOB Authorize'
S4'
wait'
Join
S6'
GET /{user}/friends
S7'
merge & store
S8
(lnUser,fbUser,lnAppId,fbAppId,"NULL", "NULL",uri,friends^connections)
SNToken

Fig. 4. A PetriNet for the composed resource: `socialNetwork`

This feature is fundamental in real life scenarios, as is shown by a popular language used for service composition in the industry, namely BPEL, which supports data transformation by means of XPath and XQuery expressions; or JOpera which uses a library of extensible adapters to satisfy the same requirement. The other aspect of *dynamic late binding*, that is, to inspect representation's content and determine from there the URI of a send transition (i.e. the HATEOAS constraint) is not discussed.

The colored Petri Net shown in Figure 4 corresponds to the example of Section 4.1. It implements a `socialNetwork` resource that composes resources from LinkedIn and Facebook. The Petri Net was modeled and simulated using a CPN tool. Places (circles) represent states of the composed resource; tokens (set of attributes) are shown near places (e.g. LnToken); places receive a determined type of token and transitions provide the mapping between tokens with different attributes. Places and transitions within dotted rectangles and labeled as `OOBn` (Out Of Band) and refer to interaction controlled by the REST OAuth service, the machine client has not control nor access to such resources but only waits until the flow is redirected through a receiving transition. Transitions may have guard conditions (`if`), perform internal tasks (e.g. `EvalAccessToken`, `Store`, etc), pass values or perform HTTP requests to external REST services (`POST`, `GET`, `Redirect`, etc.), corresponding to sender transactions ($\mathcal{T}_S$) in Decker's model. An input place (e.g. *S0*) can receive HTTP messages such as `POST`.

4.3 ReLL based dynamic late binding

The Petri Net is also modeled in XML as a simplified version of PNML (Figure 5). Concerns are separated in three layers: the REST service *resources* layer (which may introduce new resources when composing services (e.g. `http://ing.puc.cl/socialNetwork`); the ReLL service descriptions layer (which supports *dynamic late binding*); and the Petri Net model layer, that drives the client-server interaction.

Consider Figure 4, once the `socialNetwork` resource receives a `POST`, a token is generated and a `Fork` operation is performed separating the original token in two, one for each source service (LinkedIn and Facebook). Figure 5 details the token received at *s1*, it declares key-value pairs that were received in the POST request header, must be generated on run-time (e.g. timestamp), or are part of the machine-client internal state (e.g. cokies). Transition `Eval Access Token` evaluates whether the attribute `accessToken` (in this case, for LinkedIn) is set, and a guarded condition determines the next place. If the attribute is not set, tokens are moved to *s2* and the OAuth authentication process begins by triggering the `POST requestToken` transition.

This transition sends a message to LinkedIn's `requestToken` resource, the machine-client expects a response of type `requestToken`, `text/plain` (see Figure 2). The response is the input for the *s3* place and a new token (`LnToken`) will be minted by executing the regular expressions defined for `oauthUri` and `oauthToken` variables in the ReLL description. The URI for the `REDIRECT` transition will be minted by following the ReLL instructions for the `authorizeRq`

```
<place id="s1" resource="SocialNetwork">
   <token name="LnToken">
       <attr location="header" name="lnUser"/>
       <attr location="cookie" name="oauth_consumer_key" />
       <attr name="oauth_nonce" type="Random"/>
       <attr name="oauth_timestamp" type="Timestamp"/>
       <attr name="oauth_signature_method">HMAC-SHA1</attr>
       <attr location="cookie" name="oauth_callback" />
       <attr location="cookie" name="accessToken" />
       <attr name="oauth_version">1.0</attr>
       <attr location="cookie" name="connections" />
   </token>
</place>
...
<place id="s3" resource="requestToken">
   <token name="LnToken">
        <attr location="body" name="oauthUri"/>
        <attr location="body" name="oauthToken"/>
   </token>
</place>
<transition id="REDIRECT" link="authorize_rq">
   <token name="LnToken">
       <attr location="body" name="oauthUri"/>
       <attr location="body" name="oauthVerifier"/>
   </token>
</transition>
<place id="wait" resource="authorize"/>
<place id="s5" resource="callback">
...
```

Fig. 5. Petri Net XML snippet

link. To follow the minted URI, the machine-client must sent the `LnToken`, when performing a `get` operation on the `http` protocol, and expect a `text/plain` response; at most one link may be generated and the retrieved resource will correspond to the `authorize` type (Figure 2). The `authorize` resource represents the beginning of an out of band conversation between the service provider (e.g. LinkedIn) and a Web client controled by the provider. The machine-client `waits` until a response corresponding to the `callback` resource is received at place *s5*. An internal operation will store part of the received response (the `accessToken`), a new URI will be minted for the next transition (`GET connections`) and the token as detailed initialed for the `s1` place will be sent with updated values (cookies). The response of both LinkedIn and Facebook services will be merged and a new resource will be generated (i.e. `socialNetwork/1`).

A CPN tool was used to model and simulate the service composition. An XML description for the Petri was generated, and both Petri and ReLL descriptions were parsed, uploaded and executed by the machine-client. Internally, the machine-client includes libraries for parsing and executing regular and XPath expressions, as well as an HTTP client to perform requests to REST resources and expose itself as a resource: the `SocialNetwork`. Transitions are fired according to an enablement component that handles the markings and tokens passed among places.

5 Conclusions

Service descriptions (e.g. ReLL) introduce coupling between clients and servers, however even loosely coupled services need a shared set of assumptions, and a more formal way of describing those assumptions will help service providers and consumers in service documentation and consumption, as evidenced by the currently existent REST APIs documentation. ReLL allows clients to detect whether some assumptions have changed (e.g. more links than expected are served, the URIs have changed, the protocol have changed etc.), so that a proper action can be taken (e.g. extend the ReLL description and describe the new interface or even versioning the description).

By separating concerns in different layers (description, composition) instead of merging them in a full fledged service composition engine, the ReLL descriptions are reusable in scenarios other than the described in this paper (e.g. web crawling, and semantic integration), we expect that the Petri net descriptions could be also reused in complex scenarios where composition actually include composed resources. On the downside, ReLL descriptions and the composition itself are static. We plan to explore recent work on planning that make possible to introduce dynamic decisions and generate Petri Nets accordingly, as well as to explore the impact of explicit semantics in a dynamic composition scenario.

References

1. H. Overdick, "Towards resource-oriented BPEL," in *WEWST*, ser. CEUR Workshop Proceedings, C. Pautasso and T. Gschwind, Eds., vol. 313. CEUR-WS.org, 2007.
2. R. Alarcón and E. Wilde, "Linking data from restful services," in *Third Workshop on Linked Data on the Web*, Raleigh, North Carolina, April 2010.
3. C. Pautasso, "Composing restful services with jopera," in *International Conference on Software Composition 2009*, ser. Lecture Notes in Computer Science, A. Bergel and J. Fabry, Eds., vol. 5634. Zürich, Switzerland: Springer-Verlag, July 2009, pp. 142–159.
4. ——, "Restful web service composition with bpel for rest," *Data Knowl. Eng*, vol. 68, no. 9, pp. 851–866, 2009.
5. F. Rosenberg, F. Curbera, M. J. Duftler, and R. Khalaf, "Composing RESTful services and collaborative workflows: A lightweight approach," *IEEE Internet Computing*, vol. 12, no. 5, pp. 24–31, 2008.
6. X. Xu, L. Zhu, Y. Liu, and M. Staples, "Resource-oriented architecture for business processes," in *APSEC*. IEEE, 2008, pp. 395–402.
7. G. Decker, A. Lüders, H. Overdick, K. Schlichting, and M. Weske, "Restful petri net execution," in *WS-FM*, 2008, pp. 73–87.
8. M. Hadley, "Web application description language," World Wide Web Consortium, Member Submission SUBM-wadl-20090831, August 2009.
9. R. Alarcón and E. Wilde, "Restler: Crawling restful services," in *19th International World Wide Web Conference*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. Raleigh, North Carolina: ACM Press, April 2010, pp. 1051–1052.
10. R. Alarcon and E. Wilde, "From restful services to rdf: Connecting the web and the semantic web," School of Information, UC Berkeley, Berkeley, California, Tech. Rep. 2010-041, June 2010.