Towards Resource-Oriented BPEL

Hagen Overdick

Research School
Hasso Plattner Institute for IT Systems Engineering
at the University of Potsdam
D-14482 Potsdam, Germany
hagen.overdick@hpi.uni-potsdam.de

Abstract. Service orientation is the de-facto architectural style, today. But, what actually is a service and how should service boundaries be chosen? Resource orientation, once seen as a "light-weight" approach to Web services, is reshaping itself as a modeling strategy to service orientation. Along comes the realization that resources are in-fact complex state machines. Currently, there is no accepted standard for modeling the internal state of resources. In this paper, BPEL is proposed as a modeling language for resources and necessary extensions to BPEL are outlined.

1 Introduction

There has been a lot of discussion about service-oriented architectures (SOA) [1], lately. A service is a mechanism to enable access to one or more capabilities. The eventual consumers of the service may not be known to the service provider and *may demonstrate uses of the service beyond the scope originally conceived by the provider* [2]. If a provider may not know the actual use of a service, what makes a service a service? What minimum level of functionality must a service provide to be called a service? Furthermore, according to recent studies [3], about two-thirds of all services deployed today are data-centric. Is a memory cell already a service? Resource orientation [4] solves this dilemma by making every entity explicit, not just services. Such explicit entity is called a resource. If one can find a noun for an entity, it qualifies as a potential resource. All properties of services still hold true for resources, i.e. they have an independent life- cycle, a globally unique reference, and their interaction style is stateless message exchange.

Resource orientation is the dominant architectural style on the Internet, as it is the scientific foundation of the World Wide Web [5]. Resources have a globally shared request message classification system, confusingly called *uniform interface*. The idea is, that even without semantic understanding of the messages exchanged, the classification provides additional benefits to the overall architecture. However, up to now, the World Wide Web favors an informal, ad-hoc description of complex resource behaviors. Roy Fielding coined the term "hypertext as the engine of application state" [4], upgrading this ad-hoc fashion from bug to feature; quite a successful feature indeed measured by the success of the World Wide Web itself.

Enterprises and research on the other hand are very much interested in the description of complex behaviors. Out of this need, Web Services [6] were created as an additional layer on top of resource orientation, including the Web Service Definition Language (WSDL) [7] to describe service interfaces and the Business Process Execution Language (BPEL) [8] for behavioral descriptions. Recently, resource orientation is rediscovered as a viable subset of service orientation [9]. This also raises the question, if complex resource behavior can be expressed formally. With the introduction of the Web Application Definition Language (WADL) [10], a candidate for the description of interfaces is given. This paper outlines how BPEL can be adapted to describing process aspects of resources.

The remainder of this paper is structured as follows: In chapter 2, an introduction to resource orientation is given. In chapter 3, an example of a complex resource behavior is shown to illustrate the requirements of a resource-oriented process language. In chapter 4, BPEL is introduced as a viable candidate for such a language and the necessary extensions are outlined. Chapter 5 discusses related work and chapter 6 concludes this paper with a summary and outlook.

2 Resource Orientation

Resource orientation is a subset of service orientation. As such, it can be regarded as a modeling strategy for services. Instead of a few "gateway" services, with carefully crafted custom interfaces, all entities of the modelled system expose a uniform interface. To illustrate this, let us look at an example of such uniform interface. The Hypertext Transfer Protocol (HTTP) [11] defines its uniform interface for requests as:

GET : Messages labeled as GET have an empty service request and are guaranteed to have no substancial effect within the receiver of such request, i.e. they are *safe* to call. GET responses are expected to be a description of the current state of the targeted resource. These attributes allow GET to act as a universal reflection mechanism, it can be issued without any prior knowledge of the resource. Also, as GET does not alter the state of the targeted resource, the response can be cached. This has great benefits to a distributed architecture and both aspects can be seized without prior semantic knowledge of the targeted resources. In the physical world, GET request can be correlated to looking.

PUT : Messages labeled as PUT do cause an effect in the targeted resource, but do so in an *idempotent* fashion. An idempotent interaction is defined as replayable, i.e. the effect of N messages is the same as that of 1. In a distributed system, where transactions may not be available, this is a great help for error recovery as idempotent messages can be delivered at least once without any effort, just retry until acknowledged. Again, thanks to the uniform interface, the assumption of idempotency can be made without any prior semantic knowledge

of the resource involved. In the physical world, this correlates to physical interaction, although replaying the exact same "message" is only a theoretical mind exercise.

DELETE : Messages labeled as DELETE do cause an effect in the targeted resource, where that effect is expected to be a termination. Just as PUT, DELETE is defined as *idempotent*. However, as with all messages, the interpretation is solely the responsibility of the receiver, i.e. a DELETE has to be regarded as "please terminate". In the physical world, this correlates to sending a notice of cancelation.

POST : All other types of messages are labeled as POST, i.e. they cause an effect in the receiver and they are not safe to replay. This is a catch all mechanism for all messages that can not be described by the prior verbs. Without a uniform interface, all messages would be treated like this, loosing context free reflection, caching and replayability.

The uniform interface tries to lower the barrier of entry to a client and it also includes a characterization of response messages. Thus, interaction with a resource can start purely on the basis of semantic understanding of the uniform interface. If one obtains a resource identifier, the uniform interface provides a minimum level of shared semantics to start with. However, this set of semantics is not restricted. A uniform interface simply enforces that any label (or verb as HTTP calles them) may be applied to *all* resources. To increase the likelihood of understanding, both client and resource perform content type negotiation on each request. Content type negotiation honors the fact that there are many ways of encoding information.

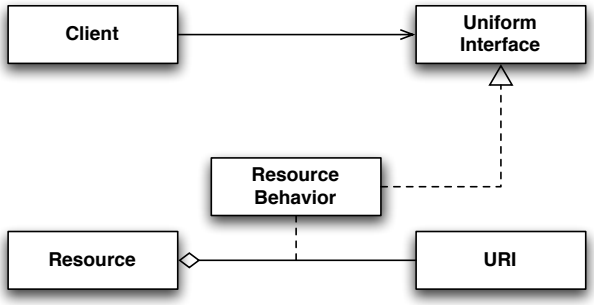

Fig. 1. Exposing behavior in resource-oriented architectures

Conceptually, a resource is beyond the communication system, i.e. a client can only communicate with it via the uniform interface it addresses by a globally

unique identifier, e.g. a URI [12]. The relationship between a resource and a URI may change over time. Yet, resource orientation today spends very little effort on describing the *underlying process* defining the relationship between a resource and its URIs. While a URI is bound to a resource, it exposes a certain resource behavior. In figure 1 the relationship between these concepts described is shown.

3 Example of a complex resource behavior

As an illustration, let us now introduce a complex resource most people should be familiar with: an online ordering process. In figure 2, a very simple version is illustrated. A shopping cart is created by the user by adding an initial item. Adding items can be repeated as many times as the user likes. If the user simply stops interacting with the shopping cart, it may time out or the user decided to check out by choosing a payment method. At this point the user is presented with the content of the shopping cart, the chosen payment method, and the total bill to confirm before actually committing the order.

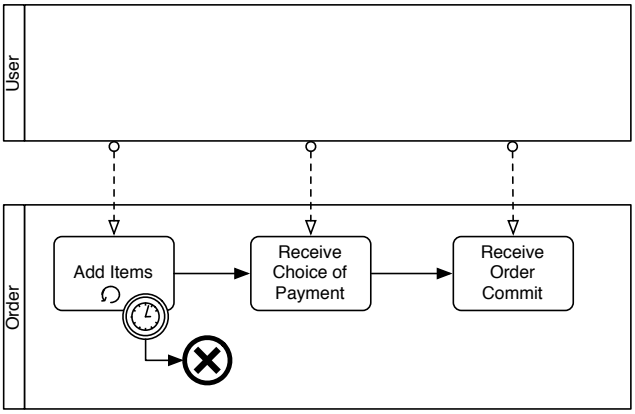

Fig. 2. Shopping cart as a complex resource

The first step towards a good resource-oriented design is to identify the relevant resources. Is a shopping cart actually a resource on its own, or just a state of an order? By modeling the later—the shopping cart to be just a state of an order resource—we can uniquely identify the order in all stages, e.g. shopping cart, check-out, assembly, in-delivery, and post-delivery. The user is given a single URI, something to bookmark in a browser. Clicking on such a bookmark will issue a GET request. A GET request in a resource-oriented view is nothing else than introspecting the current state of a resource. In the true spirit of *hypermedia as the engine of application state* the returned representation of the current resource should include all relevant links and possible interactions.

By doing so, the client is never forced to understand the process as such, being able to browse and post is the only requirement to participate as a client. This simplicity is the true strength of a resource-oriented design and the foundation of the World Wide Web's success story. At the same time, this motivates the resistance against formal descriptions of interfaces and processes as practiced in a Web services environment. While resource orientation does not conflict with formal interface descriptions and in fact would benefit from it, any attempt to introduce such formalism to resource orientation must honor the fact that resource orientation can and will work without such formalism.

Nevertheless, it should have become apparent that resources are indeed complex state machine and that such state machines can be expressed as processes, matching the business concepts used to motivate the system in the first place. We already identified a shopping cart to be just a state or dependent sub-resource of an order. However, this opens the question of how to choose resource boundaries. Is the order a resource in itself or is it a sub-resource of the store? In [13] a very pragmatic answer is given: Breaking down an application into as many resources as possible benefits scalability and flexibility, but at the same time the resource is the scope of serializability, i.e. there may not be transactions across resource boundaries. I.e. the order is not dependent on the store (at least in a transactional view), but the order items probably are dependent on the order, as an order item may only be changed as long as the order has not been committed.

Before we outline a process notation in the next chapter, let us summarize our findings: A resource may consist of several complex states, each able to expose a set of URIs. Each of these URIs expose a certain behavior of the resource. Interaction with any of the resource's URIs is classified into *safe* (one interaction has the same effect as zero interactions), *idempotent* (one interaction has the same effect as n interactions), or unrestricted, i.e. such interaction is able to produce an uncontrollable side-effect and/or change the internal state of the resource. Also, a resource must be able to extract URIs from representations received via interaction and be able to then interact with the extracted URIs, as this is a fundamental aspect of resource orientation.

4 Resource-oriented BPEL

In this section, BPEL is introduced and extensions for modeling complex resources are outlined.

4.1 BPEL

BPEL is arguably the de facto standard for specifying processes in a Web services environment. BPEL provides *structured activities* that allow the description of the control flow between the *interaction activities*. BPEL does not support explicit data flow, but rather relies on shared variables referenced and accessed by interaction activities and manipulation activities. The control flow between activities can be structured either block-based by nesting structured activities

like $< sequence >$ and $< flow >$, or graph-based by defining directed edges (called $< links >$) between activities inside $< flow >$ activities. Both styles can be used as the same time, making BPEL a hybrid language.

Beyond control flow and data manipulation, BPEL also supports the notion of scopes and allow for compensation handlers and fault handlers to be defined for specific scopes. Hence, scopes represent units of works with compensation-based recovery semantics. Fault handlers define how to proceed when faults occur, compensation handlers define how to compensate already completed activities, as processes not transactional and consequently must be rolled back explicitly. Further more, scopes allow for event handlers which can be regarded as repeatable, attached sub-processes [14] triggered by events.

4.2 BPEL without Web Services

The wide-spread acceptance and the sophistication of the control flow constructs, make BPEL a strong candidate when trying to formally express the process governing the relationship between a resource and its URIs. Both the interaction activities and the grouping mechanism that allows modeling complex message exchanges depend on WSDL. However, in [15] BPEL light is introduced, a WSDL-less version of BPEL. While BPEL light itself still is not a good match for resource orientation, a clear path on how to remove the dependency on WSDL from BPEL and adding new interaction models in a compatible way is shown clearly. In essence, the elements $< receive >$, $< reply >$, $< invoke >$, $< onMessage >$ within a $< pick >$, as well as $< onEvent >$ within an $< eventHandler >$ need to be replaced by constructs not relying on WSDL.

4.3 Using BPEL to model resource states

BPEL does not have an explicit state modeling, but an implicit via the $< scope >$ construct. Generally speaking, a POST message or an event may cause a state transition. However, while in a state, as many GET, PUT, and/or DELETE messages may arrive, as they are safe and/or idempotent.

As shown in figure 3, BPEL provides the concept of event handlers to model GET, PUT, and DELETE interaction as attached, repeatable subprocesses. Enforcing *safe* and *idempotent* characteristics of those interactions is beyond the scope of this paper. However, a straight forward solution may be disallowing write access to any variable during a GET interaction to ensure safeness. PUT and DELETE can be enforced idempotent by disallowing write access to any variable read, i.e. overwriting a variable is ok, computing a new value based on the old one is not. Such interaction may be executed several times and in parallel, while POST interaction or events move the BPEL process into a new scope.

4.4 Resource interaction in BPEL

Web services try to abstract from the communication protocol, providing support for a wide range of interaction models, such as asynchronous or one-way

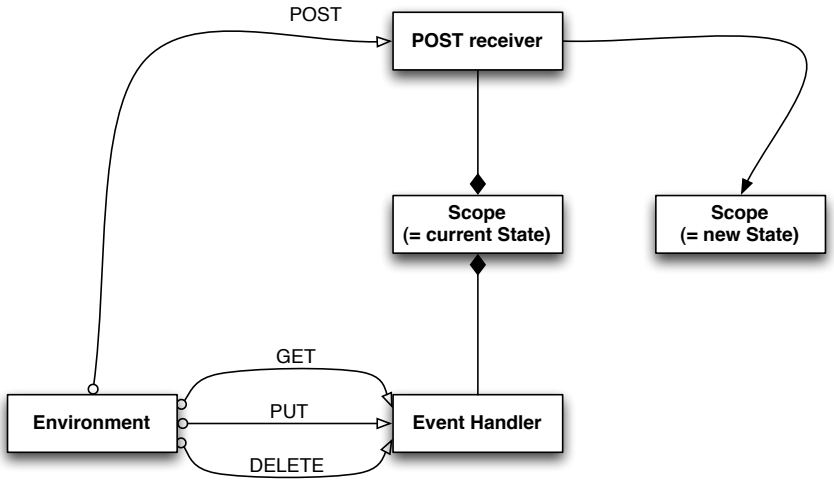

Fig. 3. Using BPEL to model resource states

interaction. Resource orientation on the other hand puts much effort into the core protocol as the lowest level of shared semantics. The dominate resource-oriented protocol is HTTP. Consequently there is no point in abstracting away from it when modeling interaction in BPEL. In fact many proponents of resource orientation have major concerns with any attempt to hide the protocol layer behind an abstraction.

All interaction in HTTP is based on synchronous request-response. Asynchronous communication is supported by identifying either the asynchronous sender or receiver by an explicit URI and sending it along in the initial request. I.e. at the protocol level, there will be a synchronous request and then an independent synchronous response push or pull. This design makes the interaction much simpler, but requires a simple mechanism to construct URIs. There is currently one attempt to standardized URI templating [16] applicable to creation, matching, and selection of URIs. Within WADL, URI templates are already used for matching and selection of URIs. To a resource itself, creation of URIs must be available, too. Coming back to our example process, upon receiving a shopping item, it must be added to the shopping cart, in turn generating one or more URIs for the newly created item.

The easiest way to provide such functionality is to offer an XPath function. Figure 4 shows how the regular $< assign >$ construct is used to create a new URI using such XPath function. URIs themselves do not need a special construct and can be kept in normal variables.

With URIs introduced to BPEL, let us look at URI interaction again, as shown in figure 5. Any URI interaction is synchronous and the tuple of *request* and *response* is grouped into a *message exchange*. Both request and response contain a header and a body, where the header includes the content type of the

```
<assign>
  <copy>
    <from>rbpel:generate-uri("./item/{itemNumber}")</from>
    <to variable="newItemURI" />
  </copy>
</assign>
```

Fig. 4. URI creation by XPath-method

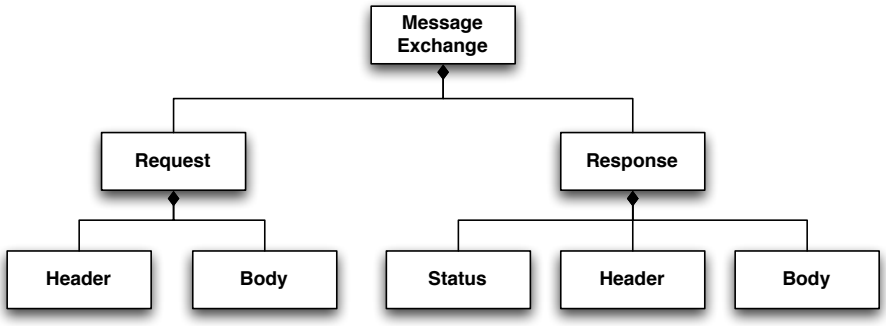

Fig. 5. URI interaction

body. The response also includes a *status*, which is part of the uniform interface of HTTP and encodes a general indication of how the request was processed.

Remember, a message exchange is always synchronous. This reduces the possible interaction patterns to *send-receive* and *receive-reply*. While it is tempting to simplify the BPEL constructs into $< send >$ and $< receive >$ elements with the complete handling of the request as child elements, the BPEL specification does not appear to allow extension activities with child elements, hence we refrain from doing so and stick with the tradtional $< send >$, $< receive >$, and $< reply >$ constructs without children. However, the newly introduced activities all have a *messageExchange* attribute by which the required data structures—as shown in figure 5 are referenced.

In figure 6 the fragment from figure 4 is completed to a complete $< onMessage >$ block. Notice the *path* attribute containing a relative URI template. The given template is relative to the BPEL process, as each instance of the process is assigned a URI itself. The exact details of the message the $< onMessage >$ activity is waiting for is described by reusing the $< method >$ element of WADL [10]. Here, the only criteria is that the message is send as a POST. WADL itself is quite descriptive and this descriptive power can be used to model pattern matching on request, i.e. several $< onMessage >$ activities waiting on the same URIs with the same verb but different contents. The $< reply >$ activity again references the $messageExchange$ data structure by attribute. Here, some con-

```
<rbpel:onMessage path="./item/new" messageExchange="createItem">
  <wadl:method name="POST" />
  <sequence>
    <assign>
      <copy>
        <from>rbpel:generate-uri("./item/{itemNumber}")</from>
        <to variable="newItemURI" />
      </copy>
    </assign>
    <rbpel:reply messageExchange="createItem">
      <rbpel:status>201</rbpel:status>
      <rbpel:param name="Location" style="header">$newItemURI</rbpel:param>
    </rbpel:reply>
  </sequence>
</rbpel:onMessage>
```

Fig. 6. Creating a shopping cart item

venience elements are shown ($< status >$ and $< param >$), their functionality could be simply mapped to $< assign >$ working on the data structure. However, this fragment shows how a new URI is generated by template and returned to the requester in the *Location Header* as outlined in the HTTP specification.

The complete BPEL for all functionality hidden in the *Add Items* activity of figure 2 is shown in figure 7.

The loop—depicted by a curved arrow on the "Add Items" activity in figure 2–is mapped to a $< repeatUntil >$ block. Upon receive a POST to the *checkout* URI the loop is left by settint the *commitRequest* variable to *true*. Also, the new internal state of the resource modeled by BPEL process outlined has a URI by itself. The requesting client is redirect to that URI by issuing a 303 status, again as outlined by the HTTP specification.

5 Related work

There are many other language available as a foundation to modeling resource behavior, such as Web Service Choreography Interface (WSCI) [17] or the Web Service Conversation Language (WSCL) [18]. However, mind share is vital to language selection and BPEL seems to be able to form a common ground for various interest groups. Also, even though some constructs my be expressed more elegantly, BPEL is designed as an open, extensible language laying a clear track of how to integrate the required functionality, as shown in the course of this paper. Describing static resource interfaces, the author is unaware of any alternative to the Web Application Description Language. On the other hand, WADL can be seen as a mashup of HTTP, RelaxNG [19], and XML Schema [20], so these standards should be mentioned here as well.

```
<repeatUntil>
  <scope
      xmlns:rbpel="http://bpt.hpi.uni-potsdam.de/ns/rbpel"
      xmlns:wadl="http://research.sun.com/wadl/2006/10">
    <eventHandlers>
      <rbpel:onMessage path="./item/{itemNumber}" messageExchange="itemShow">
        <wadl:method name="GET" />
        <!-- return representation of item $itemShow.itemNumber -->
      </rbpel:onMessage>
      <rbpel:onMessage path="./item/{itemNumber}" messageExchange="itemUpdate">
        <wadl:method name="PUT" />
        <!-- update and return item $itemUpdate.itemNumber -->
      </rbpel:onMessage>
      <rbpel:onMessage path="./item/{itemNumber}" messageExchange="itemDelete">
        <!-- delete item $itemDelete.itemNumber -->
      </rbpel:onMessage>
      <rbpel:onMessage path="./item/new" messageExchange="createItem">
        <wadl:method name="POST" />
        <sequence>
          <assign>
            <copy>
              <from>rbpel:generate-uri("./item/{itemNumber}")</from>
              <to variable="newItemURI" />
            </copy>
          </assign>
          <rbpel:reply messageExchange="createItem">
            <rbpel:status>201</rbpel:status>
            <rbpel:param name="Location" style="header">$newItemURI</rbpel:param>
          </rbpel:reply>
        </sequence>
      </rbpel:onMessage>
    </eventHandlers>
    <pick>
      <rbpel:onMessage path="./checkout" messageExchange="transfer_to_payment">
        <wadl:method href="/wadl/post/method/definition" />
        <sequence>
          <assign>
            <copy>
              <from>true</from>
              <to>$commitRequest</to>
            </copy>
          </assign>
          <rbpel:reply messageExchange="transfer_to_payment">
            <rbpel:status>303</rbpel:status>
            <rbpel:param name="Location" style="header">"./checkout"</rbpel:param>
          </rbpel:reply>
        </sequence>
      </rbpel:onMessage>
      <onAlarm>
        <for>'2h'</for>
        <exit/>
      </onAlarm>
    </pick>
  </scope>
  <condition>$commitRequest</condition>
</repeatUntil>
```

Fig. 7. Complete example of "Add Items" activity

6 Summary and Outlook

In the course of this paper, resource orientation was introduced as a viable subset of service orientation. Resource as such are complex state machines, exposing one or more uniform interfaces over time. This can be formally expressed as a complex state machine. The main contribution of this paper is to identify BPEL as a suited candidate for modeling such state machines and the necessary modifications to BPEL were outlined. All of these modifications are in the scope of the extension mechanisms of the BPEL specification.

The next steps will involve a exact specification of the extensions outlined and a reference implementation, possibly building upon an existing BPEL engine. This possibility is one of the strong arguments for using BPEL along with the already strong mind share of the BPEL community.

At the same time, a resource-oriented BPEL can be the foundation for a next-generation web framework centering around process models as the core artefact of application design.

References

1. Burbeck, S.: The tao of e-business services (2000) http://www-128.ibm.com/developerworks/library/ws-tao/.
2. Matthew, C., Laskey, K., McCabe, F., Brown, P.F., Metz, R.: Reference Model for Service Oriented Architecture 1.0. Technical Report Committee Specification 1, OASIS Open (2006) http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm.
3. Gardner, D.: Soa wikis, soa for saas, and the future of business applications. Technical report, Interarbor Solutions (2007) http://blogs.zdnet.com/Gardner/?p=2395.
4. Fielding, R.T.: Architectural styles and the design of network-based software architectures. PhD thesis, University of California, Irvine (2000) Chair-Richard N. Taylor, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.
5. Berners-Lee, T.: Www: Past, present, and future. IEEE Computer **29** (1996) 69–77
6. IBM: Web services architecture overview (2000) http://www-128.ibm.com/developerworks/webservices/library/w-ovr/.
7. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (wsdl) 1.1. Technical report, W3C (2001) http://www.w3.org/TR/wsdl.
8. Jordan, D., Evdemon, J.: Oasis web services business process execution language (wsbpel) (2007) http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
9. Overdick, H.: The resource-oriented architecture. In: 2007 IEEE Congress on Services (Services 2007). (2007) 340–347 http://doi.ieeecomputersociety.org/10.1109/SERVICES.2007.66.
10. Hadley, M.: Web application description language (2006) https://wadl.dev.java.net/.
11. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext transfer protocol – http/1.1. Technical report, The Internet Engineering Task Force (1999) http://www.ietf.org/rfc/rfc2616.

12. T.Berners-Lee, R.Fielding, L.: Uniform resource identifiers (uri): Generic syntax. Technical report, The Internet Engineering Task Force (1998) http://www.ietf.org/rfc/rfc2396.txt.
13. Helland, P.: Life beyond distributed transactions: an apostate's opinion. In: Third Biennial Conference on Innovative Data Systems Research. (2007) http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf.
14. Großkopf, A.: xbpmn. formal control flow specification of a bpmn-based process execution language. Master's thesis, Hasso Plattner Institut and SAP Research Brisbane (2007)
15. Nitzsche, J., van Lessen, T., Karastoyanova, D., Leymann, F.: Bpel light. In: 5th International Conference on Business Process Management (BPM 2007), Springer (2007)
16. Gregorio, J., Hadley, M., Nottingham, M., Orchard, D.: Uri template. Technical report, IETF (2008) http://bitworking.org/projects/URI-Templates/.
17. Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacsi-Nagy, P., Trickovic, I., Zimek, S.: Web service choreography interface (wsci). Technical report, W3C (2002)
18. Banerji, A., Bartolini, C., Beringer, D., Chopella, V., Govindarajan, K., Karp, A., Kuno, H., Lemon, M., Pogossiants, G., Sharma, S., Williams, S.: Web services conversation language (wscl). Technical report, W3C (2002)
19. Clark, J., Makoto, M.: Relax ng specification. Technical report, OASIS Open (2001)
20. Thompson, H.S., Sperberg-McQueen, C.M., Gao, S., Mendelsohn, N., Beech, D., Maloney, M.: Xml schema 1.1. Technical report, W3C (2006)