TRAP/BPEL
A Framework for Dynamic Adaptation of Composite Services

Onyeka Ezenwoye, S. Masoud Sadjadi
School of Computing and Information Sciences, Florida International University, 11200 SW 8th Street, Miami, Florida, USA
oezen001@cs.fiu.edu, sadjadi@cs.fiu.edu

Keywords: TRAP/BPEL, generic proxy, self-management, dynamic service discovery.

Abstract: TRAP/BPEL is a framework that adds autonomic behavior into existing BPEL processes *automatically* and *transparently*. We define an *autonomic* BPEL process as a composite Web service that is capable of responding to changes in its execution environment (*e.g.,* a failure in a partner Web service). Unlike other approaches, TRAP/BPEL does not require any manual modifications to the original code of the BPEL processes and there is no need to extend the BPEL language nor its BPEL engine. In this paper, we describe the details of the TRAP/BPEL framework and use a case study to demonstrate the feasibility and effectiveness of our approach.

1 INTRODUCTION

Service Oriented Computing (SOC) allows for reusable components to be dynamically discovered and integrated to create new applications. Web services play a vital role in facilitating the realization of the SOC paradigm. With Web services, autonomous, self-contained and remotely accessible components can be integrated to create composite services. The characteristics of Web services that make them so suitable for SOC also present big challenges to their reliability. The autonomy of the services in any interaction gives rise to concerns about their continued availability and trust (*i.e.,* those service will actually do what they are expected to do). The best-effort delivery method of the communication channels (*i.e.,* the Internet) through which these service interact is known to be unreliable. Also, the availability of the numerous of new services that are being developed often makes composite services quickly obsolete and leads to frequent redevelopment.

As an example, nodes on computational Grids are currently being exposed as services to ensure openness. Grid programming environments (*e.g.,* Globus) allow for the creation of applications that integrate Grid services for coordinated problem solving (*e.g.,* for Bioinformatics and Computational Chemistry). For such applications, when a Grid service partner fails, the whole application fails and has to be restarted even though there are other nodes on the Grid that can substitute for the failed service(Slominski, 2004). This problem is made more sever by the fact that such applications are often long running. This concern is a major characteristic of composite services, but it is often not addressed in the specification of composition languages. There is therefore, a need to make composite services adaptable to the changes in their execution environment. Our work focuses on adapting composite services defined in the Business Process Execution Language (BPEL), a common XML-based language for composing aggregate Web services. Specifically, we focus on how to *transparently* adapt existing composite services to encapsulate *autonomic* behavior (Kephart and Chess, 2003).

The rest of this paper is is structured as follows. Section 2 provides some background information. Section 3 motivates the need for generic proxies in more detail. Section 4 describes the TRAP/BPEL framework. In section 5, we use a case study to demonstrate the feasibility and effectiveness of our approach. Section 6 compares TRAP/BPEL to some related work. Finally, some concluding remarks and a discussion on further research directions are provided in Section 7.

2 BACKGROUND

In this section, we provide some background information that is necessary for understanding the rest of the material.

2.1 BPEL, Autonomic Computing, and Transparent Shaping

BPEL is an XML-based workflow language that can be used to create composite services that comprise other Web services An XML grammer that defines a BPEL process is interpreted and executed by a virtual machine called a BPEL engine. Although the BPEL specification provides constructs for fault and event handling, such language constructs are not sufficient to make a BPEL process deal with the failure of its partner services. Although this is a major problem for composed services, the management of such non-functional issues is assumed to be outside the language specification.

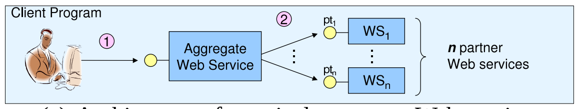
Autonomic computing (Kephart and Chess, 2003) promises to solve the management problem by embedding the management of complex systems inside the systems themselves, freeing the users from potentially overwhelming details. The ultimate goal of autonomic computing is to create *self-managing* systems that are able to function with very little direct human intervention. A Web service is said to be autonomic if it encapsulates some autonomic attributes (Gurguis and Zeid, 2005). Autonomic attributes include (1): *Self-Configuration*, for the automatic configuration of components; (2) *Self-Optimization*, for automatic monitoring and control; (3) *Self-Healing*, for automatic discovery, and management of faults; and (4) *Self-Protection*, for automatic identification and protection from attacks.

As BPEL's programming model does not sufficiently allow for the encapsulation of self-management behavior, we use Transparent Shaping (Sadjadi and McKinley, 2005) to augment BPEL processes with such behavior. *Transparent Shaping* is a programming model that provides dynamic adaptation in applications. Its goal is to adapt *existing* applications in order to better respond to changes in their non-functional requirements or execution environment. In transparent shaping, an application is augmented with *hooks* that intercept and redirect interaction to *adaptive code*. An adapted application is said to be *adapt-ready*. The adaptation is transparent because it preserves the original functional behavior and does not tangle the code that provides the new behavior (adaptive code) with the application code.

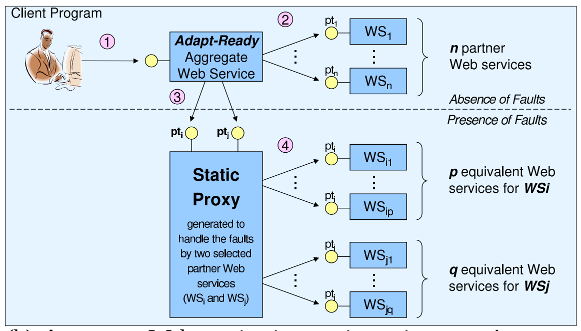
2.2 RobustBPEL

RobustBPEL (Ezenwoye and Sadjadi, 2006) is a framework that was developed as part of the transparent shaping programming model. Using RobustBPEL, an adapt-ready version of an existing BPEL process can be automatically generated to provide better fault tolerance. An adapt-ready process generated by Robust-BPEL is capable of monitoring the invocation of its Web service partners and will upon their failure invoke a proxy service through which autonomic behavior is provided.

To understand how the static proxy works , in Figure 1 we provide architectural diagrams that show the differences between the sequence of interactions among the components in a typical aggregate Web service (Figure 1(a)), and in RobustBPEL (Figure 1(b)). In a typical composite Web service, first a request is sent by the client program, then the aggregate Web service interacts with its partner Web services (*i.e.*, WS_1 to WS_n) and responds to the client. If one of the partner services fails, then the whole process is subject to failure. To avoid such situations, an adapt-ready process monitors the behavior of it partners and tries to tolerate their failure by forwarding the failed request to its proxy, which in its turn tries to find another service to substitute for the failed one.



(a) Architecture of a typical aggregate Web service.


Client Program
1
2
Adapt-Ready Aggregate Web Service
pt1
WS1
ptn
WSn
n partner Web services
3
Absence of Faults
Presence of Faults
pti
ptj
4
Static Proxy
generated to handle the faults by two selected partner Web services (WSi and WSj)
WSi1
WSip
p equivalent Web services for WSi
WSj1
WSjq
q equivalent Web services for WSj
(b) Aggregate Web service interaction using a *static* proxy.

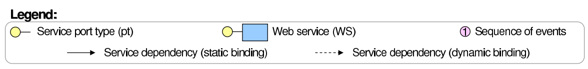

Legend:
Service port type (pt)
Web service (WS)
Sequence of events
Service dependency (static binding)
Service dependency (dynamic binding)

Figure 1: Sequence of interaction among the components in a BPEL process and its adapt-ready version in Robust-BPEL.

As monitoring all the partner Web services might not be necessary, the developer can select only a subset of Web service partners to be monitored. For example, in Figure 1(b) WS_i and WS_j have been selected for monitoring. The adapt-ready process monitors these two partner Web services and in the presence of faults it will forward the corresponding request to the *static proxy*. The static proxy is generated specifically for this adapt-ready process and provides the same port types as those of the monitored Web services (*i.e.*, pt_i and pt_j). The static proxy in its turn forwards the request to a substitute Web service. Information about the substitute services is "hardwired" into the code of this proxy at the time it is generated.

3 WHY TRAP/BPEL?

Although the RobustBPEL framework is able to provide some self-healing and self-optimizing behavior, it is limited in the level of adaptive behavior it can provide. For instance, after a service is determined to be faulty, there are no mechanisms to prevent the process from invoking that service again. Normal operation is only intercepted if a fault occurs upon the invocation of a partner Web service (Figures 1(b)), thus adaptive behavior is only exhibited at the occurrence of a fault. RobustBPEL cannot be used to provide a choice for service invocation if some other service is determined to provide better QoS than the default service in the composition.

In addition to the above limitation, each proxy service generated by RobustBPEL is *specific* to one BPEL process and cannot be reused for any other processes. Therefore, it is not possible to provide a common autonomic behavior to a set of services. This lack of support for code reuse and for scalability that comes from having numerous redundant proxies is counter to the promise of service-oriented computing. In the rest of this paper, we show how TRAP/BPEL addresses the above limitations by using a *generic* proxy.

4 THE FRAMEWORK

For the TRAP/BPEL framework we have developed a *generic* proxy that can interact with one or more adapt-ready BPEL processes.Some behavioral policy is used in the proxy to guide the adaptive behavior for each monitored service. In this section, we show the architecture of the generic proxy and explain how an adapt-ready BPEL process is generated.

4.1 High-Level Architecture

Figure 2 illustrates the architectural diagram of TRAP/BPEL at run time. As can be seen from the figure, several adapt-ready BPEL processes can be assigned to one generic proxy, which augments the BPEL processes with self-management behavior. The generic proxy uses a look-up mechanism to query a registry service at runtime to find out about available services. But unlike the static proxy (Figure 1(b)), the generic proxy has a standard interface which bears no relation to the interfaces of the monitored services. The generic proxy has as interface pt_g that is able to receive requests for any monitored Web service (*e.g.*, WS_{11} and WS_{kn}.

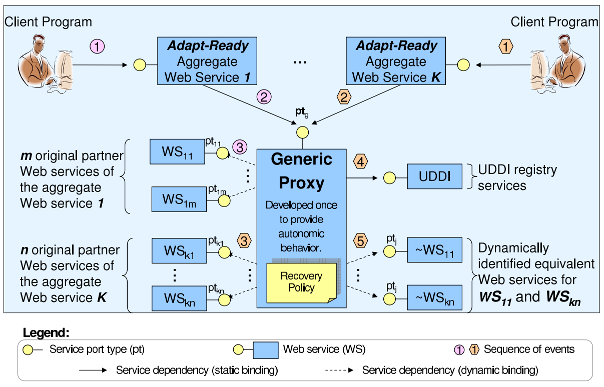

Client Program
Client Program
Adapt-Ready Aggregate Web Service 1
Adapt-Ready Aggregate Web Service K
...
1
1
2
2
pt_g
m original partner Web services of the aggregate Web service 1
WS_{11}
pt_{11}
WS_{1m}
pt_{1m}
3
Generic Proxy
Developed once to provide autonomic behavior.
Recovery Policy
4
UDDI
UDDI registry services
n original partner Web services of the aggregate Web service K
WS_{k1}
pt_{k1}
WS_{kn}
pt_{kn}
3
5
pt_j
~WS_{11}
pt_j
~WS_{kn}
Dynamically identified equivalent Web services for WS_{11} and WS_{kn}
Legend:
Service port type (pt)
Web service (WS)
Sequence of events
Service dependency (static binding)
Service dependency (dynamic binding)

Figure 2: Architectural diagram showing the sequence of interactions among the components in TRAP/BPEL

The generic proxy can provide self-management behavior either common to all adapt-ready BPEL processes or specific to each monitored invocation using some high-level policies. At this point of our research, these high-level policies are specified in a configuration file that is loaded at startup time into the generic proxy. We plan to allow runtime modification to these high-level policies in a future version of TRAP/BPEL. Figure 3 shows an example policy file where each unique monitored invocation can have a policy specified under a service element. The InvokeName element (line 4) has a value that uniquely identifies a monitored invocation in an adapt-ready BPEL process. The generic proxy checks all intercepted invocations and tries to match these invocations with the specified policies. If it finds a policy for that invocation, the proxy behaves accordingly, otherwise it follows its default behavior.

If a policy exists, the generic proxy may take one of the following actions according to the policy: (1) invoke the service being recommended in the policy (line 6); (2) find and invoke another service to substitute for the monitored service; and (3) retry the invocation of the monitored service in the event of its failure (line 10). The policy also specifies the time interval between retries (line 12). The default behavior of the proxy is to consult the registry to find and

```
<Policy>
  <Service>
    <!--name for monitored invocation-->
    <InvokeName value="WS-Invoke"/>
    <!--WSDL for a default substitute-->
    <WsdlUrl pref="true" value="f.wsdl"/>
    <!--timeout for monitored invocation-->
    <Timeout seconds="2"/>
    <!--number of retries on failure-->
    <MaxRetry value="2"/>
    <!--time to wait between retries-->
    <RetryInterval seconds="5"/>
  </Service>
  <Service>
    ...
  </Service>
</Policy>
```

Figure 3: A portion of a policy file for the generic proxy.

invoke an appropriate a service that implements the same interface as the monitored invocation.

4.2 Service Discovery

A key part of our framework lies on the ability to find services to substitute for failed services. The ability to describe the capabilities of Web services in an unambiguous and machine-readable form is vital for service requesters to be able to find suitable services. The two aspects that the World Wide Web Consortium currently defines for the full description of Web services are; (1) syntactic functional description as represented by WSDL (2) the semantics of the service and is not currently covered by a specification (Akkiraju et al., 2005). At this time, the WSDL specification (WSDL 1.1) focus on the description of the service interface. This type of description presents a limitation to automatic service discovery and composition because the interface description of a service is not sufficient to determine what a service does. Therefore, service discovery with the current specification requires some human input at some point in the selection process.

Semantic description of Web services can greatly improve service discovery and application integration since it would provide a means to expressively describe the capabilities of a service in an unambiguous machine-readable language. Although the current WSDL specification does not support semantic service description, there are several research projects (Martin et al., ; Patil et al.,) that aim to address the issue. Despite the push towards the adoption of standards for semantic description, the current UDDI (Universal Description, Discovery and Integration) specification does not support the adequate representation of semantic information in service registries since the current focus is on syntax. This limitation is however being addressed for future versions of the specification (Akkiraju et al., 2005).

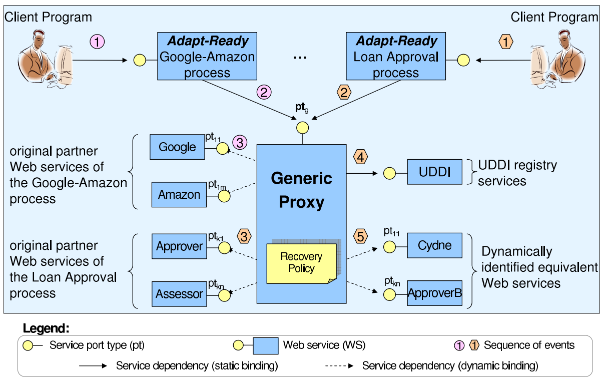

Client Program
Client Program
Adapt-Ready Google-Amazon process
...
Adapt-Ready Loan Approval process
①
①
②
②
pt_g
original partner Web services of the Google-Amazon process
Google
pt_{11}
③
Amazon
pt_{1m}
Generic Proxy
④
UDDI
UDDI registry services
original partner Web services of the Loan Approval process
Approver
pt_{k1}
③
Recovery Policy
⑤
pt_{11}
Cydne
Dynamically identified equivalent Web services
Assessor
pt_{kn}
pt_{kn}
ApproverB
Legend:
Service port type (pt)
Web service (WS)
Sequence of events
Service dependency (static binding)
Service dependency (dynamic binding)

Figure 4: The interaction between the applications and the proxy.

5 CASE STUDY

In this section, we use a case study to demonstrate the feasibility and effectiveness of TRAP/BPEL. For the case study, we use two sample BPEL process.

5.1 The BPEL Processes

The Google-Amazon business process integrates the Google Web service for spelling suggestions with the Amazon E-Commerce Web service for querying its store catalog. The business process takes as input a phrase (keywords) which is sent to the Google spell-checker for corrections. If any word in the input phrase is misspelled, the Google spell-checker sends back as reply the phrase with the misspelled words corrected (the phrase is unchanged if the spellings are correct). The reply from the Google service is used to create keyword search of the Amazon bookstore via the Amazon Web service.

The Loan-Approval process is a commonly used sample BPEL process. The Loan-Approval BPEL process is an aggregate Web service composed of two other Web services: a low-risk assessor service and a high-risk assessor service. The Loan-Approval process implements a business process that uses its two partner services to decide whether a given individual qualifies for a given loan amount. Both the business process and the risk assessor services are deployed locally. The Loan-Approval BPEL process receives as input a loan request. The loan request message comprises two variables: the name of the customer and the loan amount. If the loan amount is less than $10,000, then the low-risk assessor Web service is invoked, otherwise the high-risk assessor Web service is invoked.

5.2 The Experiment Setup and Results

We used the TRAP/BPEL generator to generate the adapt-ready versions of the two sample processes. As illustrated in Figure 4, for the Google-Amazon process, we have selected the Google spell checker service and for the Loan Approval process, we have selected the high-risk assessor service (Approver) to become adaptable. As can be followed in the figure, a policy is used to forward the intercepted calls to their original services (labels 1, 2, and 3) and use a substitute service in case an original service fails (labels 4 and 5 for the Loan Approval process).

To evaluate the performance of the TRAP/BPEL, we configured the client applications to sequentially make calls to their corresponding processes. Due to page limitations, we only show the performance chart for the Google-Amazon process in Figure 5. As the X-axis of the chart shows the number of total consequent calls is 50. The initial runs were made against the original BPEL process. The results are plotted in the chart in Figure 5 under the *original* curve. Similarly, the results of the observed completion times of the adapt-ready version are plotted in Figure 5 under the *adapted* curve.

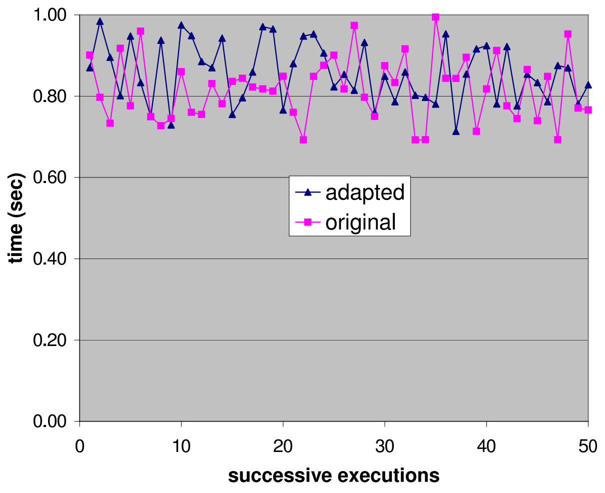


Figure 5: The response time of the Google-Amazon Process.

For the Loan-Approval process, the average completion time for the original process is approximately 0.06 seconds and for its adapt-ready version is 0.11 seconds. For the Google-Amazon process, the average completion time for the original process is approximately 0.82 seconds and for its adapt-ready version is 0.86 seconds. Although we expected that there be some performance overhead in process execution time caused by redirecting invocations through the proxy, this experiment shows that the overhead introduced by the TRAP/BPEL framework (approximately 0.045 seconds) is negligible.

6 RELATED WORK

Baresi's approach (Baresi et al., 2004) to monitoring involves the use of annotations that are stated as comments in the source BPEL program and then translated to generate a target monitored BPEL program. In addition to monitoring functional requirements, timeouts and runtime errors are also monitored. Whenever any of the monitored conditions indicates misbehavior, suitable exception handling code in the generated BPEL program will intervene. This approach is much similar to ours in that monitoring code is added after the original BPEL process has been produced. This approach achieves the desired separation of concern; however, it requires modifying the original BPEL processes *manually* and the anotation is scattered all over the original code. The manual modification of BPEL code is not only difficult and error prone, but also hinders maintainability.

Charfi et al (Charfi and Mezini, 2005) use an aspect-based container to provide middleware support for BPEL. The two inputs to the framework are the BPEL process and a deployment descriptor. The descriptor specifies the non-functional requirements (*e.g.*, security). All interactions go through the container which plugs in support for non-functional requirements. Aspects can be generated using the deployment descriptor to specify the pointcuts. Aspects specify what and how SOAP messages can be modified to add, for instance, security information. This framework is different form our because it requires a purpose built BPEL engine. Also, the adaptation is done at a much lower level (the messaging layer).

AdaptiveBPEL (Erradi et al., 2005) is much like Charfi (Charfi and Mezini, 2005), with the only major differences being that AdaptiveBPEL proposes to augment an *existing* BPEL engine with aspect weaving capabilities to address QoS concerns and adapt processes logic. In addition, adaptation is driven by a policy negotiated at runtime between the interacting endpoints.

Erradi et al. (Erradi and Maheshwari, 2005) provide reliability through a policy driven middleware named Web Services Message Bus (wsBus), which is used to transparently enact recovery actions. The wsBus intercepts the execution of composite services and transparently provides recovery services based on an extensible set of recovery policies (*e.g.*, retry, skip, and use equivalent services). The wsBus provides exception-handling and recovers from failures such as service unavailability and timeout. This approach is modular and separates the business logic of the process from the QoS requirements, however, adaptation is done at a much lower messaging layer. Our adaptation is at the application level, which simplifies maintenance of the adaptive process.

Finally, BPELJ (Blow et al., 2004) is an extension to BPEL. The goal of BPELJ is to improve the functionality and fault tolerance of BPEL processes. This is accomplished by embedding snippets of Java code in the BPEL process. This however requires a special BPEL engine, thereby limiting the portability of BPELJ processes. The works mentioned above, although are able to provide some means of monitoring for aggregate Web services, some require extensions to be made to the language or execution engine, others hinder maintainability by performing adaptation at a much lower messaging level.

7 CONCLUSION

We presented an approach to transparently incorporating self-management behavior into existing BPEL processes. Using the TRAP/BPEL framework, we demonstrated how a generic proxy can be used to encapsulate autonomic behavior through the use of self-management policies. Finally, with the use of a case study, we evaluated the performance hit introduce by the TRAP/BPEL framework, which is negligible.

In our future work, we plan to address the following issues. First, we plan to provide a GUI for developing high-level policies and enabling a developer to modify the policies at runtime. Second, we realized that the task of adding self-management behavior for multiple service collaborations is made even more complex if the collaborating services are *stateful*. We plan to investigate this problem by using Grid services, which are stateful Web services. Finally, we plan to study the existing ranking systems for service discovery.

7.1 Acknowledgements.

This work was supported in part by IBM and in part by the National Science Foundation grants OCI-0636031 and REU-0552555.

REFERENCES

Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.-T., Sheth, A., and Verma, K. (2005). Web service semantics - WSDL-S. *W3C member submission*.

Baresi, L., Ghezzi, C., and Guinea, S. (2004). Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202. ACM Press.

Birman, K. P., van Renesse, R., and Vogels, W. (2004). Adding high availability and autonomic behavior to web services. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 17–26, Edinburgh, United Kingdom. IEEE Computer Society.

Blow, M., Goland, Y., Kloppmann, M., Leymann, F., Pfau, G., Roller, D., and Rowley, M. (2004). Bpelj: BPEL for Java. *White Paper*.

Charfi, A. and Mezini, M. (2005). An aspect based process container for BPEL. In *Proceedings of the 1st Workshop on Aspect-Oriented Middleware Developement*, Genoble, France.

Erradi, A. and Maheshwari, P. (2005). wsBus: QoS-aware middleware for relaible web services interaction. In *IEEE International Conference on e-Technology, e-Commerce and e-Service*, Hong Kong, China.

Erradi, A., Maheshwari, P., and Padmanabhuni, S. (2005). Towards a policy driven framework for adaptive web services composition. In *Proceedings of International Conference on Next Generation Web Services Practices*.

Ezenwoye, O. and Sadjadi, S. M. (2006). Enabling robustness in existing BPEL processes. In *Proceedings of the 8th International Conference on Enterprise Information Systems*.

Gurguis, S. and Zeid, A. (2005). Towards autonomic web services: Achieving self-healing using web services. In *Proceedings of DEAS'05*, Missouri, USA.

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

Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., and Sycara, K. Bringing semantics to web services: The OWL-S approach. In *The First International Workshop on Semantic Web Services and Web Process Composition*.

Patil, A., Oundhakar, S., Sheth, A., and Verma, K. METEOR-S web service annotation framework. In *The Thirteenth International World Wide Web Conference*.

Sadjadi, S. M. and McKinley, P. K. (2005). Using transparent shaping and web services to support self-management of composite systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'05)*, Seattle, Washington.

Slominski, A. (2004). On using BPEL extensibility to implement OGSI and WSRF grid workflows. In *GGF10 Workshop on Workflow in Grid Systems*, Berlin, Germany.