Towards Self-healing Service Compositions

Luciano Baresi, Carlo Ghezzi, and Sam Guinea

Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza L. da Vinci 32, I-20133 Milano, Italy
baresi|ghezzi|guinea@elet.polimi.it

Abstract Service-oriented architectures are becoming the solution to integrate components in unstable and evolving contexts. The discovery phase supports flexible and dynamic component bindings. Bindings can occur either at deployment time or at run-time. Because of dynamicity, however, bindings can fail.
The paper identifies and classifies the main faults of service-oriented systems and sketches some solutions to make compositions self-healing. The proposal reorganizes a service composition in such a way that it can be suitably monitored and reorganized according to changing contexts. Proposed solutions are exemplified on a simple case study.

1 Introduction

Service-oriented architectures (SoA) [13] define a new flexible coordination paradigm as a solution for integrating components (services) in unstable and evolving contexts. In this architectural style, components can export the services they provide and clients can discover the services that fit their quality requirements. Once service requests match provisions, point-to-point interactions occur between clients and service providers. To support the matching process a third role is introduced: service brokers are placed between clients and providers to collect and advertise available services and facilitate the interaction between clients and providers.
The discovery phase can occur at different times. This impacts the degree of dynamism and uncertainty embedded in these applications. If discovery is done at *design time*, the designer selects the services by hand, without any real broker. The selected services can therefore be combined in a workflow scheme, as in the case of BPEL processes [1]. In this case, wrong design choices and the actual availability of selected services are the only problems.
If the discovery phase is done at *deployment time*, the service broker is used to “configure” the application. The set of services does not change dynamically and the binding between service requests and actual services is done once for ever (or once every time the system is restarted). This is the case, for example, of the infotainment features of modern vehicles, where control software must first discover the set of services, that is, the set of installed features, and then tailor its behavior respectively. The addition of new features is likely to lead to the shutdown of the system, which discovers the new services only after rebooting. Deployment-time selection introduces the problems associated with the discovery of services. If we do not find suitable services, we cannot

execute the application, but, if we set all bindings, services do not disappear (unless they crash).
Consider instead the case of ubiquitous applications: the changing contexts may impose different sets of services and thus the discovery phase must be postponed to *run -time*. In general, these applications require that services be selected every time they are executed. We cannot assume that a binding (request/provision) spans multiple executions, in fact the changing context could make selected services disappear and new services appear. This means that problems related to the discovery of services must be considered and handled at run-time.
If the selection phase aborts, either during deployment or at run-time, the simple shutdown of the system is not the solution, in general. The execution environment should be able to select new services and even to reorganize the process to find a solution that uses what is available, if a perfect match does not exist.
To this end, the paper identifies and classifies the main faults of service-oriented systems and sketches some solutions to make designed compositions become *self-healing* systems [19]. The proposal reorganizes a service composition, say a BPEL process, in such a way that it can be suitably monitored and customized to changing contexts. The monitoring phase adds special-purpose probes to allow the execution environment to detect anomalous conditions: for example, a service that does not answer or a service that does not match the *contract* set in the process definition. Notice that contracts predicate on the functional properties of the operations supplied by services, but also on the QoS parameters negotiated with the service suppliers [9]. The importance of non-functional requirements while setting the composition is another important feature of these systems and the capability of enforcing and monitoring them becomes a key component of the execution environment.
All the information acquired through the use of monitors is then used to handle the exceptional behaviors. The system can decide to reinvoke the same service, search for a new one, or reorganize the process to try to find an alternative solution. Our solution at this stage is quite preliminary. We reorganize processes ”locally” by exploiting single and special-purpose rules that split or merge the single nodes of the process. We do not use OWL-S-like planning strategies [11], but a graph transformation system that modifies the graph behind the composition by matching the pre- and post-conditions associated with each invocation — in the process — with those associated with the operations of available services. The paper introduces some possible solutions and exemplifies then on the case study of a *pizza delivery system*.
The rest of the paper is organized as follows. After introducing the case study, Section 2 identifies the main faults that characterize service-oriented applications and proposes possible solutions. Section 3 explains and exemplifies proposed solutions in the context of the case study. Section 4 surveys the state of the art and Section 5 concludes the paper.

1.1 Running example

The example presented in this section will be used throughout the paper to introduce the kinds of problems that can arise with complex open-world SoA scenarios and the solutions we propose to cope with them. The *Pizza Company* example was first introduced

in [10]; here we present a slightly modified version. Following the process definition of Figure 1, we can informally state the requirements of the appplication.
Suppose that a client wants to eat pizza. With a WAP enabled mobile phone, the client dials the *Pizza Company* and, after suitable identification (`Authenticate service`), his/her profile (`Profile Web Service`) determines which kinds of pizza the client likes. The `Pizza Catalog service` then offers the client four kinds of pizzas; after selecting the favorite one (Double Cheese), the client has to agree with the costs, at which point his/her credit card number (which is included in the client's profile) is validated by the `Credit Card Validation Web Service`. If everything is okay, the client's account is debited and the pizza company's account is credited. Meanwhile, the pizza baker is alerted to the order, because after the selection the pizza appears in his browser, which is integrated with his cooking gear.
At this point, the `Phone Company Web Service` is used to obtain the address of the client, by using his/her telephone number. The `GPS Web Service` is then called to get the coordinates of the delivery point. These coordinates are then passed onto a `Map Web Service`, which processes them and sends a map with the exact route to the pizza delivery boy on his PDA. The boy then only needs to deliver the pizza. In the mean time the client is sent an SMS text message on his/her mobile phone to alert about the delivery of the pizza within 20 minutes.
In the sequel we will assume that the example is implemented using an orchestrated approach, where the business process is specified using a BPEL-like language.

2 Faulty Behaviors

As we mentioned, service-oriented applications are distributed systems where services are composed by means of *brokers*: brokers let clients discover and select available services, based on their quality requirements. In this framework, all the typical faulty situations that can occur in a distributed application may arise. Additional problems may also arise in the discovery/selection phase.
Let us assume service compositions to be specified as *abstract processes*. Such processes fully model the execution flow among the different operations, specify the contracts that must be satisfied to invoke the operations, but do not identify the actual providers (services) that supply such operation. Establishing the binding with the actual providers is the responsibility of the discovery/selection phase, which may result in a *lookup failure* if no service is selected.
Besides a purely syntactical selection of services, that is, a correct matching of required and supplied interfaces, we also envisage a more semantic matching based on functional pre- and post-conditions and QoS contracts.
Given the example of Fig. 1, we could imagine that the system is unable to find a service that provides the maps of the city for free and/or with a resolution suitable for PDAs. It should also identify the best route with respect to traffic conditions. If the lookup fails, the only possible recovery activity is to modify the process to obtain the "same" results, but in a different way, that is, through a different composition of available services.
Slightly different is the case in which the application successfully binds to a service —after a lookup phase— that, at a certain point in time, stops answering to the system.

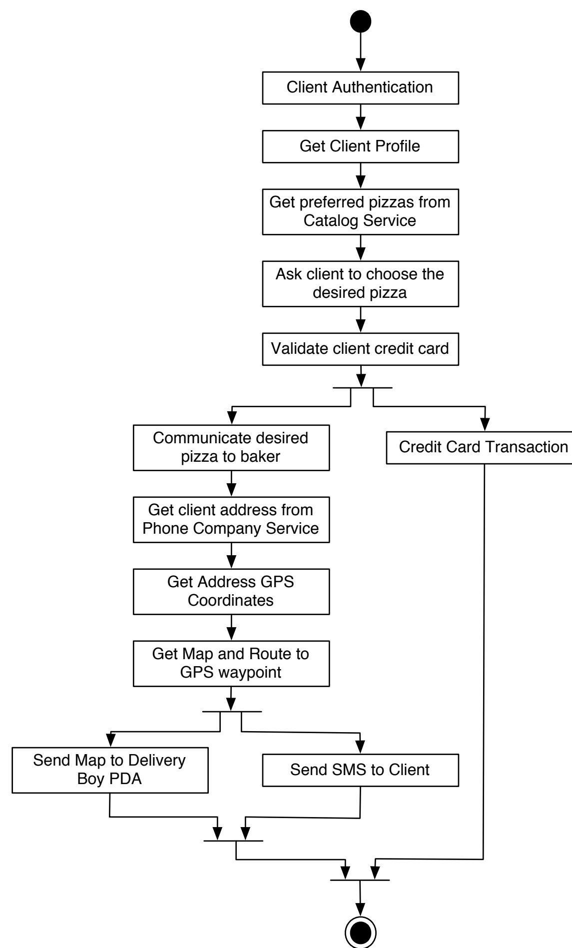


Figure 1. The *Futuristic Pizza Company* business process

The obvious approximation of such a behavior is by means of timeouts: if the answer does not arrive within the set time frame, the service is down. In the example, we can consider that the service to send the SMS message does not acknowledge the delivery of the message within 5 seconds. In this case, we could retry invoking the same service, if the failure is considered to be transient. If the problem persists, we could try to select another service that offers the same functionality, but if it does not exist, the last option is the modification of the process.

The service might also answer by throwing an exception. This requires that the process be specified in such a way that these exceptions are caught. Recovery activities are the

same as those identified so far: we can retry with the same service, try with a new one, or modify the process to obtain the same answer in a different way.
Finally, even if the service works, it might not match the contract imposed by the application, that is, it does not behave as promised. Given the distributed nature of these applications, we cannot control all service providers and thus we cannot always be sure that what is promised — and agreed during lookup — corresponds to what is actually supplied. We should be ready to react to providers who fail to provide what they promised. In the worst case, providers might deliberately cheat on us. Providers, however, can modify the exported services to improve their quality. Unfortunately, the new implementations may not always continue to satisfy the requirements on the client side. For example, the owner of the service that supplies city maps may improve its service by incrementing the resolution of its maps. This is an improvement in general, but means for more expensive services if the map has to be transferred on a PDA using GPRS/UMTS technology. In these cases, we cannot retry with the same service. We could renegotiate the QoS parameters with the provider, but negotiation is not addressed in this paper. The only feasible solution is a new lookup for either a different service or a new local composition that supplies the same functionality.

3 Proposed Solutions

The erroneous behaviors introduced in the previous section are mostly unforeseeable at design-time. For this reason, the solutions presented here rely heavily on two different mechanisms for error discovery and on special-purpose recovery actions. For run-time error discovery we propose Defensive Process Design (DPD) and Service run-time Monitoring (SrtM).
Defensive Process Design consists of designing the service-oriented business process in such a way as to permit it to cope with erroneous behaviors. Service run-time monitoring consists of using an external monitor-service capable of checking whether functional and/or non-functional contracts are violated.
Once a fault is notified, several different kinds of actions can be triggered in the attempt to reach the goals of the process, regardless of the errors. In Section 2, we briefly looked at some of the faulty behaviors that can arise in SoA scenarios; during their presentation three different recovery strategies were identified as well:

Retry: invocation of the faulty external service is retried, hoping that the fault was transient,

Dynamically bind to another service: a dynamic binding occurs to a replacement service capable of guaranteeing the same functional and/or non-functional properties of the faulty or simply unavailable service, and

Process Reorganization: a dynamic reorganization of the process at run-time, in order to overcome the problems due to a faulty or unavailable external service, for which no alternative matching service can be found.

In the sequel, the three strategies will be called *retry*, *rebind*, and *restructure* We will briefly go over both the approaches for run-time error discovery and the recovery actions introduced in Section 2, to clarify where and when each solution can play an important role in self-healing service compositions.

3.1 Defensive Process Design

This approach consists of designing the process in such a way as to give it the possibility to recover from certain types of errors that can be very common at execution-time in open-world scenarios. Timeout and exception errors can be easily addressed by use of smart process design.
In an orchestrated composition, timeout errors can cause the failure of the entire process if defensive design is not used. Starting from an orchestrated BPEL-like composition language, our approach encapsulates the remote service invocation in a scope and provides a timeout clause. This clause defines the amount of time we are willing to wait for the external service to respond to the invocation. If the timeout interval expires without the process receiving an answer from the external service, an exception —internal to the process execution domain— is launched. This way, graceful termination of the process or possible recovery strategies can be triggered. For example, in our *Pizza Company* example, the BPEL-like code we would use to recover from the SMS service not responding on time would be:

```
<scope name = "SMSScope">
  <eventHandlers>
    <onAlarm for="PT5S">
      <retry>
        <times>5</times>
        <finally>continue</finally>
      </retry>
    </onAlarm>
  </eventHandler>
  <invoke operation="sendSMS"
     inputVariable="inVar" outputVariable="outVar"/>
</scope>
```

In the example code, the service invocation is surrounded by a scope called `SMSScope` to which a timer is attached. This timer waits for exactly five seconds before giving up on the SMS service and executing the defined recovery actions. The exception handling action defined in this example is to retry once the timeout has expired. The process retries five times before giving up. At that point, the orchestration engine continues to execute the process and renounce to communicating to the client that his/her pizza is on its way.
Defensive process design can also help the designer recover from exception errors. In fact, it is possible to define a scope for each service invocation and to attach fault handlers to these scopes. In this way, a recovery strategy can be triggered every time an exception is thrown. In a similar way as to using traditional `try`, `catch` and `finally` clauses in object-oriented programming, it is possible to define fault handlers both for specific exceptions and for exceptions unknown at design-time. In our example, the code produced to overcome the problem of the `Credit Card Validation Web Service` throwing an exception (and potentially causing damage to the entire business process) could be:

```
<scope name = "CCAuthenticationScope">
  <faultHandlers>
    <catchAll>
```

```
      <retry>
        <times>2</times>
        <finally>communicate&terminate</finally>
      </retry>
    </catchAll>
  </faultHandlers>
  <invoke operation="CCAuth"
     inputVariable="inVar" outputVariable="outVar"/>
</scope>
```

In the example, in case the `CCAuth` operation responds by throwing an exception, the process retries to authenticate the credit card a second time before giving up. Since this business process should not continue if the pizza cannot be paid for, a failure of the second try causes the process to communicate an error to the client and terminate gracefully.
It is not necessary to define only one of the three recovery actions (retry, rebind, restructure) for each erroneous behavior. It is possible —and advisable— to define a hierarchical chain of recovery actions. For example, to cope with a service failure, the designer could decide first to re-invoke the service a few times (retry) before trying to replace the service with a compatible substitute (rebind). If this strategy fails as well, process reorganization should be tried (restructure).

3.2 Monitoring

Run-time monitoring can be used to verify if a service is providing the functional and/or non-functional properties it professes to possess. Our approach to service monitoring [15] is assertion-based. It consists of applying contracts —in the form of pre- and post-conditions— to remote services. Based on these assertions, the monitor can verify the correctness of service invocations. To do so, assertions are checked on data coming either from within the defined process (from which remote services are invoked) or from an external source (i.e. from the invocation of a metering service).
To impact as little as possible on existing composition languages (many of which are in the process of being standardized), we propose to keep the business logic separate from the monitoring logic. We achieve this by implementing the monitor as a remotely invokable service. This gives us a general methodology that is applicable to many different kinds of composition technologies. For example, within a BPEL-like orchestrated composition, the monitor can be invoked at any given time, just as a normal remote service would be. The only difference would consist in the parameters that must be passed to the monitoring-service: the assertions to be verified and the data on which to verify them. If an assertion happens to be false at run-time, the monitor informs the process engine, which proceeds to launch a recovery action.
This approach can be used for both functional and non-functional contracts. The difference lies in the nature of the assertions the designer negotiates with the services used in the composition. In the *Pizza Company* example, the breaking of a functional contract can arise when the `GPS Web Service` is invoked to retrieve the client's home coordinates. The post-condition of the operation states that standard and correct UTM (Universal Transverse Mercator) coordinates must be given, since that is the required

input for the `Map Web Service`. This means that the return message of the invocation must contain coordinates that are syntactically correct, complete and containing reasonable values (i.e. belonging to a ”reasonable” range). After invoking the `GPS Web Service`, its return value constitutes the data (shown below) that must be sent to the monitoring service, together with the contract assertions (defined informally in XML for the sake of simplicity and readability):

```
<data>
  <zone>18</zone>
  <easting>435000E</easting>
  <northing>276000N</northing>
</data>
<assertion>
  correct(zone, easting, northing)
 </assertion>
```

Should the assertion be false, meaning that the returned values where not UTM compliant, the recovery strategy could try to rebind to a new service capable of offering the same functionality. This requires the system do a simple lookup of an equivalent service.
An example where a non-functional property is checked can be seen when the `Map Web Service` is asked to retrieve a map containing the route that should be followed to deliver the pizza. Since the map must be used by the pizza delivery boy on his PDA, the negotiated contract requires that a fairly low resolution map should be returned (i.e. a maximum resolution is specified). In this case, the resolution of the map received from the `Map Web Service` and the desired resolution represent the data sent to the monitor service:

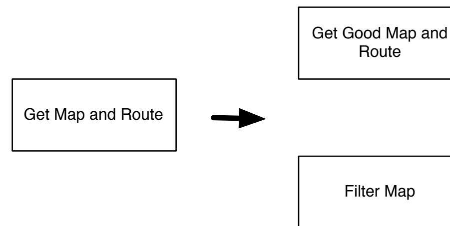
```
<data>
  <hres>800</hres>
  <vres>600</vres>
</data>
<assertion>
  <and>
    <lessEqual>
      <datum>hres</datum>
      <datum>80</datum>
    </lessEqual>
    <lessEqual>
      <datum>vres</datum>
      <datum>60</datum>
    </lessEqual>
  </and>
</assertion>
```

If the resolution of the returned map is higher than the one specified in the contract, the designer can decide to reorganize the process by introducing an invocation to a service capable of reducing the resolution to the needed size. This would require both a reorganization of the process and a lookup in order to discover and dynamically bind the new services to the process,

3.3 Recovery strategies

Recovery strategies allow a process to continue its execution even in case of faulty behaviors. We already said that these actions can be organized hierarchically and thus the system can start from a simple action, like retrying the same service, and then move to more complex recovery strategies.
As we saw, the retry strategy can be attempted a certain number of times. We can also invoke the *lookup* service that tries to rebind to another service before the process can continue its execution[1]. The third strategy we propose (restructure) is based on a *local reorganization* of the process. In this case we consider the BPEL-like process definition as a direct graph and we apply graph transformation rules to modify its topology. Rules are simple and work only locally. Each rule predicates on the node (invocation) not matched by the lookup procedure and its neighbors. These rules can transform a single node into a sequence of two or more nodes, into a parallel composition of two thread (nodes), or into a branch. Similarly, they can transform sequences, parallel compositions and branches into single nodes. Rules are applied only if the pre- and post-conditions of the operations supplied by available services match those required by the abstract process. For example, we can split a single node n of the abstract process into a sequence of two nodes n_1 and n_2 if the pre-condition of n is the same as that of n_1, the

[1] The actual negotiation of quality parameters is out of the scope of this paper.




```
Get Map and Route(c: Coordinates):
pre: c.zone = "New York"
post: hres(r.map) = 80 and vres(r.map) = 60

Get Good Map and Route(c: Coordinates)
pre: c.zone = "New York"
post: hres(r.map) = 800 and vres(r.map) = 600

Filter Map(m: Map; f: factor)
pre: TRUE
post: hres(r.map) = hres(m.map)/factor and
        vres(r.map) = vres(m.map)/factor
```

Figure 2. Example process transformation

post-condition of n is the same as that of n_2, and the post-condition of n_1 implies the pre-condition of n_2. The other rules are governed by more complex application conditions; the whole set is defined in [14].
The rule that splits a node into a sequence can be applied on an excerpt of the example process of Fig. 1. Let us assume that the operation `Get Map and Route`, with the pre- and post-conditions of Fig. 2, cannot be supplied by any actual service. The service broker offers a service that returns maps and routes with higher resolution and also a service that can filter maps lower their resolution. The simple transformation of the process, along with the pre- and post-conditions of required and supplied operations are shown in Fig. 2, where `r` is the result produced by the operation; `hres` and `vres` returns the horizontal and vertical resolution of a map.

4 Related work

Our approach to self-healing compositions originates from assertion systems previously defined in the fields of programming languages and object oriented design [20]. Eiffel [7] is a well known programming language that embeds assertions —pre-, post-conditions and invariants— and recovery actions directly into the code. Other assertion systems, which provided a valuable input to our research, are iContract [3], one of the first assertion systems available for Java, and Anna [2], an extension to the Ada programming language [12]. Anna allows for the addition of *virtual code* to control and probe the software under test. Our notion of annotations and transformations originates from this concept.
Ongoing research in the field of monitoring services in open-world scenarios has taken a number of diverse directions. Of particular interest is Robinson's work [4], which can be viewed as a requirements analysis approach to the monitoring problem. Requirements analysis derives goals and obstacles. Obstacles define situations which, if encountered, prevent the system from reaching its goals. Run-time monitors are then deduced from the obstacles the system must look out for. Other solutions to the monitoring problem can be found in the research and development done in the field of SLA-based approaches. IBM's work in the development of WSLA [8][9] has brought to the development of the WSLA Framework, an implementation of a system capable of monitoring WSLA contracts at run-time. In [15], Ghezzi et al. propose an assertion-based approach to monitoring service compositions. For each service of the composition, functional contracts are defined in terms of assertions (pre- and post-conditions). Monitor services —external to the composed process— are then used to verify the correctness of the contracts. Two different implementations are illustrated: one based on a first-order logic (CLIX) and one based on the properties of a fully fledged object oriented programming language (C#).
Many different approaches have been proposed for solving the dynamic service composition problem. In particular, WSCG (Web Service Composition Graph) [18] proposes a framework that provides visual design, validation and development of compositions using graph theories. It uses a transformation engine based on the definition of pre- and post-conditions for the nodes of the graph that represent single services. Other approaches are based on the use of Petri Nets. In one of these, Narayanan et al. con-

sider encoding DAML-S web services in petri nets and providing decision procedures for simulation, verification and composition [17]. In another, Pernici et al. propose a framework that, thanks to a proprietary service description, can orchestrate dynamically composed service compositions using petri net representations [16].

5 Conclusions and future work

The paper has presented a first characterization of the faulty behaviors typical of service-oriented systems. The different degrees of dynamism and uncertainty can lead to problems in the process of discovering and selecting services, but also in the "normal" execution of the system. The simplest case is when a service does not answer to its clients, but it could also raise exceptions or violate the contracts — functional requirements and QoS — negotiated with the client. The proposed solutions exploit probes to monitor the execution of the composition and suitable recovery activities to make the system continue its execution: this is a move towards *self-healing* compositions of services.
Our future work includes a better assessment and fine tuning of the approach. More precisely, we still need to thoroughly investigate the rule-based approach to reorganize compositions to stress its applicability and design a sufficient set of rules. In the meanwhile, we are applying these ideas to new and more complex case studies and we are building a prototype CASE tool to model and annotate compositions of services. This is implemented as an add-in to the Eclipse framework [6].

References

1. BEA, IBM, Microsoft, SAP and Siebel. Business Process Execution Language for Web Services Version 1.1. 2003.
2. D.C. Luckham. Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs. *Texts and Monographs in Computer Science*, Oct 1990.
3. R. Kramer. iContract - The Java Design by Contract Tool. In *Technology of Object-Oriented Languages. TOOLS 26 Proceedings*, pages 295–307, Aug 1998.
4. W. Robinson. Monitoring web service requirements. In *Proccedings of the International Conference on Requirements Engineering*, 2003.
5. The World Wide Web Consortium (W3C). Web Service Choreography Interface (WSCI) 1.0. 2002.
6. Eclipse. Eclipse. 2004. `http://www.eclipse.org`.
7. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.
8. IBM Research Report. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. 2002.
9. IBM Corporation. Web Service Level Agreement (WSLA) Language Specification. 2003.
10. IBM T.J. Watson Research Center. The Futuristic Pizza Company Example. 2004. `http://researchweb.watson.ibm.com`.
11. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services. 2003. `http://www.daml.org/services/owl-s/1.0/owl-s.html`.
12. R. Riehle. *AdA Distilled - An Introduction to Ada Programming for Experienced Computer Programmers*. AdaWorks Software Engineering. `http://adaworks.com`.

13. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard. Web Services Architecture. 2004. http://dev.w3.org/cvsweb/~checkout~/2002/ws/arch/wsa/wd-wsa-arch-review2.html.
14. T. Lehner. *Dynamic Reconfiguration of BPEL Processes - Master Thesis - Politecnico di Milano and Universitat Passau Fakultat fur Mathematik und Informatik*. 2004
15. L. Baresi, C. Ghezzi and S. Guinea. Smart Monitors for Composed Services. *ICSOC'04. ACM. To appear*, 2004.
16. F.P. Pesicce, M. Mecella and B. Pernici. Modeling E-Service Orchestration through Petri Nets. *TES*, 2002, pages 38-47.
17. S.A. McIlraith and S. Narayanan. Simulation, Verification and Automated Composition of We Services. *Eleventh International World Wide Web Conference (WWW2002)*.
18. B. Jun, Z. Ren and J. Li. A New Web Application Development Methodology: Web Service Composition. *WES*, 2003.
19. R. Delemos, C. Gacek and A. Romanovsky. Workshop on Software Architectures for Dependable Systems (WADS). *ICSE 2003*.
20. D.S. Rosenblum. Assertions a Decade Later. *Presentation at ICSE 2002 for ICSE 1992 Most Influential Paper Award.* http://www.cs.ucl.ac.uk/staff/D.Rosenblum/presentations.html.