A Fault Taxonomy for Web Service Composition

K.S. May Chan[1], Judith Bishop[1], Johan Steyn[1], Luciano Baresi[2], and Sam Guinea[2]

[1] Department of Computer Science, University of Pretoria, Pretoria, South Africa
[2] Politecnico di Milano, Dipartimento di Elettronica e Informazione Milano, Italy
{ksmchan,jbishop}@cs.up.ac.za, johan.steyn@elogics.co.za, {baresi,guinea}@elet.polimi.it

Abstract. Web services are becoming progressively popular in the building of both inter- and intra-enterprise business processes. These processes are composed from existing Web services based on defined requirements. In collecting together the services for such a composition, developers can employ languages and standards for the Web that facilitate the automation of Web service discovery, execution, composition and interoperation. However, there is no guarantee that a composition of even very good services will always work. Mechanisms are being developed to monitor a composition and to detect and recover from faults automatically. A key factor in such self-healing is to know what faults to look for. If the nature of a fault is known, the system can suggest a suitable recovery mechanism sooner. This paper proposes a novel taxonomy that captures the possible failures that can arise in Web service composition, and classifies the faults that might cause them. The taxonomy covers physical, development and interaction faults that can cause a variety of observable failures in a system's normal operation. An important use of the taxonomy is identifying the faults that can be excluded when a failure occurs. Examples of using the taxonomy are presented.

Keywords: Web services, service composition, fault taxonomy, self-healing.

1 Introduction

Web services are increasingly used as the key building blocks for creating inter- and intra-enterprise business processes. Such business processes are created through the composition of several existing Web services in a logical order that satisfies certain requirements. Earlier work on Web service composition focused on developing languages and standards to specify and configure the composition process. WS-BPEL [14] and BPML [1] enable the description, definition and modelling of compositions while OWL-S [12] is an ontology for marking up the properties and capabilities of Web services. All of these aim to facilitate the automation of Web service discovery, execution, composition and interoperation.

Web services themselves are inherently dynamic, and cannot ever be assumed to be stable. It is in their very nature to reflect the latest version of an activity or function. There is no guarantee that the natural evolution of a service will not change its interface, and therefore upset a previously working composition. Previous work on the composition of services has identified that the discovery phase of service composition is crucial, and has proposed run-time monitoring and reaction strategies to make compositions self-healing [3]. Other work has considered quality of service [5], model checking and verification [7], [13] and proactive failure handling [6]. The trend is towards the development of mechanisms to monitor the composition and to detect and recover from faults automatically, that is, to be able to perform self-healing. In working with Web service compositions [18], we realised that a key success factor is to know what faults to look for, in order to suggest and apply a suitable recovery mechanism. Thus we need knowledge of what possible faults can occur, their causes as well as their effects.

The distinction between a fault and a failure is addressed in document ISO/CD 10303-226, where a *fault* is defined as an abnormal condition or defect at the component, equipment, or sub-system level which may lead to a *failure*. In their seminal work, Avižienis *et al* [2] further refine the definition of the creation and manifestation mechanisms of faults and failure. Faults cycle between dormant and active states, and a service failure occurs when a fault becomes active (through the application of some input pattern) and is propagated beyond the service interface. In Web service composition, the reason for the failures can easily be lost. Classifying the faults can assist in pinpointing the cause of failures.

This paper proposes a novel taxonomy that captures the possible cause of failures that can arise in a Web service composition and classifies them as faults. The taxonomy proposed can be used to distinguish between different observed failures and thereby aid recovery. Our classification of Web service composition faults is based on discussions in [2], [10], [17] and [19], on examples that we have run [18], and on the related work in [11]. Although the taxonomy is important, a further contribution of the paper is the discussion and elucidation of the various faults that can occur in Web services. In the examples we have used to illustrate different faults that make up the taxonomy, we make use of examples in Oracle BPEL Process Manager Suite [15].

The remainder of the paper is organised as follows: in Section 2 we look at the background to this work, specifically the relationship between Web services and software components (where faults are already being classified), and we mention how failures can be detected automatically via monitoring. In Section 3, we consider the relationship between cause and effect and discuss in detail the faults that can occur. Section 4 presents our fault taxonomy for Web service composition. In Section 5 we discuss current work on correctness of Web service composition that related to the classification of faults. Lastly, we conclude our findings in Section 6.

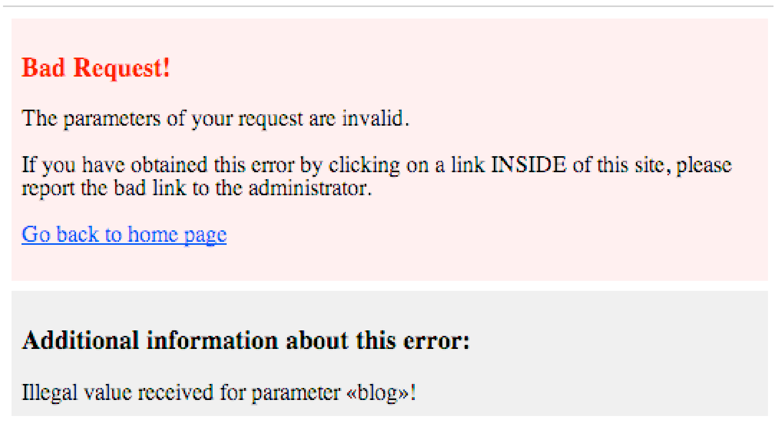
2 Background

Recently, a taxonomy has been developed for faults in component-based software [11] based on the premise that

- high quality components are necessary, but not sufficient, to ensure the quality of composite systems, and
- classical approaches to analysis and test are not directly applicable to component software.

The same is true of Web services: high quality services can still fail if incorrectly composed. There are many similarities between software components and Web services [8], but the differences suggest that a new taxonomy for faults is required. Web services execute remotely, whereas components are mostly downloaded to execute locally on the client have to deal with considerable heterogeneity in platforms and languages, whereas in the component world this is limited or contained by middleware. This means that in the world of Web services, clients are being faced with error messages that would previously only have been seen by developers. It is therefore important that the faults that cause these failures can be classified and repaired as soon as possible. If the system can distinguish between the different types of failures, we can then start to apply the correct failure recovery method for that failure. For example, if a time-out fault or availability fault occurs, we can first retry to invoke the service before trying another error recovery method. If the fault persists, we can then try to use another recovery method (rebinding to a new service, recomposing the composite service etc.).

Consider the following real example which shows the response from a blog site accessed from a technical forum (Figure 1). The identified error here had evidently to do with the way in which the connection between the services was being used.



Bad Request!

The parameters of your request are invalid.

If you have obtained this error by clicking on a link INSIDE of this site, please report the bad link to the administrator.

Go back to home page

Additional information about this error:

Illegal value received for parameter «blog»!

Fig. 1. Resulting of accessing a technical blog

If the developer or the deployment software can identify what seems to be a faulty service or faulty connection, they can replace it. However, doing so does not guarantee the correctness of the next composition. Services could be inherently incompatible, and such a fault would remain even after replacement. Therefore it is wise to identify the fault itself rather than service where the failure was observed.

Failures do not always reveal themselves on a user's screen. They can be detected by software, since most failures are capable of producing some error message or undesired result. Our earlier work [3] discusses methods for failure detection in their self-healing approach. The proposed strategy is to retry the invocation and if that fails, the system tries to rebind to another service. In the worst case (if the previous methods fail) the system can try to do a local reorganization of the composite service. While these methods can be used at run-time, there also exist methods that can be used in off-line mode [16] and to some extent can classify what type of fault had occurred.

3 The Causes of Failure in Web Services

As we have seen, languages that are being developed to detect failures in Web services can relate the effects observed to a possible cause, and then pass the information to the recovery process. Different causes (faults) clearly impose different reactions, but a case-by-case decision is not feasible. The identification of classes of faults helps abstract the problem, identify generalize reactions (or reaction patterns), and also organize new faults to automatically identify the most suitable reactions. Following the work of [2], we start by dividing faults into three major partially overlapping groups: physical, development and interaction faults.

3.1 Physical Faults

Physical faults are observed as failures in the network medium, or failures on the server side. They include communication infrastructure exceptions and failures in the correct operation of the middleware of the hosting servers. Good examples of such failures would be a server that is out of action or a severed connection to the server. Web services are the building blocks for Web service composition. In most trivial cases, a composition fails because of the building blocks are **unavailable**. The availability of a Web service is influenced by the server and by the networking media.

There are two causes of service unavailability – the service is down or the network connection to the service is down. It is difficult to pinpoint the exact problem but any failure that results in no response from the above two forms can be classified as an unavailability fault. This fault, and any other fault residing at the host, can only be fixed by the service provider.

3.2 Development Faults

Development faults may be introduced into a system by its environment, especially by human developers, development tools, and production facilities. Such development faults may contribute to partial or complete failures, or they may remain undetected until the use phase.

A **parameter incompatibility fault** is the case of a service receiving incorrect arguments, or incorrect parameter types as input. Under normal circumstances, such a fault can be avoided if catch blocks and exception handlers are used. Consider the error message in Figure 2, as produced by Oracle BPEL Process Manager Suite [15]. In the experiment, the service that was invoked expected an integer value as input, but received a string value instead. The service failed to be invoked since we did not give it the appropriate exception handling mechanisms.

```
Message handle error.

An exception occurred while attempting to process the message
"com.collaxa.cube.engine.dispatch.message.invoke.

InvokeInstanceMessage"; the exception is: XPath expression failed to
execute.

Error while processing xpath expression, the expression is
"((bpws:getVariableData("inputVariable", "payload",
"/client:DummyService_3ProcessRequest/client:input") mod 2.0) = 1.0)",
the reason is NaN is not an integer.

Please verify the xpath query.
```

Fig. 2. Error produced on incorrect input

This type of failure can occur when using a composite service. If we dynamically compose a composite service, we might end up using two services that send incompatible types to each other. This can be handled by an appropriate translator between the two services, but only if we are able to catch the failure. In some cases, if a Web service is invoked using incorrect parameter types, the service will return an error message or error code that can be used to identify what type of fault occurred, as shown in Figure 2. When using Web services in the .NET environment, translation tools exist that will translate the Web service's WSDL file into usable and readable program code that can be used to invoke the service. When using a Web service in such a way it is almost impossible to cause such a fault since the compiler will pick up on an incompatible parameter before it can be invoked.

A Web service often does not have full knowledge about services that it will interact with. Selection of the services is based on the interfaces or ontologies (descriptions) provided. Often assumptions need to be made during the selection procedure, but these assumptions can be violated. This could happen simply because the developers are too optimistic or pessimistic about their assumptions or the service update changes its interface. After the update, the **interface might have changed** and subsequent queries to the old interface would fail.

In the worst case, the interface changes, but the workflow logic stays the same and this can cause a **workflow inconsistency** problem. In the case of such a fault, the service cannot be invoked since the interface does not correspond to the workflow description. If service users are not informed about such changes, they would assume that the service is broken or no longer exists. These types of faults can occur after updating the Web service description. Unfortunately these errors are also the hardest to pick up, since they will behave very much like a physical fault and might even be mistaken for a physical fault unless the service returns a sensible and usable notification of the type of error that occurred.

As an example, consider the following pieces of code. In the example code, the BPEL code was left unchanged, but the developer changed the interface (the WSDL code). If such a change was made whilst a requester was using this service, it would become unavailable.

The highlighted sections of Figures 3 and 4 represent the parts that were affected. The interface to the service was changed (the WSDL code) and the workflow code was left unchanged (the BPEL code). Then some of the variables' message types were changed in the interface (WSDL code). The input variable was changed from **MapServiceRequestMessage** to **MapServiceInvoked-Message**. The fault variable was also affected.

A **fault due to non-deterministic actions** is the case where a service will have more than one possible outcome or return value. Such a service might produce any one of a number of outputs that are determined by an operation that might or might not precede the output. Even though such services are rare and difficult to construct, they can occur if they are created without the help of a tool.

```
<partnerLinks>
    <partnerLink name="client" partnerLinkType="tns:MapService"
                  myRole="MapServiceProvider"/>
</partnerLinks>
<variables>
    <variable name="input" messageType="tns:MapServiceRequestMessage"/>
    <variable name="output"
              messageType="tns:MapServiceResponseMessage"/>
    <variable name="fault" messageType="tns:MapServiceFaultMessage"/>
  </variables>
```

Fig. 3. BPEL Code of service with Workflow error

3.3 Interaction Faults

Faults from various classes can manifest and propagate from one service to another during the execution phase of the composite service. The faults may cause unacceptably degraded performance or total failure to deliver the specified service.

An interaction faulsts occurs when the given composite service fails more frequently or more severely than acceptable. Overall, interaction faults can be further sub-divided into the following categories:

- Content fault: include incorrect service, misunderstood behaviour, response error, QoS and SLA faults.
- Timing faults: include incorrect order, time out, misbehaving workflow faults.

Content faults occur when the content of the service delivered, based on the service description, deviates from the expected composite service, whereas timing failures are concerned with the time of arrival or the timing of the service delivery that will result in the system to deviate from its originally specified functional requirement.

Integration of Web services into a composite service requires communication between all participating Web services. The communication protocol Web services use to carry messages between each other is normally Simple Object Access Protocol (SOAP). During message-passing the message packets may arrive in a different order to the order in which they were sent. This can cause an **incorrect order** fault. The cause of this scenario is often due to a slow network. In a composition, services are often dependent on each other to reach a desired result. In the case where a message arrives in a different order it may cause undesirable results. A possible solution to this problem is to make use of Lamport timestamps [9].

```
<types>
  <schema>
       <element name="request" type="string"/>
       <element name="response" type="string"/>
       <element name="error" type="string" />
  </schema>
</types>

<message name="MapServiceInvokedMessage">
  <part name="payload" element="tns:request"/>
</message>
<message name="MapServiceResponseMessage">
  <part name="payload" element="tns:response"/>
</message>
<message name="MapServiceErrorMessage">
  <part name="payload" element="tns:error" />
</message>

<portType name="MapService">
  <operation name="process">
       <input message="tns:MapServiceInvokedMessage"/>
       <output message="tns:MapServiceResponseMessage"/>
       <fault name="MapNotFound" message="tns:MapServiceErrorMessage"/>
  </operation>
</portType>
```

Fig. 4. Corresponding WSDL code of service

Another possible fault caused by a slow network is a **time-out** exception. If proper measures are taken, time-out exception and communication medium failures can be caught in time using exception handlers. For example, catch blocks and exception handlers in BPEL can be used to recover from certain failures and exceptions.

There are several classes of faults that form part of the interaction fault category, one of which is **misbehaving execution flow**. Dynamic service composition means creating a composite service on the fly. Predefined interaction is not possible and it varies according to the dynamic situation. This increases the probability of a fault, because a single service may have several dependencies that are stimulated to correctly perform their tasks, but the developers may not know the identity of the services that will fulfil the request. In the same way, the service that will be used to satisfy a dependency may not be aware of this fact until deployment. To better understand a misbehaving execution flow fault, consider the example shown in Figure 5. The composite service is defined by its interface and the internal workflow of the composition that is contributed by participating Web services (w_a, …, w_z). A misbehaving workflow fault may therefore be caused by one of the following:

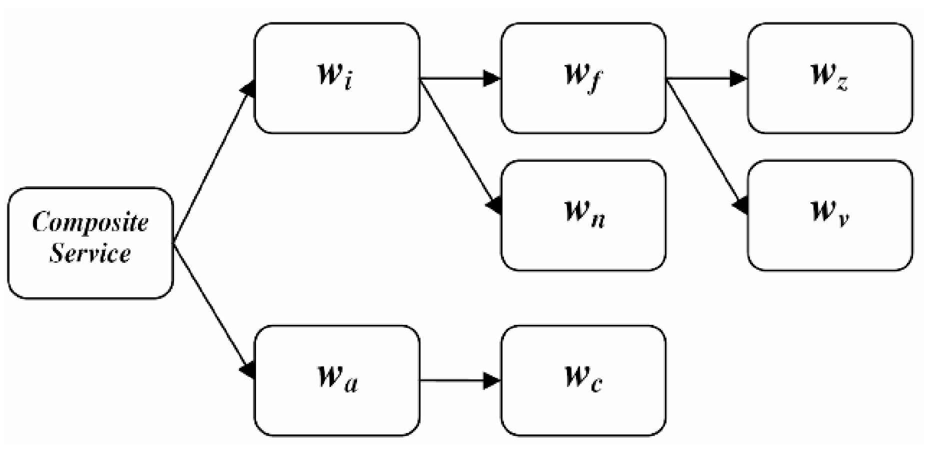


Fig. 5. Composite service example

- The workflow of a composite service is not correct. That is, the result returned is not what was expected or the service returns no results at all.
- An individual service (e.g. w_c) used by a composite service is not correct, or
- An individual service (e.g. w_f) used by a composite service does not work well with other participating services.

A **misunderstood behaviour fault** is the case of a service requester that requests a service from a service provider, expecting a service different from the provided one. A simple example would be if the requester requests a service for stock exchange quotes, and the provider returns a service supplying exchange rate quotes. These types of faults can occur if the description of a service is incorrect, or if the service provider misinterpreted the request from the requester.

Response faults are closely related to behaviour faults and incompatible input. A service that exhibits this type of faults will sometimes produce incorrect results even if the correct input was received. The incorrect results will be caused by incorrect internal logic of the service. Some faulty services might even randomly produce outputs for a requester. In these cases, it would be better to rebind to a different service. While response faults generate wrong values, they are often due to the misunderstanding of service behaviour and wrong input values.

Non-functional aspects of Web services include the **Service Level Agreement (SLA)** and the **Quality of Service (QoS)** agreement. These non-functional properties can be captured at runtime and used to monitor the performance of the service with respect to the contract between different parties.

According to Ludwig *et al.* [10], SLAs are used for the reservation of service capacity. They were traditionally used only between organisations to reserve the use of services between them. SLAs have only recently been used as a way to ensure delivery of services in Web services and Service-Oriented Architecture (SOA). A fault caused by or during the SLA will manifest in the requester receiving an incorrect service. During SLA negotiations all parties involved will have to agree on the interface and the service provided by the provider. If the provider is providing a service that was not originally promised or agreed upon, then we have a SLA disagreement fault.

QoS includes not only SLAs, but also the promise to deliver quality service in terms of speed and information. If any of these factors are not of good quality (i.e. if the speed is slow or if there are delays in the responses from the service) then we have a fault caused by the QoS factors. There are ways to use QoS constraints to ensure that a service can deliver what is promised. Yu and Lin [21] propose a method for binding to a service by making use of QoS constraints as a heuristic to choose the service. Faults caused by these, and other non-functional aspects, can only be caught after execution and use of the service, and the best way to fix them would be to rebind to a different service that promises to offer a better quality of service.

An **incorrect service** fault occurs when the provider provides a service under false pretences. It does not mean that the service is not working. The service might be working and might deliver the best results yet, but it is not the service that you requested. This can happen if the ontological description of the service is incorrect. It can also happen if the WSDL description of what the service is incorrect. This fault, most of the time, goes hand in hand with SLA and QoS faults. In such cases we need to rebind to another service since the service is not as requested.

4 A Fault Taxonomy

Given the above discussion of faults and the observed effects, we have distilled all the information obtained into a representation based on that in [2] as shown in Figure 6. The original three broad categories of faults are shown along the top of the taxonomy, broken down into the subcategories identified in Section 3. At the bottom of the matrix are the observed effects. An effect can be observed in more than one way. For example, the second fault, interface change, can causes unresponsive service, incorrect results or incoherent results.

Avižienis *et al* [2] identify 16 elementary fault classes, of which we recognize the following six as being relevant for Web services:

- Development faults which occur during system development or maintenance
- Operational faults that occur during service delivery
- Internal faults originating inside the system boundary
- External faults that originate outside the system boundary and are propagated into the system by interaction or interference
- Hardware faults that originate in or affect the hardware
- Software faults that affect programs or data.

(Note that the reuse of the term "Development" in our broad categories and these classes cannot be avoided.) These classes are not at the same level, but are seen from viewpoints during the life cycle of a system. For example, the first two refer to the phase of occurrence, the second two to system boundaries and the last two to a fundamental system dimension. Our taxonomy does not address these viewpoints, but rather provides the user with knowledge of fault classes to be included in the specification of self-healing.

The taxonomy can now be used as originally proposed. Consider the following three examples:

1. On the right, Outdated Results are caused by SLA or QoS faults, and in both cases can be in the Operational, Internal or Software categories.
2. If we have an unresponsive service, then there are six lines to follow, to Unavailability, Interface change, Workflow inconsistency, Timeout, Misbehaving flow or QoS. Checking the markers on the matrix, we see that the vote for these faults, the failure likely to be caused during Operational of the service by an External factor and it is a Software fault. It is unlikely to be a fault at Development time or in the Hardware, but this is not ruled out.
3. Slow service goes to five faults – Unavailability, Incorrect order, Time-out, SLA or QoS – but is much less easy to pin down in a broader category. All agree it is an Operational problem, but it could be Internal, External, Hardware or Software.

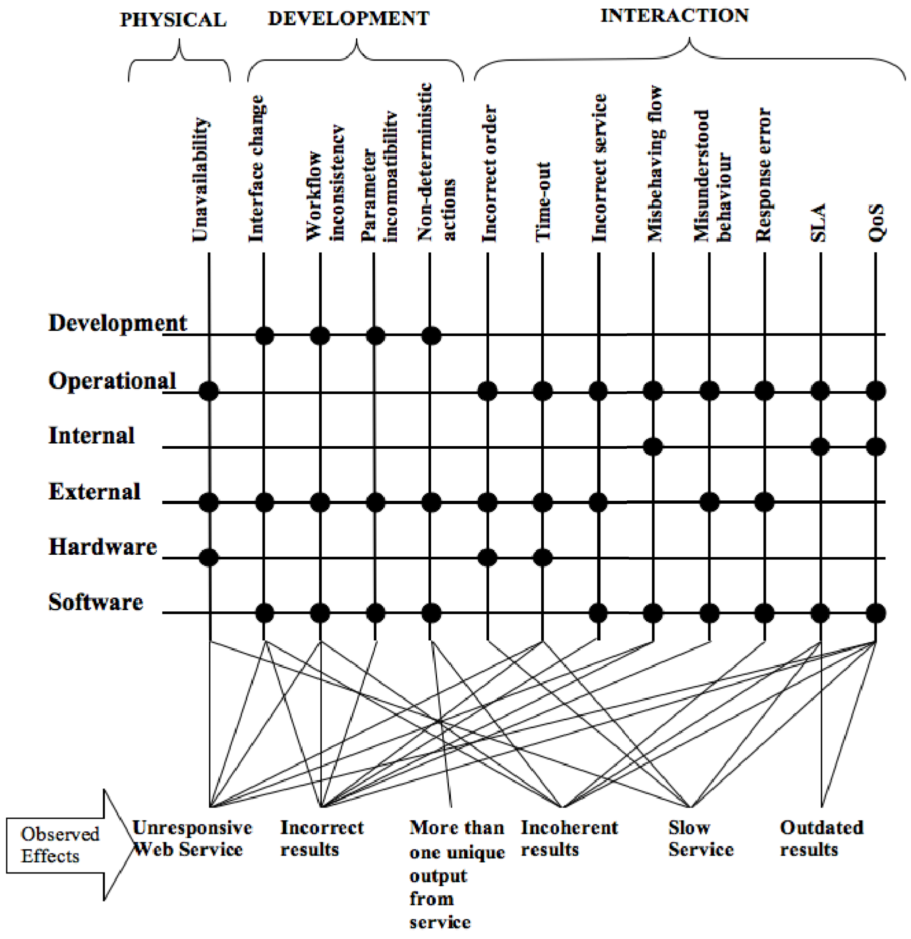


Fig. 6. Taxonomy of faults, combined with observed effects

The value of the matrix is that it excludes possibilities. Hardware faults are few, as are internal ones. However, External faults are frequent, as is to be expected in a Web service environment.

5 Related Work

We earlier proposed run-time monitoring and reactive strategies to ensure the correctness of the composition. Run-time monitoring acts as a probing phase that checks whether the composition behaves correctly, and it ensures functional and non-functional expectations are met. Reactive strategies are used to detect and recover from erroneous situations. Tartanoglu *et al.* [20] mentioned two error recovery methods: 1) forward error recovery, 2) roll-back mechanism (a transaction-based approach).

From a different approach, Fu *et al.* [7] propose a framework that analyses the interaction of composite Web services. The framework takes a Web service workflow specification (in BPEL) and translates it into an intermediate representation, followed by the translation of the intermediate representation into a verification language. This framework concentrates on verifying conversations. However, it lacks the ability to handle dynamic process instantiation and correlation sets. Ouyang *et al.* [16] use a similar method with their automated analysis tool, WofBPEL. Their tool translates the workflow specification into a different language before it can be analysed. The analysis is done in an off-line fashion though.

An alternative to the normal Web services description in WSDL is called Semantic Annotations or ontologies. Using ontologies to describe Web services has proven to be useful for various different tasks including discovery and composition. AI planning techniques have been used to address Web service composition problems described using ontologies [6]. From a Quality of Service (QoS) point of view Cardoso *et al.* [5] present a predictive QoS model that computes the QoS for workflows automatically based on atomic task QoS attributes. Yu and Lin [21] present a self-healing model that makes use of QoS constraints as a heuristic.

6 Conclusion

This paper proposed a fault taxonomy related to the Web service composition with a brief explanation of causes of failure and their consequences. Figure 6 presents a summary of the fault taxonomy proposed. Even though we pointed out some existing work that has the ability to automatically recognise and recover from failures, they are based on different techniques. In addition, the current available recovery mechanisms are far from complete. Investigation on fault representation in addition to recovery techniques and integrating them into one holistic recovery framework is underway.

The definition of a first taxonomy of faults is also interesting for the definition of "recovery patterns" that provide general-purpose and customizable solutions to recover from the different classes of faults. The final ambition is the blending of these classes with Dynamo [4] which is the infrastructure proposed by some of the authors to add recovery capabilities to BPEL processes. The framework proposes both a language to render the expressions in charge of discovering the different faults (failures) and also a language to encode complex and flexible reaction strategies. This means that for each class, we would like to define the checks needed to elicit the problem and how to change things to keep the composition on track.

We have only performed qualitative analysis on composition faults and quantitative analyses have not been considered. Our future work will include a complete breakdown of the probability of failure and its impact on the composition recovery mechanism. The impact of the failure measures the cost to the application when such a failure occurs. Once the probabilities and impact of failures have been identified, a recovery guideline can be developed to guide the recovery.

Acknowledgments. This work is funded by a South African-Italy Government Cooperation grant. Thanks to Stefan Gruner for his suggestions on an early draft of this paper.

References

1. Arkin A.: Business Process Modelling Language, `http://xml.coverpages.org/-BPML-2002.pdf`
2. Avižienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–32 (2004)
3. Baresi, L., Ghezzi, C., Guinea, S.: Towards Self-healing Compositions of Services. In: Krämer, B.J., Halang, W.A. (eds.) Contributions to Ubiquitous Computing. Studies in Computational Intellligence, vol. 42. Springer, Heidelberg (2006)
4. Baresi, L., Guinea, S.: Dynamo and Self-Healing BPEL Compositions. ICSE Companion, 69–70 (2007)
5. Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J., Kochut, K.: Quality of service for workflows and Web service processes. Web Semantics 1(3), 281–308 (2004)
6. Fan, X., Umapathy, K., Yen, J., Purao, S.: Team-based Agents for Proactive Failure Handling in Dynamic Composition of Web services. In: Proc. IEEE Intl. Conference on Web services, pp. 782–785 (2004)
7. Fu, X., Bultan, T., Su, J.: Analysis of Interacting BPEL Web services. In: Proc. 13th ACM Intl. Conf. on the World Wide Web, pp. 621–630 (2004)
8. Iribarne, L.: Web Components: A comparison between Web services and software components. Colombian Journal of Computation 5(1), 47–66 (2004)
9. Lamport, L.: Concurrent Reading and Writing of Clocks. ACM Trans. Comput. Syst. 8(4), 305–310 (1990)
10. Ludwig, H., Gimpel, H., Dan, A., Kearney, R.D.: Template-Based Automated Service Provisioning - Supporting the Agreement-Driven Service Life-Cycle. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 283–295. Springer, Heidelberg (2005)
11. Mariani, L.: A Fault Taxonomy for Component-Based Software. In: Proc. Intl. Workshop on Test and Analysis of Component-Based Systems (TACoS 2003) (April 2003). Electr. Notes Theor. Comput. Sci., vol. 82(6) (2003)
12. Martin, D. (ed.): OWL-S 1.1 Release, `http://www.daml.org/services/owl-s/1.1/`

13. Narayanan, S., McIlraith, S.A.: Simulation, Verification and Automated Composition of Web services. In: Proc. 11th ACM Intl. Conf. on the World Wide Web, pp. 77–88 (2002)
14. Oasis: Web Services Business Process Execution Language Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf
15. Oracle BPEL Process Manager Suite 10g, Oracle
16. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: WofBPEL: A tool for automated analysis of BPEL processes. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 484–489. Springer, Heidelberg (2005)
17. Singhal, M., Shivaratri, N.G.: Advanced concepts in operating systems: distributed, database, and multiprocess operating systems. McGraw-Hill, New York (1994)
18. Steyn J.: Approaches to Failure and Recovery in Service Composition, Technical Report, Polelo Research Group, University of Pretoria (2006), http://polelo.cs.up.ac.za/
19. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice-Hall, New Jersey (2002)
20. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N.: Dependability in the Web services Architecture, http://www-rocq.inria.fr/~tartanog/publi/wads/
21. Yu, T., Lin, K.: Service selection algorithms for composing complex services with multiple qoS constraints. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 130–143. Springer, Heidelberg (2005)