Electronic Notes in Theoretical Computer Science 82 No. 6 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 11 pages

A Fault Taxonomy for Component-based Software

Leonardo Mariani [1]

DISCO
Università degli Studi di Milano Bicocca
Milano, Italy

Abstract

Component technology is increasingly used to develop modular, configurable, and reusable systems. The problem of design and implement component-based systems is addressed by many models, methodologies, tools, and frameworks. On the contrary, analysis and test are not adequately supported yet. In general, a coherent fault taxonomy is a key starting point for providing techniques and methods for assessing the quality of software and in particular of component-based systems. This paper proposes a fault taxonomy to be used to develop and evaluate testing and analysis techniques for component-based software.

1 Introduction

Component technology is increasingly used to develop complex hardwqare and software systems, for enhancing composition, reuse, modularity and configurability. Component-based systems are developed by assembling existing components, thus the focus is shifted from module implementation to unit composition. Compoment-based technology boosts reuse because the design concerns mainly with linking existing components, and the implementation is relegated to the glue code. Component-based systems are modular because it is possible to substitute, add, or remove components after system deployment to obtain different configurations.

Customers demand for high quality systems that typically require a large amount of time to be developed and must be released in stable versions. Component technology can reduce time to market by increasing reuse of third-party

[1] Email: mariani@disco.unimib.it

[2] This work has been partially supported by the Italian Ministry of University and Research within the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems"

components, but the quality of the system becomes strongly dependent from the quality of externally developed code and the architecture of the systems. High quality components can be obtained by using a particular development process, by assuring a complete documentation, by executing tests and by analysing the final code. High quality components are necessary, but not sufficient to assure the quality of the derived systems. Thus, we need specific techniques for testing and analysis the final system, regardless of the quality of the used components. Classical approaches to analysis and test are not directly applicable to component software due to the many differences between classic and component-based systems.

To efficiently test any software system, it is important to know the possible failures and their causes. This paper proposes a first taxonomy for component-based systems that captures the many new types of faults that can arise in this kind of systems.

2 Software Components

Components have been defined in many ways. A commonly accepted definition has been given by Clemens Szyperski, who defines components as units of composition [12] with the following characteristics:

- with contractually specified interfaces
- with explicit context dependencies
- independently deployable
- and subject to third party composition

In referring to this definition, we stress that the most important characteristics of the component technology are:

- independent development
- and third party composition

Independent development means that the component is designed and implemented without knowledge about the systems and the components with which it will interact. The component developers will inevitably make same assumptions on both the environment and the type of interactions. Contractually specified interfaces and explicit context dependencies define explicitly developers' assumptions.

Third parties composition is a key feature strictly related to independent development: in general, developers do not know neither the environment in which the component will be used, nor the users of the components.

Component-based system development is supported by several technologies, e.g. EJB [7], CORBA [8] and .NET [3], but there are no unified frameworks addressing the problem to assure quality for this technologies. Existing testing techniques are not directly applicable, because in different systems,

software components may be used in ways that differ from the original design purpose; thus certification of the components may not be enough for a given system. For example, if a component has been largely tested, but features effectively used are lowly tested, the quality certification is misleading [11].

3 Fault Taxonomy

A component-based system is made up of interacting components that all together present the expected behavior. The testing activity is performed on single components (unit testing), sets of integrated components (integration test), or the whole system (system testing). A single component is similar to any other software product. It can presents traditional faults and can be tested with traditional techniques. Some techniques address the testing of single components aiming at gaining additional information for integration testing, but this approach is strictly related to testing and not to the type of faults. Other testing techniques address particular scenarios, e.g. regression test [6]. In our analysis we will not take into consideration possible scenarios except for some brief considerations relating them with errors.

The focus of this paper is on integration faults that characterize the component technology. We classify known faults according to their causes and effects. Causes are related eiher to the technology or to a particular scenario, e.g. system maintenance. Effects are the failures caused in the system.

We identify two main classes of faults: service-related and structure-related faults. Service-related faults can be syntactic, semantic, or non-functional. Structure-related faults, i.e., faults related to the structure of the system can derive from faulting connectors, the infrastructure, or the topology.

4 Syntactic Faults

The syntax of interractions defines the structure of the requests. Necessary condition to serve a request is that both the requesting and the serving components agree on the syntactic form of interaction. Agreement on the syntax is not enough to assert correctness of the interaction; components need to agree on the semantic and the protocol as well. In general, compilers assure the syntactic compatibility during the compilation phase, but if units are compiled separately and then combined at deployment-time or run-time this check cannot be performed.

For the independent development hypothesis, there is no full knowledge about components that will interact with the component under implementation. Often assumptions are made by constraining the *interfaces* implemented by other components, but in the final system these assumptions can be violated. This could happen simply because developers may be too optimistic with their assumptions, or because a system update changes the interface.

Syntactic interface faults take several forms and can be the cause of dif-

```
Class c = Class.forName("Car");
InterfaceCar vehicle = (Car)c.newInstance();
System.out.println(vehicle.capacityValue());
```

Fig. 1. A sample code that can fail when a component tries to bind to another component at start-up.

ferent failures depending on both the used component framework and binding mechanism. Figure 1 shows a sample code that can fail when a component tries to bind to another component at start-up. If the interface `InterfaceCar` does not match the interface implemented in `Car`, the second assignment cannot be executed. This type of syntax errors occurs frequently after updating the system, e.g., by updating a component. In other component frameworks, such as CORBA, the failure may be quite different, in fact the run-time binding mechanism works properly only if the searched and the retrieved interfaces match.

5 Semantical Faults

A semantical fault consists of a violation of explicit or implicit assumptions on the component behavior. The behavior can be specified with different formalism, e.g., with natural language or some kind of logic.

Differently from syntactic faults that cause the interruption of the interaction and, in absence of suitable exception handlers, the system crash, semantic faults lead to more subtile consequences. Semantic faults do not necessarily manifest with the impossibility of interacting, but may result in wrong results that manifest much later with a failure of another component.

There are several causes of semantic faults, e.g., wrong implementation or misbehaving execution flow. The compositional approach of the component-based systems increases the probability of semantic fault. Single components may have several dependencies that are stimulated to correctly perform their tasks, but the component developers may not know the identity of the components that will fulfill the request. In the same way, the component that will be used to satisfy a dependency may not be aware of this fact until deployment. Correctness of the interaction depends from the syntactic and semantic agreement of components.

We identify several possible semantical faults: misunderstood behavior fault, misunderstood parameter fault, misunderstood event generation fault, and misunderstood interaction protocol fault.

A *misunderstood behavior fault* is the case of a component that requests a service of another component, expecting a service different from the prvided one. A simple example is the case of a component `A` that requests for the arrangement of an array to another component `B` offering data manipulation services. A faulting behavior may be due to the fact that `A` assumes that

multiple copies of the same elements are put close to each other, but the component `B` deletes all multiple copies of an element. The final result of the interaction will be different from expected.

A *misunderstood parameter fault* is the case of a component `A` that wrongly interprets the arguments of a service. Referring back to the previous example, the component `B` could offer a printing service that prints the content of an array from the element of position 0 to the element of position $n-1$, where n is the parameter of the service. The component `A` may wrongly interpret the parameter and use the service as if the elements to be printed range from position 0 to position n. In this scenario a request for printing the whole content of an array issued with argument $n-1$ results in the component printing elements ranging from 0 to $n-2$ with unpredictable consequences.

A *misunderstood event generation fault* is the opposite situation of a misunderstood behavior fault: the component reacting to an event recognizes the event, but makes wrong assumptions on its meaning. For example, a component receiving an event `WindowMoved` may recognize the event, but not all the causes of the event. Generally, an event has an associated description, so there must be on agreement on the way to interpreter this description, if this agreement is not reached the event can be misinterpreted.

The last type of semantic fault is the *misunderstood interaction protocol fault*. Each component can be stimulated by many request sequences, but not all sequences are correct and not all sequences produce the expected behavior. To interact correctly, two components must agree on the used interaction protocol. In particular, a misunderstood interaction protocol fault consists on a correct flow of requests that produce a behavior different from the expected one. An example of this situation is the case of two components `A` and `B`, where component `B` provides services that enable the `HTML` presentation of `XML` documents by a `XSL` stylesheet. The component works by first defining a default `XSL` document and then by loading `XML` documents. The `XML` document is immediately transformed with the `XSL` stylesheet and then stored waiting for a `getHTML()`. When we need to change the presentation style, a new `XSL` document is set up. The component `B` is configured by the definition of a `XSL` document and behaves as a factory of `HTML` pages. If `A` assumes that the choice of a new style affects the loaded documents, it could use this flow of invocations:
`loadDocument()`→`setXSL()`→`getHTML()`

In the case the old `XSL` style is compatible with the loaded document, the `HTML` output is different from the desired one. The right sequence is
`setXSL()`→`loadDocument()`→`getHTML()`

Faults related to the semantic of an interaction have been addressed in several ways: design by contracts [9,4], formal specifications [10], and state machines (in the case of protocols) [2].

6 Non-Functional Faults

Like many systems, also component-based system must satisfy both functional and non-functional requirements. Often non-functional requirements propagates among components. This is the case, for example, of a component `A` that must produce a result within x seconds and needs a datum from a component `B` to complete the computation. Often, we do not have explicit information about non-functional properties of a component, because they are difficult to be formalized. Thus, guaranteeing non-functional properties in component-based systems is particularly hard. Scenario where components negotiate the quality of the service before interacting, further complicate this framework.

The main kinds of non-functional requirements concern interfaces (e.g. user-friendless and usability), performances (e.g. time and memory), quality of service (e.g., reliability), process (e.g. standard for the production of high quality components), and costs (e.g. bound on the costs). Here, we focus on performances and quality of service. In both cases the property of the system is affected by the performance and the quality of service of its components. In the case of performance, the dependencies are straightforward. In the case of quality of service, the situation can be very complex: a low-quality component may result in the impossibility of guaranteeing a required quality of service for the whole system. The interactions among components can be more subtile in the case of a machine that hosts more than one component, for example because two running components influences each other (simply because they share the CPU), or even the component framework can affect a single component, for example in the case the component framework broadcasts an event while a component is performing a computation the CPU could not serve the component efficiently.

7 Faulty Connectors

Components are linked by connectors, which can be faulty. A faulty connector may include a mismatching protocol fault or an incompatible data model [5]. Mismatching protocol faults derive from the absence of agreement on the used protocol. This case must not be confused with the component level protocol, in fact connector protocols are used to exchange messages in a single interaction, e.g. callback, while component protocol defines admissible sequence of invocation that can be issued to a components. Incompatible data model fauls are due to connectors that cannot transmit the desiderata data.

Examples of faulty connectors are provided by Garlan et al. in [5]. The mismatching protocol case is examplified by the impossibility to directly connect two components that make conflictual hypothesis about the protocol of the connector: callback versus client-server. The data model case is illustrated by the attempt to let interact the Softbench Broadcast Message Server and Mach RPC. The former expects ASCII stream while the latter expects `C`-based

data structures, interaction is possible only by implementing glue code.

8 Faults on the Infrastructure

The component execution and deployment is supported by the component infrastructure, if any, and by the overall underlying system. Obviously each component makes some assumptions about the nature, the available services and the structure of the system, but for the independent development hypothesis, it is not possible to predict the assumptions made by the other components. When a component-based system is developed, these assumptions may generate conflicts. Two components that unconsciously interact via the infrastructure are a potential source of failures. Problems of this type may have critical consequences, but manifest very rarely, and are thus difficult to diagnostic and remove. For example, two components may be in conflict because they use two different versions of the same dynamic library or they could interact through services of the operating system, e.g., the registry, or by accessing the same file, or by using the same service, or by sharing the disk. These interactions are difficult to be located and usually non mentioned in the set of dependencies, and even worse, they rarely manifest with a failure. Assumptions on the infrastructure are not only due to the adoption of a particular component, but are also consequence of the connectors and of any other feature used in the system.

9 Faults on the Topology

At deployment time, the component assembler defines the topology of a component-based system by linking components together. Hence, the component developer does not know how the component under development will be used. This scenario implies that any problem related to the topology of the system must be faced exclusively by the component assembler. On the other hand, the developer may facilitate the work of the assembler by following a design-for-testability approach [1] that consists on the adoption of a methodology facilitating the testing phase.

The topology of a system directly influences the execution control flow, in fact each link connecting two components describes a dependence and so it is a path that the control flow can traverse. The topology of a system can be defined at development time (static binding), at system start up by configuration files (dynamic binding), or at run-time (run-time binding) [12]. In the case of run-time bindings, the links defined during development and start-up of the system are used for moving references within the system. So if a component `A` is linked with a component `B` at design time and the component `B` is linked with the component `C` at system start-up, it is possible to use these two links to let a reference travel from `A` to `C`. When this operation has been accomplished `A` and `C` will be directly connected. Once established, run-time

references can be used to create further run-time bindings.

In this complex scenario several topology-dependent faults are possible: callbacks, re-entrance and inter-component recursion [12]. Callback fault is the case of a component `A` that suspends its execution and invokes a method of a component `B` that reacts invoking a method of `A`. During this second invocation, `A` is accessed in a critical status (its execution has been suspended) and its status can be inconsistent. The callback is a special case of re-entrance where an execution flow starts from a component `A`, traverses several components, and finally come back to `A`, which is in a critical status, as in the callback case.

An example of a re-entrance problem is the case shown in Figure 2 of a server managing the login of the users in a chat. The client requests to the server to login in the chat (1). The `Chan Server` lets the client enter the chat without voice (the client is logged in, but cannot send messages) and then checks the permissions (1.1). The `Permission Manager` component asks the `Data Manager` component for the credential of the client (1.1.1). If the client does not have permissions, the `Permission Manager` asks the `Chan Server` to send a message describing the error to the client (1.1.2); the `chechPermission()` concludes with a failure, and the user is kicked off from the channel. So far all works well. Let us now assume to update the `Chan Server` and change the login policy: now the user is not logged in without voice, but it is paused until the verification of the permissions is completed. This update causes the system to fail because the `sendMSG` cannot be performed because there is no valid receiver in the chat.

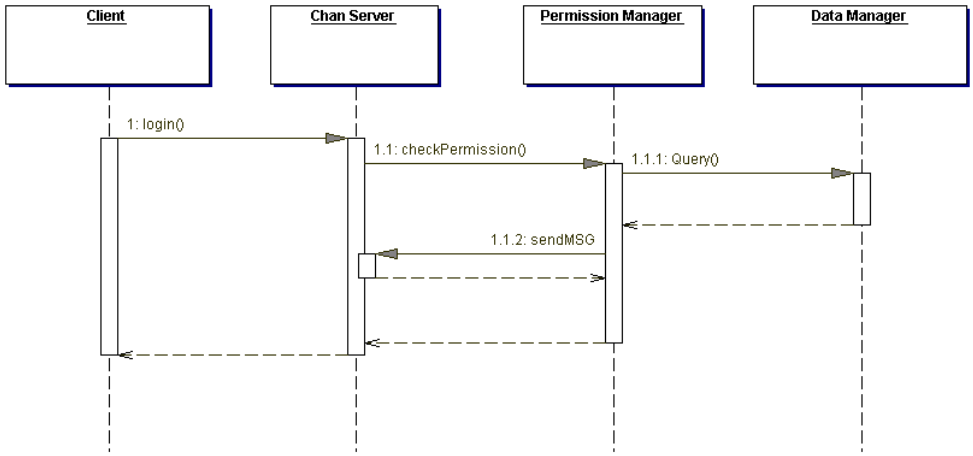

Fig. 2. An example of faults due to re-entrance

10 Other Faults

Component-based systems can present faults related to multi-threading. Traditional techniques address these faults using global knowledge about the system. In component-based systems we cannot use such approaches any more, but we need to make use of local knowledge only.

Components are heterogenous units and we integrate components developed with different programming languages. In some cases, this integration can lead to subtile faults. For example, `C++` arrays begin with index 0, while Visual Basic arrays begin with index 1, so components developed with these two languages may fail if they use an index of an element as the argument of a service.

Components are usually persistent, so serialization mechanisms, such as the Java serialization, are often involved. In this case, objects are stored into streams and then later restored. This procedure can introduce other kinds of faults, in fact static variables are not serialized (they need to be addressed singularly), code in the constructor is not executed during object restoration (but sometime at least some of the constructor's instructions need to be repeated during restoration), with an object are serialized all referred objects too (unwanted large stream may be created), and a token mechanism prevents from serializing two times the same object (but we must explicitly address changes on an object state). Currently, no existing testing technique effectively addresses these faults.

We already discussed re-entrance and control-flow faults. Such properties may lead to the generation of inconsistent events. This is a common problem of event-based system, but it becomes harder in component-based systems, since it must be addressed starting from local and incomplete information. For example, a component broadcasting messages can establish several links at run-time, i.e. by callbacks. If the component notifies the addition of files in a directory, the inconsistency is obtained simply considering the case that the first component that reacts to the "new file" event reacts deleting the file. The second component that receives the "new file" event will receive an inconsistent event. The same inconsistency happens in the case a component concurrently deletes the new file, while the broadcasting component is generating events.

Table 1 summarizes the taxonomy presented in this paper.

11 Conclusions

Component-based software systems are increasingly used in different application domains, and are supported by several development methodologies and tools, but there are still few approaches addressing analysis and test. The first step towards the development of effectives test and analysis method is the identification and classification of common faults.

This paper proposes a fault taxonomy related to the component technology

Main Category	**Sub-Categories**
Syntactic	• Interface Violation
Semantic	• Misunderstood on the Behavior • Misunderstood on Parameters • Misunderstood on Events • Misunderstood on the Interaction Protocol
Non-Functional	• Performances • Quality of Service
Connectors	• Disagreement on the Protocol • Quality of Service
Infrastructure	• Underlying Services • Underlying System
Topology	• Callback • Re-entrance • Recursion
Other	• Multi-thread • Heterogeneous Languages • Persistence • Inconsistent Events

Table 1
A summary of the error taxonomy presented in this paper

with a brief explanation of causes and consequences of faults and provides some examples. We plan to review the existing testing techniques according to the taxonomy proposed in this paper to identify open problems.

References

[1] Binder, R. V., *Design for testability in object-oriented systems*, Communications of the ACM **37** (1994), pp. 87–101.

[2] Buy, U., C. Ghezzi, A. Orso, M. Pezze and M. Valsasna, *A framework for testing object-oriented components*, in: *Proceedings of the First International ICSE Workshop Testing Distributed Component-Based Systems*, Los Angeles, California, 1999.

[3] ECMA, *Common language infrastructure (CLI) partition I: Concepts and architecture*, Final draft, Published by ECMA TC39/TG3 (2002).

[4] Edwards, S. H., *A framework for practical, automated black-box testing of component-based software*, Journal of Software Testing, Verification and Reliability **11** (2001).

[5] Garlan, D., R. Allen and J. Ockerbloom, *Architectural mismatch or why it is hard to build systems out of existing parts*, in: *Proceedings of the 17th International Conference on Software Engineering*, Seattle, WA, 1995.

[6] Harrold, M. J., *Testing evolving software*, Journal of Systems and Software **47** (1999), pp. 173–181.

[7] Matena, V. and M. Hapner, *Enterprise Javabeans^{TM} specification*, Technical report, Public Draft version 1.1, Sun Microsystems (1999).

[8] Merle, P., *Corba 3.0 new components chapters*, TC Document ptc/2001-11-03, Object Management Group (2001).

[9] Meyer, B., *Applying "design by contract"*, IEEE Computer **25** (1992), pp. 40–52.

[10] Necula, G., *Proof-carrying code*, in: *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, 1997.

[11] Rosenblum, D., *Challenges in exploiting architectural models for software testing*, in: *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, 1998.

[12] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," ACM Press and Addison-Wesley, New York, NY, 1998.