Architecture-based Strategy for Interface Fault Injection

Regina Lúcia de Oliveira Moraes
regina@ceset.unicamp.br
+55 19 3788-5872
Superior Center of Technology Education (CESET)

Eliane Martins
eliane@ic.unicamp.br
+55 19 3404-7165
Institute of Computing (IC)

State University of Campinas (UNICAMP)

Abstract

This paper presents a strategy that is an adaptation and an evolution of a previous one proposed to validate an isolated component. Faults are injected using a previously developed tool, Jaca that has the ability to inject faults into Java objects' attributes and methods. One of the key issues in component-based systems is its architecture, not only for development but also for testing. By analyzing the architecture we can define the points of control and observation of the system's components during testing. Another important issue is the selection of components to be injected and monitored. A risk-based strategy is proposed, in order to prioritize the components for testing which represent higher risks for the system. In this way, test costs can be reduced without undermining the system's quality.

1. Introduction

1.1. Motivation

Increasingly systems are being developed as a composition of several components; they can be developed in house (by the same team or by another) or by third-party. The use of components helps to attend the increasing pressures on reduced time and money, but it can introduce new problems, such as architectural mismatch [8] that can arise when the expectation of a component do not match those of other components or the environment in which they are operate. This increases the importance of the systems' architecture: the system's quality is increased when the used architecture is a good solution for the system. However the success of the implementation of architecture depends on two aspects [19]: i) that each component's implementation behaves in accordance to its specification, and ii) that components interact adequately.

This paper addresses the second point mentioned above. Specifically, our interest is to determine whether a malfunctioning in the interaction among components can compromise the overall system quality. Fault injection is used for that purpose.

1.2. Fault Injection

Fault injection has been widely used to evaluate the dependability of a system and to validate error-handling mechanisms. This technique consists of introducing faults during an execution of the system under test and then observing its behavior. By doing so, it is possible to know how the system will behave in the presence of faults in its components or in its environment.

Fault injection approaches vary according to the system's life cycle where they are applied, and the type of faults that are injected. Among the various existing approaches (see [9]), software fault injection is getting more popular. In this approach, logical faults are introduced in a prototype of the system and specific error conditions are injected to simulate software faults (internal, e.g., variable/ parameters that are not initialized among others), as well as faults that occur in external components that interact with the tested application (all external factors that alter the system's state)[20].

Fault injection can be a valuable approach in component-based development, not only to validate components in isolation but also to validate their integration into a system. By introducing faults or errors in different components of a system (in-house or third-party components), it is useful to answer questions such as: does a component fail when receiving invalid inputs from other components or from the environment? Does a failure in a component cause the whole system to fail?

1.3. Jaca - Fault Injection Tool

Our approach is based on the introduction of interface faults, in which faults are introduced at a component's interfaces by affecting input or output parameters as well as returned results. A software fault injection tool, Jaca [12], was used to introduce the faults, aiming at validating Java applications. Jaca does not require access to an application's source code, though it is a solution for the validation of a system composed of multiple COTS (Commercial Off The Shelf) components, which are generally black box. All instrumentation needed for fault injection and monitoring purposes are introduced at byte code level during load time. However, a minimum controllability (ease in controlling a component's inputs and outputs) and observability (the ability to observe a component's inputs, outputs and operational behavior) is required.

In order to inject faults using Jaca the *controllability* can be achieved when the tester knows the public methods' signature and may observes the tests' results and exceptions raised through Jaca's interface. Nevertheless, when the architecture is composed by third-party and in-house developed codes, specially parts that "glue" components together, the tester may use these

units to improve the system's *controllability* and *observability* by taking them as control and observation points during the process of fault injection.

1.4. Objectives of This Work

The goal of this work is to propose a fault injection strategy to test component-based systems. Our interest is in the interaction among components. For that reason, we introduce interface faults, by corrupting input data as well as interface output data.

Interface fault injection is useful in that it is at the interfaces that corrupt data come into a component. Components' internal faults can reach their interfaces, the errors' propagation occurs through the interfaces, and in the validation of component-based software this technique can be the only way to inject faults.

In Section II we discuss related works. Section III presents our proposed strategy. In section IV we point out some difficulties of the strategy and Section V presents the contributions of this work. Finally, in section VI we outline future research.

2. Related Works

This work is an evolution of that presented in [13] where we tested an isolated component. In that work we used an application to activate the component under test. The differences between them are twofold: (i) the target is no longer a component in isolation, but a system integrating various heterogeneous components, where some of them may be black-box; (ii) the units considered for fault injection purposes are components rather than classes. Thus, the strategy cannot use source code dependent metrics as in [13] and the architecture becomes essential for planning fault injection.

A closely related approach to the one presented here is the Interface Propagation Analysis (IPA) [24, ch.9.2]. IPA takes a black-box view of software components, injecting faults at the interfaces between the hardware and the software as well as between the operating system, microkernel and so on. The difference is that we are considering that not all components are black box.

The steps described in our work were strongly influenced by those described in [4], where the generation of error sets was based on field data. The difference in our case is that we considered field data to be unavailable.

TAMER [6] is another work that describes a tool that injects interface faults aiming to observe fault propagation. The main focus of that work is code coverage. We are not interested in source code coverage, but rather in the exceptions raised by the component, and whether these exceptions cause the whole system to fail.

The work in [7] is quite similar to ours since they use a tool based on computational reflection, called Java Wrapper Generator (JWG), which modifies the bytecode at load time, in order to provoke an exception and observe the behavior of the exception handlers. In their case the focus is on objects, whereas in our approach the focus is on components, which may be composed by several classes.

From the Ballista [10] approach we utilized the definition of the error model that was proposed by the authors for robustness testing.

We also borrowed ideas from studies that use risk for test costs reduction. Many risk-based testing strategies have been proposed [2], [17]. The approach presented here is specially related to [2], from which we use the heuristic risk-based testing presented below.

3. The Approach

Fault injection experiments can be characterized by the FARM model [1], where F designates the set of faults/errors to be injected, A the activation mode of the system, R is the set of data collected during the experiments and M the verdict of whether or not the system behaves as specified, when the fault injection goal is to reveal design and/or implementation faults.

The approach used to validate a component-based system using fault injection is quite similar to that proposed in [20, ch.9.2], where faults are injected at the interfaces between components to simulate the situation where a component fails and its outputs corrupted information to the others components System robustness is determined by checking post-conditions at specific points as well as at the system-level interface.

The remainder of this section presents an approach that helps to determine the F set by answering: which components should be injected? How should they be injected? What fault model should be used? and When should they be injected?

3.1. Architectural View

The software architecture of a system represents the software structures that form the skeleton of the application. It defines how the system is structured in terms of the components that form the system, assigns the responsibilities to the components, defines the interactions among them and assures that the component's interactions satisfy the system requirements [5][18]. An important issue is the definition of an interconnection mechanism for gluing the pieces together, the connectors [8][19]. Thus a system architecture is defined as a collection of components, which is a unit of software that performs a function at run-time and a collection of connectors, which is a unit of software that "glues" components together and mediates the interactions (communication, coordination, or cooperation) among these components [8]. Through the connectors' format conversions, two incompatible components can share data as well as connectors augmented by performance and behavior monitoring, authentication and audit-trail capabilities [18]. Although

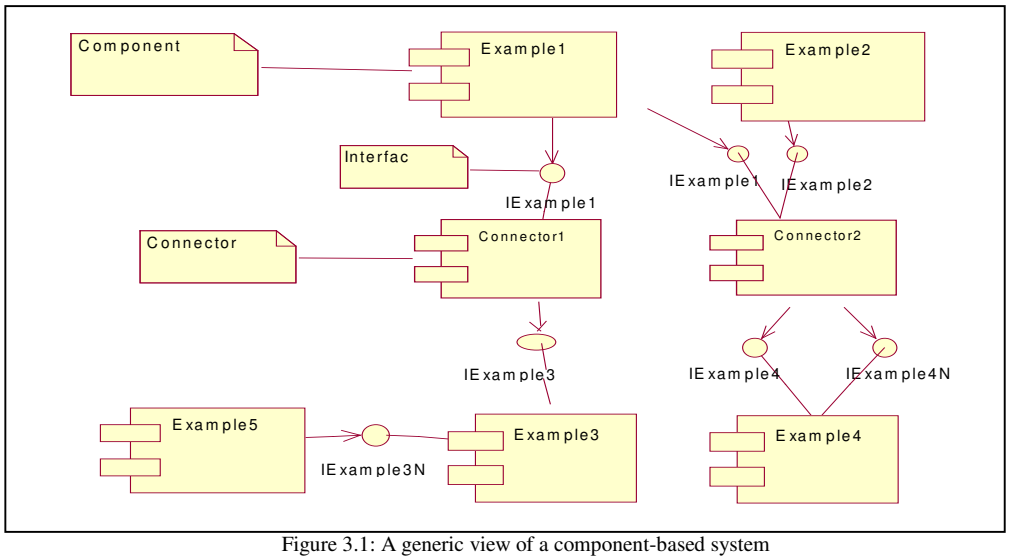

Figure 3.1: A generic view of a component-based system

the system may be composed of COTS components, from which no source code is available, the system is not considered a black-box: the system's architecture, which represents how the various components are interconnected, is known. Figure 3.1 shows, using UML notation and the system's architecture as viewed for fault injection purposes.

Components can provide one or more interfaces, through which they specify their services. They can also require interfaces that describe the services they need in order to provide theirs own interfaces [5][14]. In Figure 3.1 the Example4 component offers its services through the interface designated as IExample4 and IExample4N. The Example5 component requires services specified by interface IExample3N.

Connectors can also provide services, such as persistence, invocation and transactions, which are independent of the interacting component's functionality [19] and are responsible for the association between a required interface and one or more provided interfaces. This association is called interface connection [8] [19].

As shown in Figure 3.1, connectors link the interface required by a component to an interface provided by another component (shown as "lollipops in the UML notation for interfaces), but these interfaces can interact directly with each other without the mediation of a connector (interaction between Example3 and Example5).

3.2. Generation of the F Set

Inspired by the work of [4] we defined the following steps to determine the F set:

1. Prioritize the components
2. Select the components that should be injected
3. Select the operations of each selected component
4. Generate a list of injection points within each operation
5. Define error model
6. Decide temporal characterization of the faults

Steps 1 and 2 are not injection tool dependent, all other steps are. To answer the question "Which components to inject?" we defined steps 1 and 2. The question "How to inject?" can be answered in steps 3 and 4. "What fault model to use?" can be answered in step 5 and finally "When to inject?" in step 6.

3.2.1. Components prioritization

Given a system of multiple components, what components should be selected, when it is not possible to inject all of them? An approach could be to select the components based on information about the distribution of faults over the component observed either during development or in the field [4]. If such information is not available, a risk-based testing strategy can be applied. The idea is to allocate test effort in code parts that are most error prone, and where failure would have the highest impact [17].

There are several techniques to assess risk (see [11], [17]). We use a heuristic method, as proposed in [2], where the following criteria are used to assess a components' risk: New (freshly developed), Changed (has been modified), Upstream Dependency (failure in it will cause cascading failure in the rest of the system) Downstream Dependency (it is especially sensitive to failures in the rest of the system), Critical (failure in it could cause substantial damage), Popular (will be used a lot), Strategic (has special importance to business-feature that set apart from the competition), Third-party (developed outside the project), Distributed (spread out in time or space, yet whose elements must work together). To these criteria we add one more: Understandable (how much information about the component is provided and how it is presented).

With these criteria, a component risk matrix can be constructed, as shown in Table 3.1. If a check is placed

Table 3.1: Summary table of component risks

Component	New	Changed	Upstream Dep	Downstream Dep	Critical	Popular	Strategic	Thid-Party	Distributed	Not Understandable	Risk
Component1		✓			✓	✓	✓	✓		✓	High
Component2			✓				✓				Low
Component3					✓			✓		✓	Medium
Component4	✓			✓		✓	✓	✓	✓	✓	High

in a column, it means that this criterion is significant for that component. Although we can count the number of checks placed for a component, the risk judgments cannot be considered with this simplicity. It is possible to have a situation in which a component has less heuristics checked than another but it is considered more risky than the latter. These heuristics have to be seen as tools for assessing risk not for determining it. They do not substitute an expert's opinions [14].

3.2.2. Selection of the Components

Given the high risk components obtained in step 1, we continue to select those components that can be injected using Jaca. Two more criteria are used:

- *Controllability*: which shows how easy it is to inject faults on a component's inputs/outputs.
- *Observability*: which regards the ease with which a component can be monitored in terms of its operational behaviors, input parameters, and outputs.

When Jaca is used, the injectors and sensors needed for injection and monitoring purposes during runtime are inserted at bytecode level when the system is being loaded. This is achieved through the use of the Javassist toolkit[3], which extends the Java Reflective Interface.

If a selected component has low *controllability* and *observability*, the user has the following options: (i) to inject faults into the component's predecessors and successors developed in-house which have the desired *controllability* and *observability* (ii) to inject faults into the user-developed connectors (connectors can be taken as injection points or as observer points, not both, as the results could be influenced by the faults injected).

3.2.3. Selection of the Operations

The components' operations are distributed among the components' interfaces. Once a component has been chosen, faults are injected in the component's interfaces indistinctly, considering all of their operations as a whole. To select the operations in which parameters will be affected during fault injection, we can use a technique similar to partition testing, as presented in [16, ch.22.5], where each operation can be categorized, for example, as Initialization, Computational, Queries and Termination operations. We can also use a state-based partitioning that categorizes operations according to their ability to change the state of the component. Faults are selected so that they affect operations in each category at least once. Faults are distributed uniformly among the categories.

3.2.4. Generation of the List of Injection Points

Once the operations that are to be injected are selected, the next steps are to determine the parameters to inject, and what error model to apply.

In what concerns the parameter selection, we have to cope with Jaca's current limitations: only non-structured values can be affected (integer, real, and Boolean). The only structured type that can be affected is string [13].

3.2.5. Selection of the Error Model

The error model is based on Ballista's [10], alongside the one proposed by [20]. According to these two approaches the values to be used for each type are:

- Integer / long: 0, 1, -1, MinInt, MaxInt, neighbor value (current value ± 1)
- Real floating point: 0, 1, -1, DBLMin, DBLMax, neighbor value (current value*0.95 or *1.05)
- Boolean: inversion of state (true -> false; false -> true)
- String: null, largest string, string with all ASCII, string with pernicious file modes and printf format which can be composed by conflicting characters.

These values potentially represent exceptional test values for each data type.

3.2.6. Temporal Characterization

The temporal characterization of faults is strongly related to the mechanisms used to trigger fault injection. In Jaca, faults are triggered when the target operation is accessed.

Faults can be injected either permanently (each time a target operation is accessed), transiently (where faults are injected only once), or intermittently (injected repeatedly according to a pre-specified frequency, established in terms of the number of accesses to an operation).

4. Difficulties of the strategy

We identified some difficulties, such as:

(i) How many criteria must be satisfied?
(ii) How should these criteria be weighted?
(iii) How should each factor be quantified?

This should be based on experts' experiences as suggested by [14]. Experts can comprise developers, key users, customers and test personnel, who should brainstorm the risks associated with the specific software system [15]. Some factors are more difficult to quantify

than others, such as, popular, critical, upstream /downstream dependence, understandable.

(iv) How should the risk of a component be determined?

Risks are normally a quantitative value. To obtain this final value, it is necessary to rank risks for each factor, i.e. in the range 1-3 rank low to high (relational severity indicator, one-third for each degree). The severity associated with each factor is also application-domain-dependent and must be established by experts. It can be determined subjectively, as in [15]. It can also be determined according to safety analysis classification, as in [11]. . The risk of the component should be the sum of all partition's risk[20].

(v) How should successors and predecessors be determined? Following the determination of upstream/downstream dependency?

Based on [20], predecessors are components upon which the target component depends for input information and conversely, successors are those components to which the target component sends information. Thus, the predecessor can send corrupted data for the component under test and the latter can send corrupted data to its successors. Static analyses can help in determining successors and predecessors of a component[15].

(vi) What error model should be selected?

We combine the partition testing model with the error model based on Ballista's robustness testing. However, the later considers only the input space, which, as determined by boundary-value analysis, is not enough. The output space should also be considered. Which inputs may cause the component or the system to produce the wrong output? Using Fault Tree Analysis we can map the wrong output to the potential inputs.

(vii) How can good *controllability* and *observability* of the system's components be achieved?

Jaca is designed for interface fault injection, so, only externally visible operations are considered. Nonetheless, there are still some limitations to overcome, such as injection and monitoring of non-scalar types; insertion of post-condition and invariant checkers to indicate failure occurrence; and tracking of exceptions and activation of exception handlers. We are proposing the use of connectors to achieve the desired *controllability* and *observability*.

5. Contributions of Our Work

This work focus on the systems' architecture, in particular the connectors, for improving the controllability and observability that is necessary for fault injection tests.

What is being proposed is a systematic way to perform fault injection to characterize the behavior of components and systems in the presence of faults. This also contributes to dependability benchmark, in that it provides a uniform and repeatable way to perform fault injection.

6. Future Research

Our short-term goals are (i) define how safety techniques (methods based on FMEA, FMECA, FTA) can help to better define the selection of the injections points; (ii) the application of the strategy in a real world application aiming to assess the value of using the architecture to guide fault injection (we will perform experiments comparing the results obtained when fault injection points are randomly chosen among the components).

References:

[1] Arlat, J.; Aguera, M.; Amat, L.; Crouzet, Y.; Fabre, J. C.; Laprie, J. C.; Martins, E.; Powell, D. "Fault Injection for Dependability Validation–A Methodology and some Applications". IEEE Transactions on Software Engineering, 16 (2), Feb/1990, pag 166-182.

[2] Bach J. "Heuristic Risk-Based Testing", Software Testing and Quality Engineering Magazine, November 1999

[3] Chiba, Shigeru. "Javassist – A Reflection-based Programming Wizard for Java", proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java, Oct/1998.

[4] Christmansson, J ; Chillarege, R. "Generation of an Error Set that Emulates Software Faults-Based on Fields Data", 26th Int Syposium on Fault-Tolerant Computing, pp 304-13, Sendai, Japan Jun/1996.

[5] Cheesman, J; Daniels, J "UML Components – A Simple Process for Specifying Component-Based Software", The Component Software Series, Addison-Wesley, 2001.

[6] De Millo, R. A.; Li, T.; Mathur, A. P. "Architecture or TAMER: A Tool for dependability analysis of distributed fault-tolerant systems", Purdue University, 1994.

[7] Fetzer, C.; Högstedt, K.; Felber, P. "Automatic Detection and Masking of Non-Atomic Exception Handling", proceedings of DSN 2003, pages 445/454, San Francisco, USA, June/2003.

[8] Garlan, D. ; Allen, R.; Ockerbloom, J. "Ärchitecture Mismatche, or, why it's hard to build systems out of existing parts", proc. of the 17th ICSE, April/95.

[9] Hsueh, M.C; Tsai, T.; Iyer, R.. "Fault Injection Techniques and Tools". IEEE Computer, Abril/1997.

[10] Koopman, P.; Siewiorek, D.; DeVale, K.; DeVale, J.; Fernsler, K.; Guttendorf, D.; Kropp, N.; Pan, J.; Shelton, C.; Shi, Y. "Ballista Project : COTS Software Robustness Testing", Carnegie Mellon University, http://www.ece.cmu.edu/~koopman/ballista/, 2003.

[11] Levenson, N.G. "Safeware, System Safety and Computers" Addison-Wesley Publishing Company, 1995.

[12] Martins, E.; Rubira, C. M. F.,Leme, N. G. M., "Jaca: A reflective fault injection tool based on patterns", proceeding of the IPDS, 2002.

[13] Moraes, R; Martins, E "A Strategy for Validating an ODBMS Component Using a High-Level Software Fault Injection Tool", proc. of the First Latin-American Symposium, pages 56-68, SP, Brazil, 2003.

[14] Peters, J. F.; Pedrycz, W. "An Engineering Approach", John Wiley & Sons Inc, 2000.

[15] Perry, W. "Effective Methods for Software Testing", John Wiley & Sons, New York, 1995.

[16] Pressman, R. S. "Software Engineering a Practitioner Approach", 4th edition, Mc Graw Hill , 1997.

[17] Rosenberg, L; Stapko, R; Gallo, A "Risk-based Object Oriented Testing ", 13th International Software / Internet Quality Week (QW2000), San Francisco, California USA, 2000.

[18] Shaw, M.; Clements, P. "Toward Boxology: Preliminary Classification of Architectural Styles", proceedings of SIGSOFT 96 Workshop, San Francisco, CA, USA, 1996.

[19] Silva, M. C.Jr.; Guerra, P. A. C.; Rubira, C. M.F. "A Java Component Model for Evolving Software Systems", proc. of Automated Software, Engineering, Canada, 2003.

[20] Voas, J.; McGraw, G.. "Software Fault Injection: Inoculating Programs against Errors", John Wiley & Sons, NY, EUA, 1998.