ROBUSTNESS TESTING TECHNIQUES FOR HIGH AVAILABILITY MIDDLEWARE SOLUTIONS

ZOLTÁN MICSKEI, ISTVÁN MAJZIK

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
{micskeiz, majzik}@mit.bme.hu

FRANCIS TAM

Nokia Research Center, Nokia Group, Finland
francis.tam@nokia.com

To increase the interoperability of availability management software (also known as high availability middleware) the Service Availability Forum has released a set of open specifications. With the development of a common interface the comparison of multiple products can be achieved. For high availability (HA) solutions, assessing the robustness of the HA middleware is as important as measuring its performance. This paper investigates the sources of inputs that can activate robustness faults of a HA middleware and recommends the corresponding testing techniques to check the existence of such faults. We investigated the automated construction of the robustness test suites and compared the efficiency of different techniques using a case study with an open-source HA middleware.

1. Introduction

In the past few years dependability became a key attribute even in common computing platforms. High availability (HA) can be achieved by introducing *redundancy* in the system, like warm standby spares, redundant communication channels etc. The configuration of the redundant components, thus the management of the availability of the whole system, is often application independent. The necessary services (e.g. membership, recovery) can be implemented as a generic middleware.

To increase the interoperability of availability management software (known as HA middleware) major users and vendors formed a consortium, the Service Availability Forum with the goal to develop open specifications for availability management of software and the underlying hardware. SA Forum's Application Interface Specification (AIS) [1] defines the interface between the HA middleware and the custom application. It is a C language interface partitioned into a number of services. For example, the Cluster Membership Service (CLM) provides a consistent view of the computing nodes, while the Availability Management Framework (AMF) manages the availability of redundant components. Three major versions have been released for AIS so far, the latest being B.02.01. There are several implementations available for AIS; we used in our experiments an open-source middleware, OpenAIS [2] (alpha release, version 0.69).

Having a common specification for the HA middleware products, the demand to compare the various implementations naturally arises. Most of the comparisons and benchmarks of similar middleware products address performance, but in case of a HA middleware, the *robustness of the implementation* is also a crucial attribute. Robustness failures in the middleware can be activated by poor quality application components, and one such component may render the whole application inaccessible. Thus, our long-term goal is to define a method to evaluate and compare the robustness of different AIS based HA middleware implementations.

Robustness is a secondary attribute of dependability and it is used in this paper as defined in [3], i.e., the degree to which a system operates correctly in the presence of *exceptional inputs* or *stressful environmental conditions*. Accordingly, *robustness faults* are those faults that can be activated by these inputs and conditions, resulting in an incorrect operation (e.g. crash, deadlock) of the system.

Although there is an open-source implementation of AIS, most of the implementations are (and will be) commercial products with limited information about their internal structure. Without a detailed behavioral model or source code, the evaluation can only be based on the common interface specification. Accordingly, the services (functions) defined by the AIS can be tested for robustness faults externally by executing specific test sequences called *robustness tests*. The approach of robustness testing is similar to functional "black box" testing, but it concentrates on the activation of potential robustness faults by providing exceptional inputs and generating stressful conditions. Thus the basis of the comparison of AIS implementations is the common interface specification, as the number of robustness faults per functions is measured.

Robustness testing is a time and resource consuming activity. Generating an effective test suite, executing it and evaluating the results usually needs a lot of manual work. In a model-based development process test construction and test execution can be partially automated. AIS provides a semi-formal description of the interfaces, which can be used to gather the possible inputs and output acceptance conditions, and thus it allows automated test construction and test execution. Moreover, this interface specification can be utilized to construct more sophisticated test sequences than the commonly used ones (based on input variable domains only). Accordingly, in this paper we focus on the following aspects of robustness testing:

- *Automated construction of robustness test suites for AIS based HA middleware.* The exceptional input values are generated by automated tools on the basis of the functional specification.
- *Elaboration of extended robustness testing techniques.* Scenario-based robustness testing techniques are proposed to cover non-trivial robustness faults in state-based functions of the AIS. In the case of these functions a specific call sequence is required to reach the state in which the service can be used, otherwise a trivial error code is returned without executing the service and thus activating the potential robustness faults.
- *Evaluation of the test results using intelligent data processing techniques.* On-line analytical processing and basic data mining methods are proposed to identify the key factors (e.g. product version, OS version, workload) that influence robustness.

In the paper Section 2 summarizes the previous robustness testing projects. In Section 3 the concepts of our robustness testing framework for AIS-based HA middleware are presented. The different robustness testing techniques are described in Section 4 and 5. The efficiency of the techniques is compared in Section 6. Finally, Section 7 concludes our results and lists future plans.

2. Related work

Robustness testing was the goal of several research projects in the past. Different methods were applied to measure the dependability of complex systems at various abstraction levels.

Fuzz [4] was one of the first tools designed especially for robustness testing. It utilized randomly generated character strings to test common UNIX console utilities. This simple method found for 20% of the tested 80 applications an input sequence that crashed the program.

The Riddle tool [5] was used to test the operating system API in Windows NT. Two techniques were applied for input generation. The *generic technique* used a fixed input domain for all parameters of the API while the so called *intelligent one* used a specific generator for each type. The tests found abort failures in 10% to 80% of the functions in three system DLLs. The four-year Ballista project [6] assessed the robustness of POSIX API implementations and conducted a great number of experiments on 14 UNIX versions. The robustness test suite, which also applied type-specific input generators, was used mainly for comparing the different UNIX products. Later the method was extended for CORBA, Windows and for a simulation backplane testing.

The goal of the recent *dependability benchmarking projects* was slightly different; they defined benchmarks to characterize the system behavior under typical load and common fault conditions. A general framework for creating dependability benchmarks was developed in the EU project DBench [7]. The method was implemented e.g. for operating systems [8]. Software and hardware vendors are also providing availability benchmarks for their products, e.g. IBM for autonomic computing [9] and Sun for the R-cubed framework [10].

In our work we tried to integrate the complementary solutions for robustness testing and extend them with advanced methods specific to HA middleware.

3. The AIS robustness testing framework

The first step of defining the testing strategy in the case of a "black box" AIS middleware is to identify the possible sources of inputs that can activate robustness faults. These inputs are depicted in Figure 1(a). In the following the potential robustness faults are grouped on the basis of the source of activation, defining in this way the *type* of the fault. For each fault type a testing technique was selected as shown on Figure 1(b):

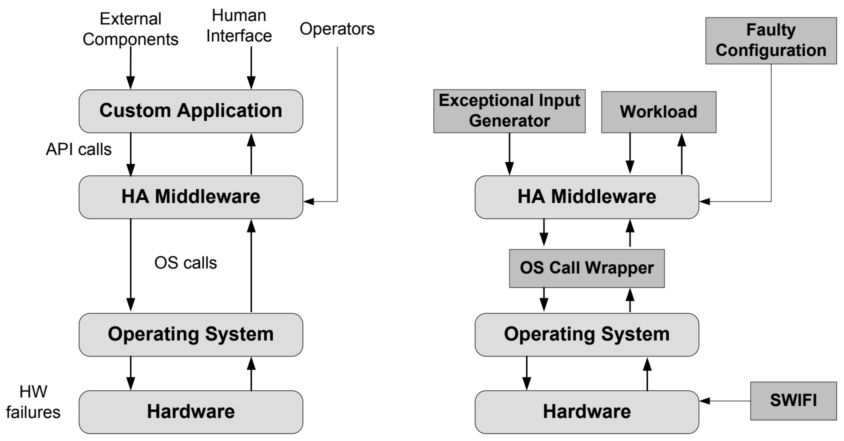

Figure 1. Inputs that can activate robustness faults in a HA middleware (a) and the proposed robustness testing techniques (b)

- The calls from the custom application (which propagate the effects of human operators and external components as well) are provided by an *exceptional input generator* and a background *workload*. The workload represents the typical operation of the system. Note that in case of a HA middleware it should include failovers, administrative restarts and other fault-masking activities, because they are part of the normal operation.

- Configuration inputs (given by system operators) are represented by providing *faulty configurations*.
- Exceptional results of operating system (OS) calls are given by an *OS call wrapper* that catches the return values of OS calls, injects the exceptional return values defined by the *faultload* and provides observability. The faultload defines the type and timing of the injected faults. Note that simulating failures of operating system calls has two purposes. It checks the reaction of the HA middleware not only in case of a fault in the OS itself (which has quite low probability), but also in the case of many other failures in the environment (e.g. wrong file access settings, insufficient resources) that are manifested in exceptional results from OS calls.
- Hardware level faultload is provided by *software implemented fault injection* (SWIFI).

These techniques can be executed in two distinct phases of testing: (1) testing the API with a robustness test suite containing exceptional inputs, (2) workload based benchmarking with injected faults representing stressful environmental conditions.

In our paper we focus on the first phase of testing. The selected techniques are supported by the following set of tools that allow an automated construction of test suites containing exceptional inputs:

- *Template-based type-specific test generator* tool. Templates specify type and function information on the basis of the AIS API and the tool generates the test programs automatically (see in Section 4).
- *Scenario-based sequential test generator* tool. To construct test sequences needed to test state-based API functions, besides the AIS specification the *functional test sequences* provided by the vendors of the HA middleware were also utilized. The tool will process these sequences, and executes (1) parts of them to reach specific states in which type-specific test inputs can be used and also (2) applies mutation operators (e.g. changing the sequence of tests, modifying parameters or function names) to construct exceptional sequences (see in Section 5).

In the second phase of testing the stressful environmental conditions can be provided by implementing a *workload* with a *faultload* (as in other previous dependability benchmarks). The following tools are proposed:

- *Faulty configuration generator* tool. The administrative interface of AMF was introduced recently (January 2006), but the configuration was not standardized, in this way this tool could not be realized. As soon as products will support this specification, the (mutation-based) administrative actions and faulty configurations can be generated and executed.
- *OS call wrapper*. OS level errors are injected by a wrapper between the OS and the middleware, like in [11]. A lightweight wrapper can be implemented on Linux with the LD_PRELOAD environmental variable, which can be used to reroute the system calls to modified libraries.
- *SWIFI tool*. Besides explicit component failures, like abrupt node shutdown or network interface failure, lower level hardware faults can be injected by external tools like FAUmachine (formerly UMLinux [12]).

One of the most labor-intensive part of robustness testing is the evaluation of the test outcome. Functional test cases usually contain the expected result and compare the actual result to this reference value. In case of robustness testing there is a widely accepted *simplified approach*. Obvious robustness failures, i.e. crash and abort-like answers are recognized. However, no differentiation is made between the other possible results, i.e., successful answer, valid error code according to the specification, misleading error code and silent errors. (This simplification was necessary in most systems to reduce testing costs and avoid the problems originating in missing or incomplete specification, especially in case of erroneous behavior.) We refined this method by assigning the possible error codes (as potential results) to test inputs values. The test outputs are then filtered and only those test runs are inspected, in which the output was not among the expected error codes.

Even in our early tests thousands of robustness test cases were generated, thus an automated method was needed to analyze the results. Two previously recommended techniques were used to accomplish this. Online Analytical Processing (OLAP) was applied to filter and compare the results of different systems [13] and data mining to identify the possible fault sources [14].

In the following sections we describe the implemented tools and techniques of API testing.

4. Generic and type-specific testing

Exceptional inputs can be grouped into the following categories:

- *Syntactically not correct values*: e.g. invalid string for an IPv4 address.
- *Semantically not correct values*: e.g. non-existing version number.
- *Values used in invalid context*: e.g. not initialized handle.

The majority of types used in API functions is defined by complex structures. Constructing exceptional values from all possible combinations of the *basic types* in these structures, like int, char, etc., would result in far too many values, because many structures are built from more than four basic types and the AIS functions have on average two or three parameters. Thus, finding good exceptional values is not as obvious as for example in case of the basic types like integers. The following subsections describe the two techniques that were used for generation of exceptional inputs for individual API calls.

4.1. Generic input testing

In the case of *generic input testing* the same set of values are used for all parameters of basic types. In C language, most of the basic types can be represented and cast to a four-byte number, as Listing (1) illustrates.

```
int paramValues[] = {0, -1, (int) &validAddress};
...
SaAmfHandleT * param1 = (SaAmfHandleT *) paramValues[i];      (1)
SaNameT * param2 = (SaNameT *) paramValues[j];
```

A few values can result in a huge number of test cases in complex structures, however, if the values are not chosen carefully, the resulting failures would not be related to robustness. The efficiency of the following values was examined.

- *0*: It is a common test value since it represents a NULL when cast to a pointer. Using zero as an input caused many segmentation faults in OpenAIS 0.69 because in the A.01.01 version of the specification many parameters are pointers and in several functions the checking of the NULL value was not implemented yet.
- *-1, 1*: These values are helpful when there are parameters of integer (or float) type. However, in the case of pointers they will be cast to memory address usually reserved for the system. When de-referenced, they cause a segmentation fault, which is surely a robustness failure, but, as far as we know, this kind of invalid pointers cannot be checked in the API functions without specific compiler extensions.
- *Random value*: Random values are popular in robustness testing, however, the repeatability of the tests is not guaranteed.
- *Address of a valid variable*: We added this value for the sake of combining exceptional values with valid values (in case of pointers of variables). In this way the sensitivity to exceptional values of function parameters can be checked one by one.

Finally, we used two input sets. The first set {0, -1, 1, fixed random} resulted in several robustness failures but in this case the failures could not be traced back to the individual parameter values (i.e., which one activated the failure) since all values used in the function calls were (potentially) exceptional ones. The second set {0, valid address} was used specifically to determine which functions failed to implement null value checks.

4.2. Type-specific testing

In the case of *type-specific testing*, unique test values were constructed for each type used in the API. The following techniques were used to enhance the efficiency of this method.

Establishing type hierarchy: The types inherit the test values of their ancestors. This technique was very effective in Ballista. In AIS there are only a few types having ancestors in the type hierarchy, so this technique was used mainly for defining a basic type with common exceptional values.

Chaining of methods: This technique was introduced in JCrasher [15]. A call graph of methods is built, where an arc between two methods represents that an output of a method can be used as an input for the other. We applied this method on the AIS AMF in case of two functions (SaAmfInitialize and SaAmfCompNameGet) that produced output for others.

Identifying valid test outputs: We observed that for some test values valid test outputs can be a priori identified. E.g. using the exceptional value 'D.5.4' for SaVersionT could result in SA_ERR_VERSION. Similarly, the results of obvious exceptional values like e.g. not initialized handle, not valid version, non existent component name, can be identified and this information can be used to classify the test results reducing in this way the number of undecided tests.

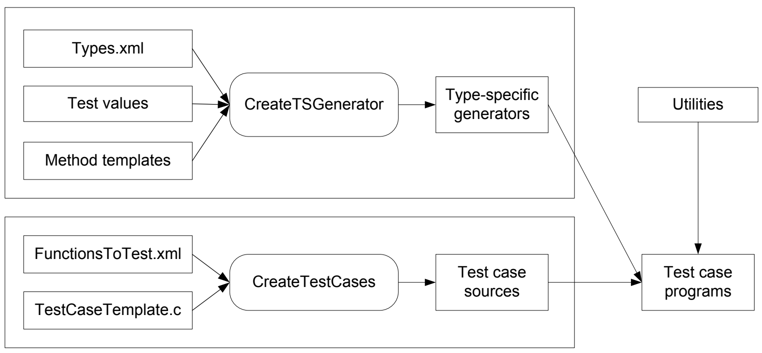

Figure 2. Architecture of the testing framework

Template-based test generation: The generic and type specific tests are implemented as separate C programs for each API function. Each program calls the API function with all combinations of the values returned by the *input value generators* and forks a new child process for each test case. The architecture of the testing framework is detailed in Figure 2. The type-specific input generators and the test sources are constructed automatically, based on templates as follows.

CreateTSGenerator constructs the C code for the type-specific input value generators. It uses the following sources and parameters:

- The *metadata of the types* for which exceptional values should be generated (types.xml, an example is found in Listing 2). Here ValidValueMethod designates the index of a valid test case. PointerMethod can initiate the construction of a method to access test values via pointers. If ParentName is present, all test cases of the given ancestor type are re-used.
- The *exceptional test values* stored in stand-alone files as C code snippets.
- The *C skeleton of the generator* and the templates for the methods.

```
<Type>
   <Name>SaDispatchFlagsT</Name>
   <ValidValueMethod generate="true" validValueIndex="1"/>
   <PointerMethod generate="false" />                         (2)
   <ParentName value="BaseType"/>
</Type>
```

The test case sources are constructed by CreateTestCases which is an XSL transformation that uses the following input files:

- *Test case templates* to be populated with test values.
- Information about the *API functions and their parameters* (FunctionsToTest.xml, an example is given in Listing 3). ParameterOrder is included explicitly, and IsPointer identifies whether the parameter is a pointer or not. In this way the later transformation will be easier.

```
<Function name="saAmfFinalize">
  <ReturnType>SaAisErrorT</ReturnType>
  <Parameters>
    <Parameter>
      <ParameterOrder>1</ParameterOrder>
      <ParameterName>amfHandle</ParameterName>
      <ParameterType>SaAmfHandleT</ParameterType>           (3)
      <IsPointer>true</IsPointer>
      <Type>in</Type>
    </Parameter>
  </Parameters>
</Function>
```

Finally, the input generators and the test case sources are compiled and linked with a utility library, which contains functions for logging the results.

5. *Scenario-based testing*

The previous techniques tested individual API calls without considering that the service of several AIS functions depends heavily on the current state of the middleware and they can only be used when a sequence of previous calls have set a specific state. These call scenarios could be obtained from two sources.

- The AIS specification contains several *sequence diagrams* that capture the basic operation of the system. Using a *model-based approach*, these diagrams are re-drawn as UML sequence diagrams and the skeleton of the call sequence is generated automatically.
- The other source is the *functional test suites* of the AIS implementations. There is a public test suite, SAF Test [16], which is an open-source project for testing the conformance to SA Forum's specifications. It includes the call sequences as C test programs that can be re-used for our purposes.

When a set of scenarios is constructed from the above sources, it could be used for two purposes. First, it can be used to *reach specific states* needed by the API functions. The scenario containing the function to be tested is selected and the execution sequence preceding the call of this function is applied before initiating the generic or type-specific tests. Second, additional test cases can be generated with the help of *mutation operators* that may activate robustness failures: substituting a pointer parameter with NULL, removing a call from the scenario and changing the order of function calls.

6. Efficiency of the testing techniques

The goal of our first experiments was to compare the *effectiveness of the techniques* and to highlight the advantages of implementing the testing tools.

The tests were executed on the AMF (17 functions) and CLM module (7 functions) of OpenAIS 0.69. Table 1 illustrates the complexity (and cost) of the generic and type-specific testing techniques. The initial version of the generic testing was created in approx. three days, while the implementation of the framework of type-specific testing required about two weeks. The main advantage of the automated testing approach is that the type-specific testing of a new function requires only the completion of the metadata, and supplying the test values and logging code for the new types used in the function.

Table 1. Number of lines in the source code of the robustness testing framework

Technique	Test template	Transformations	Metadata	Test values	Sum
Generic	120	80	417	1	618
Type-specific	323	690	726	254	1993

Table 2 lists the ratio of API calls that resulted in robustness failures and the number of test calls executed. (In case of functions with more than five complex parameters the number of test cases was limited to 4000.) CLM was more resilient to generic testing since it used less pointers than AMF.

Table 2. Comparison of the different exceptional input generation and testing techniques

Technique	OpenAIS AMF	OpenAIS CLM
Generic testing with invalid addresses	2406 / 2456	60 / 424
Generic testing with null and valid address	87 / 136	0 / 44
Type specific testing	8001 / 13640	65 / 2280

In case of several functions type-specific testing identified additional robustness faults in comparison with generic testing, while in case of three functions only type-specific testing was effective (Table 3). Scenario-based testing was necessary e.g. in case of initializing callback functions.

In our experiments the *decision tree method* of a data mining tool (IBM Intelligent Miner) was used to trace back robustness failures to faults, hence a *metric* to compare OpenAIS with different implementations in the future was obtained. In this way, the influencing factors could also be separated.

Table 3. Faults found in OpenAIS by functions. X + Y means that generic testing found X faults while type-specific identified Y more. The star denotes a critical error, which caused segmentation fault in the middleware executive.

Function name	Faults
saAmfCompNameGet	1
saAmfComponent CapabilityModelGet	1
saAmfComponentRegister	2
saAmfComponentUnregisterRegister	2
saAmfDispatch	1
saAmfErrorCancelAll	1
saAmfErrorReport	3
saAmfFinalize	1
saAmfHAStateGet	2
saAmfInitialize	0 + 2
saAmfPendingOperationGet	1
saAmfProtectionGroupTrackStart	2
saAmfProtectionGroupTrackStop	2
saAmfReadinessStateGet	1 + 1
saAmfResponse	1*
saAmfSelectionObjectGet	1 + 1
saAmfStoppingComplete	1*
saClmClusterNodeGet	0 + 1
saClmClusterTrack	0 + 1
saClmClusterTrackStop	0
saClmDispatch	0
ClmFinalize	0
saClmInitialize	0
saClmSelectionObjectGet	0

7. Conclusion and future work

Our paper discussed the problem of robustness testing of high availability middleware. We proposed a testing framework that integrates previous testing techniques and extends them by introducing tool-supported methods including scenario-based testing and test result classification. The case study conducted on

OpenAIS showed that while even simple techniques can identify robustness problems, it is necessary to implement the more complex methods, since they are able to find faults not detected by the simple techniques. In the future we plan to apply stressful environmental conditions and we will run the test suite on other AIS implementations to compare the robustness of the different products.

References

1. Application Interface Specification, Service Availability Forum, Feb. 2006., URL: http://www.saforum.org/
2. OpenAIS, AIS implementation, URL: http://developer.osdl.org/dev/openais/
3. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Standard 610.12.1990, 1990, URL: http://standards.ieee.org/
4. B. Miller *et al.*, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," Tech. Report, University of Wisconsin, 1995.
5. M. Schmid, F. Hill: "Data Generation Techniques for Automated Software Robustness Testing", in *Proceedings of the International Conference on Testing Computer Software*, Washington, USA, June 14-18 1999.
6. P. Koopman *et al.*, "Automated Robustness Testing of Off-the-Shelf Software Components," in *Proc. of Fault Tolerant Computing Symposium*, pp. 230-239, Munich, Germany, June 23-25, 1998.
7. H. Madeira *et al.*, *DBench Dependability Benchmarks*, project full final report, IST-2000-25425, 2003, URL: http://www.laas.fr/dbench/
8. K. Kanoun *et al.*, "Benchmarking Operating System Dependability: Windows 2000 as a Case Study," in *Proc. of 10th Pacific Rim Int. Symposium on Dependable Computing*, Papeete, French Polynesia, 2004.
9. IBM Autonomic Computing, 2006., URL: http://www.ibm.com/autonomic/
10. J. Zhu *et al.*, "R-Cubed (R3): Rate, Robustness and Recovery - An Availability Benchmark Framework," TR-2002-109, Sun Microsystems.
11. A. Ghosh, M. Schmid, "An Approach to Testing COTS Software for Robustness to Operating System Exceptions and Errors," in *Proc. of International Symposium on Software Reliability Engineering*, 1999.
12. V. Sieh, K. Buchacker, "UMLinux — A Versatile SWIFI Tool" In *Proc. of Fourth European Dependable Computing Conference*, 2002, pp. 159-171.
13. H. Madeira *et al.*, "The OLAP and Data Warehousing Approaches for Analysis and Sharing of Results from Dependability Evaluation Experiments," in *Proc. DSN-DCC 2003*, USA, June 2003.
14. Pintér G. *et al.*, "A Data Mining Approach to Identify Key Factors in Dependability Experiments," in *Proc. EDCC-5*, pp 263-280, 2005.
15. C. Csallner, Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java", *Practice & Experience*, vol. 34, no. 11, Sep. 2004, pp. 1025-1050
16. SAF Test, SAF-conformance test suite, URL: http://saftest.sourceforge.net/