UML-Based Modeling of Robustness Testing

Regina Moraes*, Hélène Waeselynck†‡, and Jérémie Guiochet†‡
*School of Technology – UNICAMP, Rua Paschoal Marmo, 1888, Limeira, SP, Brazil
†CNRS, LAAS, 7 av. du colonel Roche, F-31400, Toulouse, France
‡Univ. Toulouse, F-31400, Toulouse, France
Email: regina@ft.unicamp.br, helene.waeselynck@laas.fr, jeremie.guiochet@laas.fr

***Abstract*—The aim of robustness testing is to characterize the behavior of a system in the presence of erroneous or stressful input conditions. It is a well-established approach in the dependability community, which has a long tradition of testing based on fault injection. However, a recurring problem is the insufficient documentation of experiments, which may prevent their replication. Our work investigates whether UML-based documentation could be used. It proposes an extension of the UML Testing Profile that accounts for the specificities of robustness testing experiments. The extension also reuses some elements of the QoSFT profile targeting measurements. Its ability to model realistic experiments is demonstrated on a case study from dependability research.**

Keywords*—*Robustness Testing ; UML Testing Profile ; UML Profile extension ; Case Study

I. Introduction

Robustness testing is a specific form of black-box testing that complements conformance testing by studying whether erroneous or stressful input conditions (e.g., faults, or attacks) may alter the system's regular behavior. A robust system can handle unexpected inputs arising from other systems or the computational environment [1] and can deliver a dependable service even when submitted to an aggressive environment [2].

The dependability community has a long tradition of robustness testing based on fault injection [3]. The injection techniques span physical, design or unintentional interaction faults, as well as attacks. It is deemed crucial to assess the possible impact of faults and attacks on systems, in order to provide adequate protection mechanisms that mitigate or reduce this impact. The test outcome is typically a set of measurements rather than a pass/fail verdict. Effort to standardize this kind of testing has yielded the emergence of the concept of dependability benchmarking [4].

A recurring problem is the lack of approaches to document experiments, which might be one of the reasons why studies reusing tests to compare and to consolidate results are seldom available [5]. Our research investigates whether the Unified Modelling Language (UML) [6] could be used for documenting robustness testing. Its starting point is the UML Testing Profile (U2TP) released by the Object Management Group (OMG). This profile offers a set of concepts for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of system testing [7]. Note that advanced research around U2TP goes beyond documentation purposes. It investigates complete development chains where test code (or code skeleton) is produced from the UML models [8], [9]. This is done in the framework of conformance testing. We are not aware of work using U2TP in the framework of robustness testing with fault injection. In particular, U2TP has not gained attention in dependability research, where UML-based work focuses on guiding early system design choices [10] [11].

This paper proposes a specialization and extension of U2TP that addresses the documentation of robustness testing experiments. It also reuses a few elements from the Quality of Service and Fault Tolerance profile (QoSFT) [12] to represent robustness measurements. The core of the contribution is then on the testing concepts to be reused, adapted or created based on U2TP. A set of key elements for extending U2TP is first identified. Its capability to represent real experiments is explored, taking an example from dependability research in the area of service-oriented systems [13]. The UML models were validated with the authors of the original experiments.

The next sections are organized as follows: Section II presents a brief overview of robustness testing in dependability research; starting from U2TP, adaptations for robustness testing are presented in Section III; Section IV applies the documentation approach to the study; Section V concludes this paper.

II. Robustness testing in dependability research

According to Avizienis *et al* [14], the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable. Threats to dependability are *failures* (service deviates from correct service), *errors* (system state that may cause a subsequent failure) and *faults* (adjudged or hypothesized cause of an error).

To achieve high levels of dependability, fault tolerance mechanisms must be implemented. To experimentally assess the adequacy of the implemented mechanisms, testing based on fault injection is frequently used [3], [15]. The test input domain has then two main dimensions: the functional activity (workload) and the faults (faultload, or attackload when security is the focus). A key issue is the identification of faultloads representative of real fault sets [16], [17], [18]. Another one is how to combine the faultload and the workload, so as to avoid useless experiments where the injected faults would not be activated [19]. The test output domain includes readouts (observation data collected) from which measurements are derived. For example, the aim may be to get an estimate of the coverage provided by the fault tolerance mechanisms [20], or to measure the error latency [21].

Besides the evaluation of fault tolerance mechanisms, fault injection has proven a useful technique to characterize the behavior of systems in the presence of faults, with targets as different as embedded control systems [19], operating systems

[22] [23], middleware [24], communication infrastructure and protocols [25] [26], and web services [13].

This work focuses on documenting such fault injection experiments issued from the dependability community. The typical profile of an injection campaign is to have a golden run followed by one or several injection runs. During the golden run, the workload is executed without the injection of the faults and the readouts are collected (baseline phase). During the injection run, the very same workload is executed in the presence of the faultload, and the new readouts are collected (test phase). At the end of the experiments, the observed results are analysed and measurements are derived (check phase).

III. UML MODELS FOR ROBUSTNESS TESTING

After a brief presentation of U2TP [7], we show how robustness measurement can be introduced by using a fragment of either QoSFT [12] or MARTE [27] profiles. We then give an overview of all new stereotypes and interfaces dedicated to robustness testing.

A. OMG UML testing profile

OMG published the UML 2.0 Testing Profile (U2TP) [7] to describe test components and activities. The goal of U2TP is to provide a means to specify static (structural) and dynamic (behavior) aspects of UML models and to incorporate existing test technologies for functional black-box testing. The UML Testing Profile is organized in four logical groups of concepts: Test Architecture, Test Behaviour, Test Data and Test Time.

The Test Architecture group describes the architecture of components and their configuration. The Test Behavior includes concepts to specify the dynamic aspects of test procedures and the implementation of the test cases. Test Data defines the syntax and semantics of processed input / output values (i.e., data that is used in test cases). Time concepts are used as a means to manipulate and control test behavior, or to ensure the proper termination of a test case.

The robustness testing case study (Section IV) will exemplify the use of many U2TP concepts like: SUT (system under test), Test Context (contains a collection of test cases, specifies the test configuration and control), Test Components (interact with the SUT to realize the test cases), and Scheduler (controls the execution of test components) However, some concepts need to be specialized or newly introduced to cover the specificities of robustness testing. We want measurements to be used in place of verdicts. We also want the models to explicitly identify the structural and behavioral elements in charge of the injection of faults.

B. Robustness measurement concepts

Robustness testing involves quantitative assessment; hence we need concepts for measurement. QoSFT [12] and MARTE [27] are two standardized profiles including quantitative concerns. Either one or the other can provide a solution for us. As regards QoSFT, note that the focus is on its QoS Characteristics package addressing quantifiable characteristics of services.

As presented in Figure 1, both QoS and MARTE allow the definition of robustness dimensions (here we define two dimensions, FaultCoverage and ErrorLatency). QoS offers a

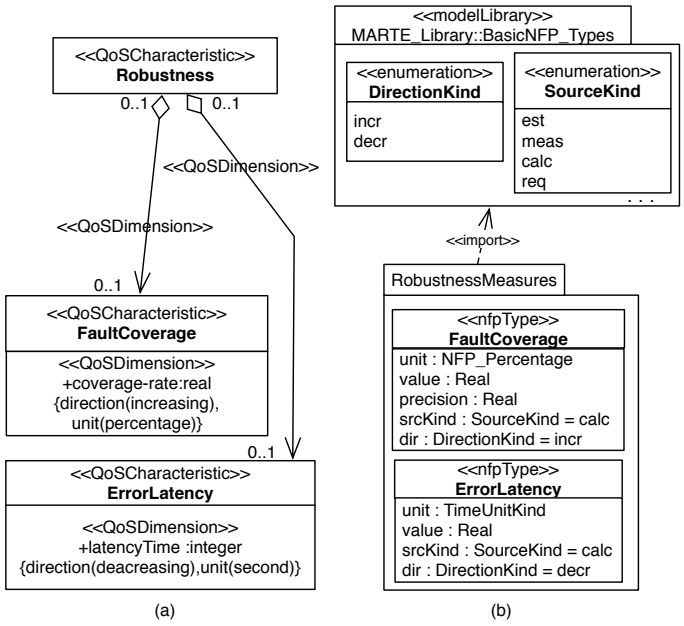

Fig. 1: Specification of robustness measurements using (a) QoS or (b) MARTE

recursive notion of QoSCharacteristics. Indeed, a characteristic may have several dimensions, where each dimension can be a characteristic on its own (e.g. FaultCoverage might have several dimensions). For each leaf dimension, a compact notation is offered to specify the measurement unit (e.g., a time unit for error latency), and also the direction to order two values compared with a higher-quality relation (e.g., the higher the fault coverage, the better the quality). This is specified as a constraint put on this dimension: {direction(increasing)}. MARTE does not provide the multi-dimensional and multi-level structuring of measurements. Indeed in MARTE, a non functional property (nfp), can only be associated with one nfpType (a nfp dimension). Hence, we used a package (RobustnessMeasures) to introduce a structure similar to the one of QoS. Even if MARTE also offers a rich set of predefined elements (e.g., NFP_Percentage, SourceKind, DirectionKind), we chose QoS Characteristics to represent robustness measurements, favoring structuring facilities and compactness.

C. Robustness Testing Extension

This proposal considers the reuse and integration principles that UML suggested through the profiles idea. The main construct in a profile is the stereotype. This extension mechanism allows us to create new modeling elements, bringing specific properties related to robustness testing. Stereotypes help us to identify elements of interest in a model.

Our proposal inherits and extends elements from the UML Testing Profile, the vast majority of elements being directly reused as they were defined in the original profile. In addition, QoS characteristics are re-used. The new elements we introduce are presented in Figure 2. In this figure, elements on the left originate from UML Metamodel definition, elements in the middle originate from U2TP or QoSFT Profiles, and elements on the right represent our new stereotypes and interfaces.

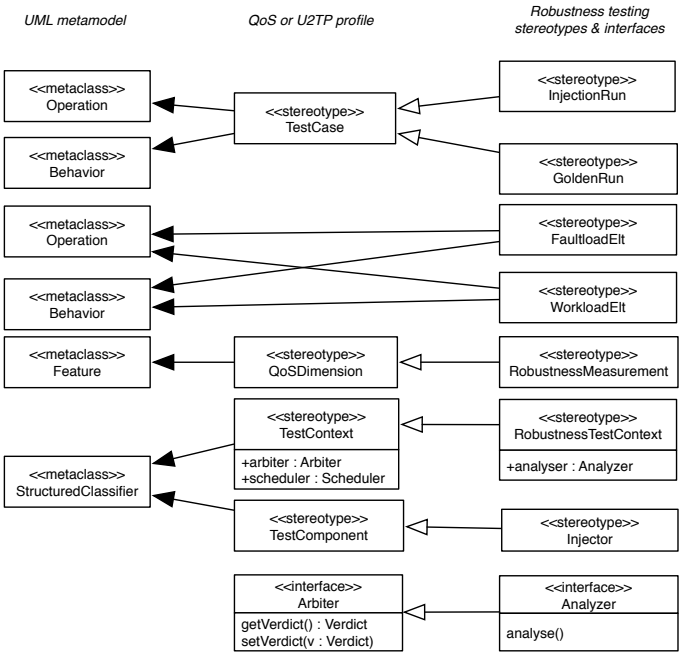

Fig. 2: Stereotypes and Interfaces of the Robustness extension

The central role is the <<RobustnessTestContext>> stereotype, specializing the original U2TP <<Test Context>> (see Figure 2). It is used to specify the context of a robustness test, aggregating the test configuration and test cases to be applied on the SUT. The set of injected faults is defined as the faultload. Each specific fault is represented by faultload elements. The <<FaultloadElt>> stereotype extends UML 2.0 Behavior and Operation metaclasses. This stereotype can be used to specify a complete behavior (e.g., a wrong order in a sequence of messages), or a single operation (e.g., a message with wrong parameters). It allows us to represent injection procedures of varying complexity. Similarly, the workload is defined as a set of <<WorkloadElt>>. It is responsible for the activity of the SUT, during the <<GoldenRun>>, when no artificial fault is present and also during the <<Injection-Run>> when <<FaultloadElt>> are injected. The golden and injection run concepts are represented as stereotypes, because they are two different test cases in the robustness test context. The former is the nominal testing scenario, whereas the latter includes injected faults. The fault injection task is performed by the <<Injector>> that takes advantage of its knowledge about both the workload and the faultload to place the specific <<FaultloadElt>> into a <<WorkloadElt>>.

Finally, an essential task is to collect results for quantitative analysis. The <<Analyzer>> is responsible for the analysis of all the results obtained in order to compose the <<Robustness-Measurement>> based on a set of measurements. It replaces the original Arbiter of U2TP that was in charge of the elaboration of pass/fail verdicts.

Figure 3 presents the main concepts used in our approach, using the Meta Object Facility (MOF) notation, i.e. using a sub-set of the UML notation to represent concepts [28]. All new elements can be presented on the same class diagram in order to define relations and their cardinalities. The meta-

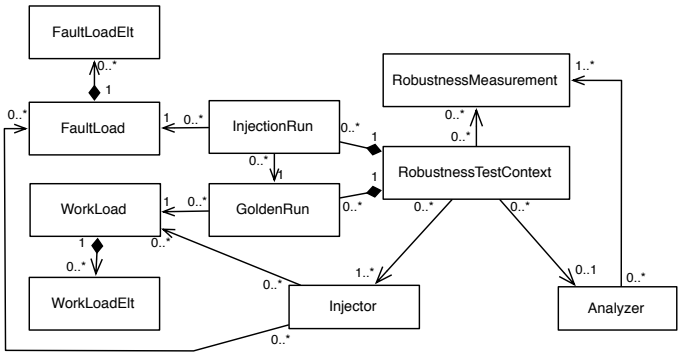

Fig. 3: MOF Metamodel for Robustness Testing concepts

model also contains high-level concepts (like Workload and Faultload) that are meaningful to the domain, but are not used as stereotypes or interfaces in concrete models.

IV. A CASE STUDY ON WEB SERVICES

To investigate the usability of the proposed concepts, our case study is the work by Vieira *et al.* [13] titled Assessing Robustness of Web-services Infrastructures. Web-Services (WS) are increasingly being used as a strategic way for data exchange and content distribution. Robustness of individual services and server infrastructures is highly desirable: when several WS work together to achieve an objective, a local failure could severely impact the composition result.

The robustness testing approach of Vieira *et al.* is applied to WS specified in the TPC-App performance benchmark [29], which considers a retail distributor on the Internet. It accepts WS Requests to place orders, view and make changes to the catalogue items, update or add customer information, or request the status of an existing order. The experiments in [13] focus on two different implementations of a subset of services that are tested and compared. They both run on top of a JBoss application server. The injected faults consist in modifying the SOAP messages sent to request the services.

For space constraints, this section presents a subset of the UML models we developed for this case study. It should be sufficient to show the profile concepts in action, and demonstrate their potential to document a non-trivial robustness testing experiment.

A. Measurement of Robustness

The measurements selected by the authors are modeled in Figure 4. Their representation makes use of QoS stereotypes as defined in Section III-B where $QoSDimension$ stereotype has been specialized as $robustnessMeasurement$.

One of the measurements is the crash scale. It was first introduced by Koopman and De Vale [22], and is now commonly used in robustness testing experiments. With slight adaptations to address WS, the C.R.A.S.H. scale categorizes the failure modes of SUT as follows: **Catastrophic**: the application server becomes corrupted or the machine crashes or reboots; **Restart**: the web service execution hangs and must be terminated by force; **Abort**: abnormal termination of the web-service with an exception raised; **Silent**: after a timeout, no response from the

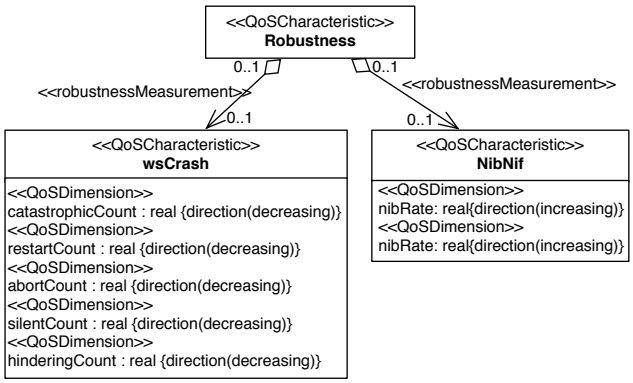

Fig. 4: WS robustness measurements

server and no error report; **Hindering**: the error code returned is not correct or the response is delayed.

The analyzer is in charge of determining the number of failures observed for each mode (only the Abort mode occurred during the experiments). In conformance with the QoSFT profile, the decreasing direction indicates that the lower the number of failures, the higher the robustness.

The other measurements are intended to characterize the performance of the services with and without faults. NI_b is the baseline average number of successful interactions per minutes (golden run) while NI_f is the equivalent measure in the presence of faults (injection runs). The favorable direction is increasing.

The case study nicely exemplifies the differences between an analyzer (used in robustness testing) and an arbiter (used in conformance testing). An arbiter reports one ordinal verdict, e.g., pass or fail; the elaboration of the verdict is often part of the execution of a test case. In contrast, the analyzer works off-line from a log file. It performs a quantitative evaluation, and there are usually several values to be reported because no single value can characterize robustness.

Indeed, comparing the robustness of alternative implementations is an acute issue. Let us take the example of a comparison according to the crash scale. Generally speaking, it is obvious that the catastrophic failure mode is more severe than, say, the hindering one. However, whether it is preferable to have sparse crashes and a correct handling of invalid parameters the rest of the time, or no crash but frequent abnormal terminations and inappropriate error codes returned, is left to the decision of service integrators. The number and repartition of failure modes are useful indicators; they could not easily be aggregated into an absolute robustness level. Moreover, other indicators may also be of interest, like the degradation of performance. The inherent difficulty of robustness evaluation is captured by the concepts we reused from the QoS profile, i.e., evaluation must account for multiple characteristics and multiple dimensions of the characteristics.

B. Experiment Architecture

Figure 5 provides a partial view of the test architecture. The target of the test is the service provider (*Server*) as indicated by the *SUT* stereotype. The *ConsRBE* component, playing the role of a consumer, thus acts as a test driver for the server (see its *testComponent* stereotype). The server processes the requests sent by *ConsRBE* and sends back a response message. In normal operation, the network mediates client/server communication. In the test experiments, the proxy intercepts all messages issued by the client, and possibly modifies them before they are passed to the server. In Figure 5, *Proxy* has the *injector* stereotype: it is a test component with the capacity of injecting faults. This is a major role for robustness testing.

A partial view of the architecture is a convenient support to show the major components, their roles, and the interaction flows between them. The interfaces of individual components can be further documented in dedicated models.

Most of the time, robustness testing requires a complex test environment. Besides the major test components, it is not rare to use many auxiliary components to aid in the control and monitoring of test execution. The case study we took from [13] is no exception and exhibits a complex test architecture.

In the UML Testing Profile, the composite structure of a test context is a collection of test component objects and connections between these objects and the SUT. Figure 6 shows this composite structure for the case study. It involves component objects already discussed, like the server, the network and proxy. The client *ConsRBE* is connected to a controller that is responsible for telling *ConsRBE* when to start and stop the generation of requests (controls the generation of the workload). In addition to them, a *Coordinator* is responsible for managing all the experimental process; it implements the *Scheduler* interface of the UT2P profile. *Executor* is a daemon component running on each machine. It processes Coordinator's requests to create new processes during the experiment execution. *Analyzer* is responsible for analyzing data logged in a File and for determining final measurement results (see Section IV-A). Finally, the *Loader* is a component to load data. It provides remote access to a *Database* and other resources used during execution.

It is important to properly document the set of required test components, as well as the connections between SUT, test components and the fault injection part. Pure textual descriptions are likely not to be clear enough. Diagrammatic views, such as the UML ones, offer a much more adequate support. The stereotypes attached to model elements also contribute to the clarity of the diagrams, by making it explicit which roles are played by the components.

C. Fault Model and Data Representation

Fault injection focuses on corrupting SOAP service requests sent to the provider through the network (using http or https protocols, for example). The injection changes the value of a parameter in either the message header (http version number, message length) or the message body (e.g., the customer id of a *ChangePaymentMethod* request). The mutation operators depend on the type of the parameter. For instance, the http version number in the header can be replaced by a null value, another valid version number or an invalid version number. In [13], the authors represent the fault model by mutation tables for the considered types of parameters. We demonstrate how the data mutation concept can be built from existing data concepts in U2TP (data partition and data pool).

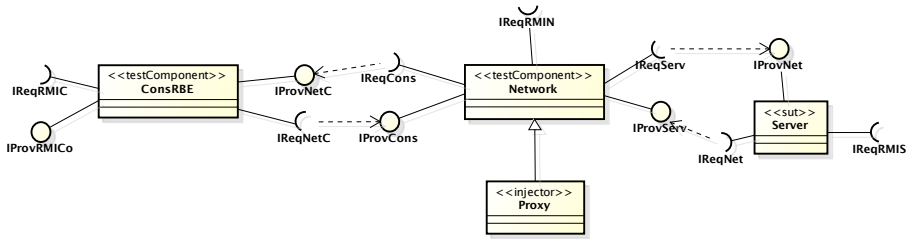

Fig. 5: Partial architecture of the WS system

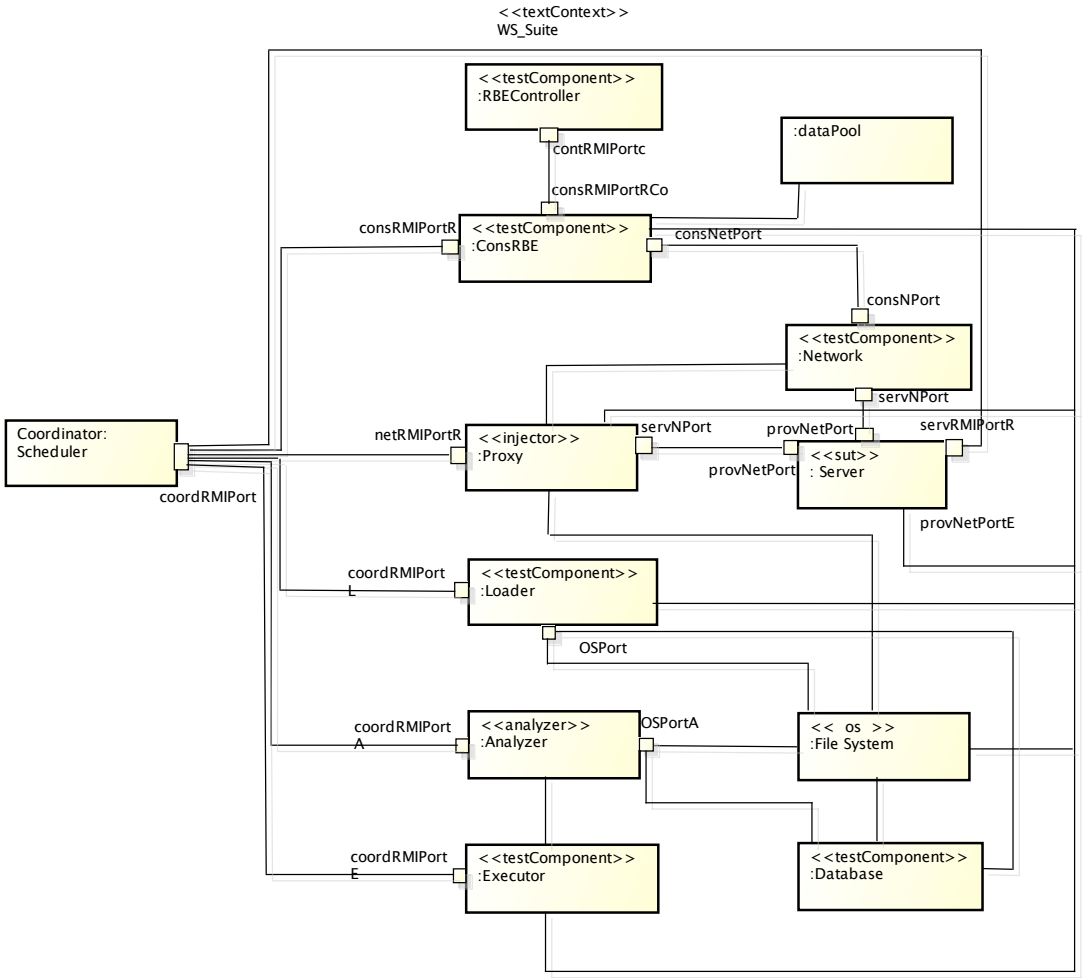

Fig. 6: Composite structure of the test context

A data partition defines an equivalence class for a set of values used in a stimulus or observation. A data pool aggregates the partitions and concrete values used in a test context. Data selectors provide strategies for selecting them. Figure 7 presents our approach to model the corruption of existing data. Its principle is generically shown for a service request with two parameters (in the case study, *Request1* would be a request like *NewCustomer*, *ChangePaymentMethod*, etc.). The message data is modeled by class *MsgRequest1* that inherits a header from *Msg*. Fault models are attached to the various parameters. Both valid and faulty requests can be selected from the data pool. For illustrative purposes, two exemplary partitions of faulty requests are shown: *Request1FaultHttpVersion* and *Request1FaultParam1*. Each of them is linked to a valid partition and a fault model. Accordingly, the *changeMsg* method returns faulty requests (note its *FaultloadElt* stereotype). Of course, other partitions could be built to accommodate diffrent injection strategies. For instance, a partition could be linked to several fault models for multiple corruptions of parameters at the same time.

D. Golden Run and Injection Runs

The Golden Run precedes the fault injection experiments. Its goal is to characterize the behavior of the system in the

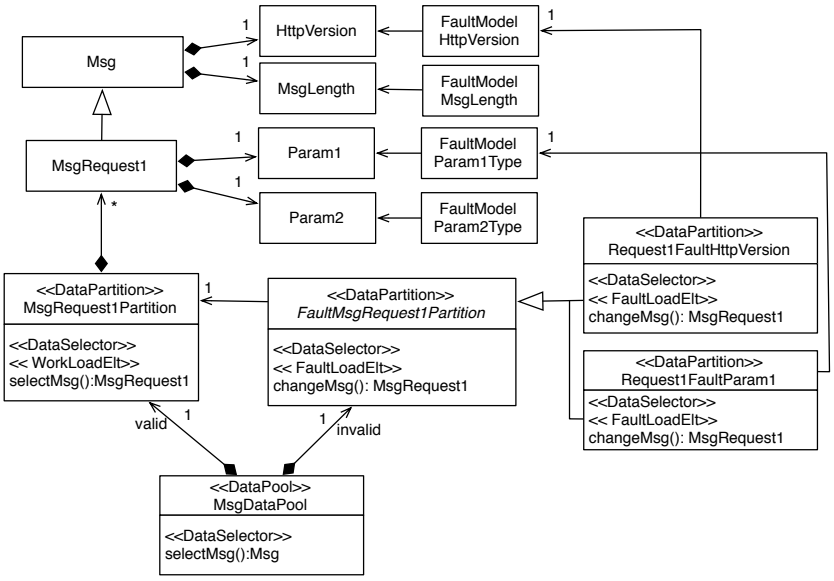

Fig. 7: Data representation of valid and faulty requests

absence of artificial faults. In the case study, it consists of sending requests (Stimulus) to the server, and logging the observable events resulting from their processing (Observation). The results are stored for comparison with the injection runs. The sequence diagram in Figure 8 gives a generic view of the interactions among the objects. Test components are created, the server is configured and functional activity is started. The activation of the server has predefined time duration. Workload messages are produced until time is elapsed (see loop fragment). During this time, the behavior of the server is observed. Both normal and abnormal behavior is logged, as shown in the *alt* fragment. Although the workload messages go through the proxy, they are forwarded unchanged to the server. The proxy's fault injection capability is inhibited.

After the Golden Run, the very same workload is performed in the presence of faults. Figure 9 presents the loop fragment of the fault injection procedure. The proxy now modifies either the message header or the value of a parameter in the message body (in accordance with the type of the target parameter). The modified message continues its route to the server, whose behaviour is observed.

V. Conclusion

This work proposes an extension to the UML Testing Profile that makes it suitable for the documentation of robustness testing experiments. *Golden* and *injection runs* are introduced as a specialization of test cases. Their behavioral description involves *workload* and *faultload elements*. Test components can have the *injector* role. An *Analyzer* in charge of measurements replaces the Arbiter in charge of verdicts. Robustness is a multi-dimensional *QoS characteristic*, as defined by the QoSFT profile. An alternative definition could have used a set of MARTE's non functional properties grouped into packages.

To show the proposed concepts in action, we used experiments from a robustness testing research group. The UML modeling exercise involved both structure and interaction diagrams. It proved challenging as regards the complex test architecture. We found that the visual support of UML diagrams, specialized with stereotypes to indicate the roles of components, was really useful for documentation purposes. In the sequence diagrams, stereotypes attached to workload and faultload elements are also convenient to identify which interactions correspond to the activation part of the test, among the many other interactions necessary for test configuration and logging. Overall, the outcomes of the case study were quite promising in respect of the capacity to model realistic robustness testing experiments.

The models were validated with the help of the authors of the original experiments. Several iterations were necessary to incorporate details that would not have their place in [13], but are important for documentation. The feedback we received was encouraging. No deep expertise on UML or U2TP was necessary to understand the models. The authors could recognize the representation of their experiments and even point at errors when we misunderstood the experimental settings. It bodes well for the intelligibility of our modeling approach for peers in dependability research.

Our future work will elaborate on the reproducibility of experiments. We will take a recent example of experiments developed by some of us at UNICAMP, and will try to replicate it at LAAS based on its documentation. It will give us further insights into the methodology, e.g., what is worth modeling and to which level of details.

Acknowledgements

This work was initiated during Regina Moraes six-month stay at LAAS/CNRS, supported by CAPES BEX3587-10-0. The work was also supported by RobustWeb - CAPES 0580/08. We thank Nuno Laranjeiro from the University of Coimbra for his availability to validate our models.

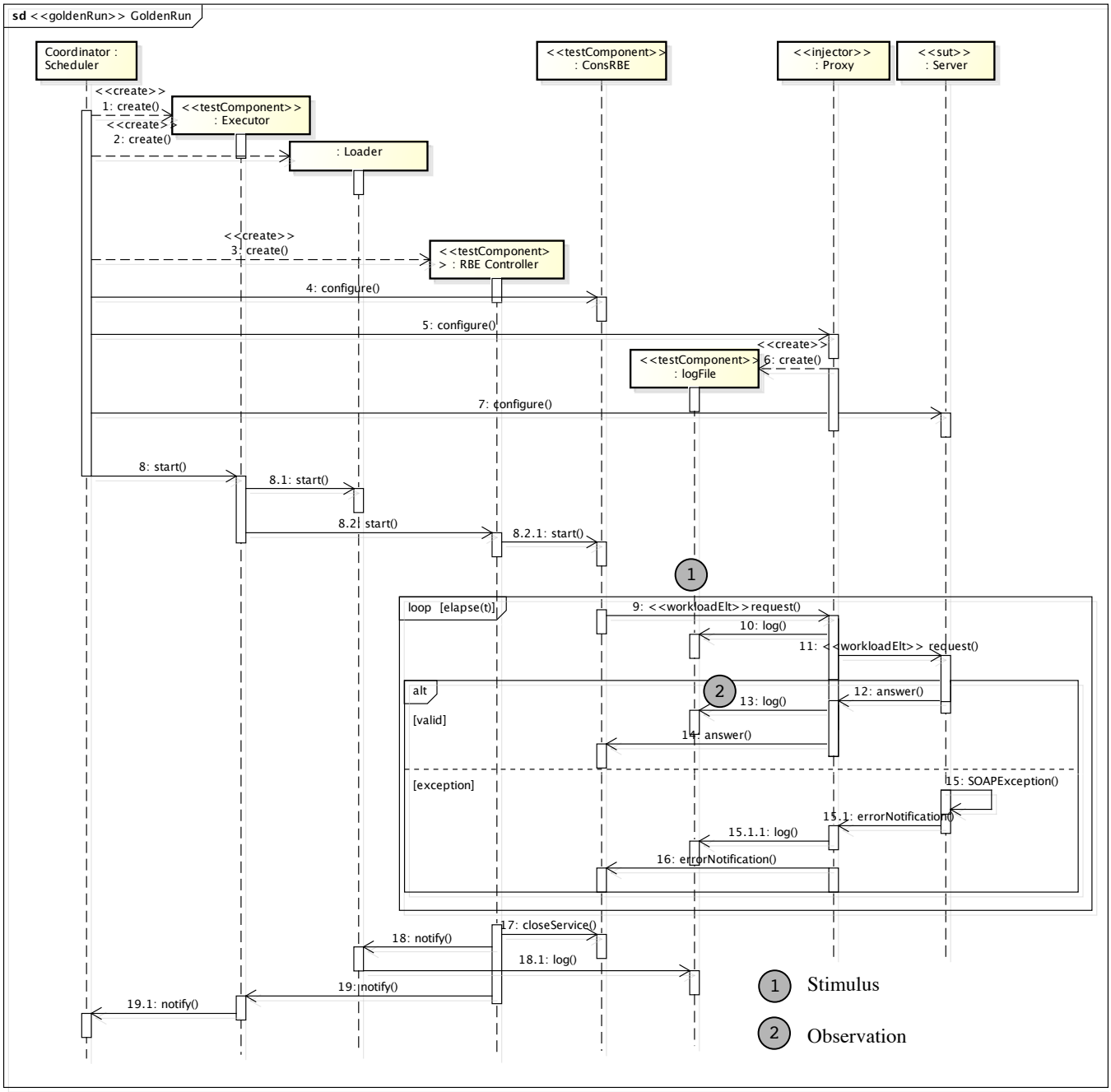

Fig. 8: Golden Run procedure

REFERENCES

[1] J. Cohen, D. Plakosh, and K. Keeler, "Robustness testing of software-intensive systems: Explanation and guide," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2005-TN-015, 2005.

[2] J. Voas, "Certifying off-the-shelf software components," *IEEE Computer*, vol. 31, no. 6, pp. 53–59, 1998.

[3] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation - a methodology and some applications,," *IEEE Trans. on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[4] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. Wiley and IEEE Computer Society Press, 2008.

[5] J. Arlat and R. Moraes, "Collecting, analyzing and archiving results from fault injection experiments," in *Proc. Fifth Latin-american Symp. on Dependable Computing*, Sao Jose dos Campos, SP, Brazil, 2011.

[6] OMG, "UML 2.4.1 Superstructure and Infrastructure," Object Management Group, 2011.

[7] ——, "UML 2.0 Testing Profile v.1.1, formal/2012-04-01," Object Management Group, Tech. Rep., 2012.

[8] J. Zander, Z. Dai, I. Schieferdecker, and G. Din, "From U2TP models to executable tests with TTCN-3: An approach to model driven testing," in *Proc. Int. Conf. on Testing of Communicating Systems TestCom*, 2005, pp. 289–303.

[9] P. Baker and C. Jervis, "Testing UML2.0 models using TTCN-3 and the UML2.0 testing profile," in *Proc. SDL 2007, LNCS 4745*. Springer,

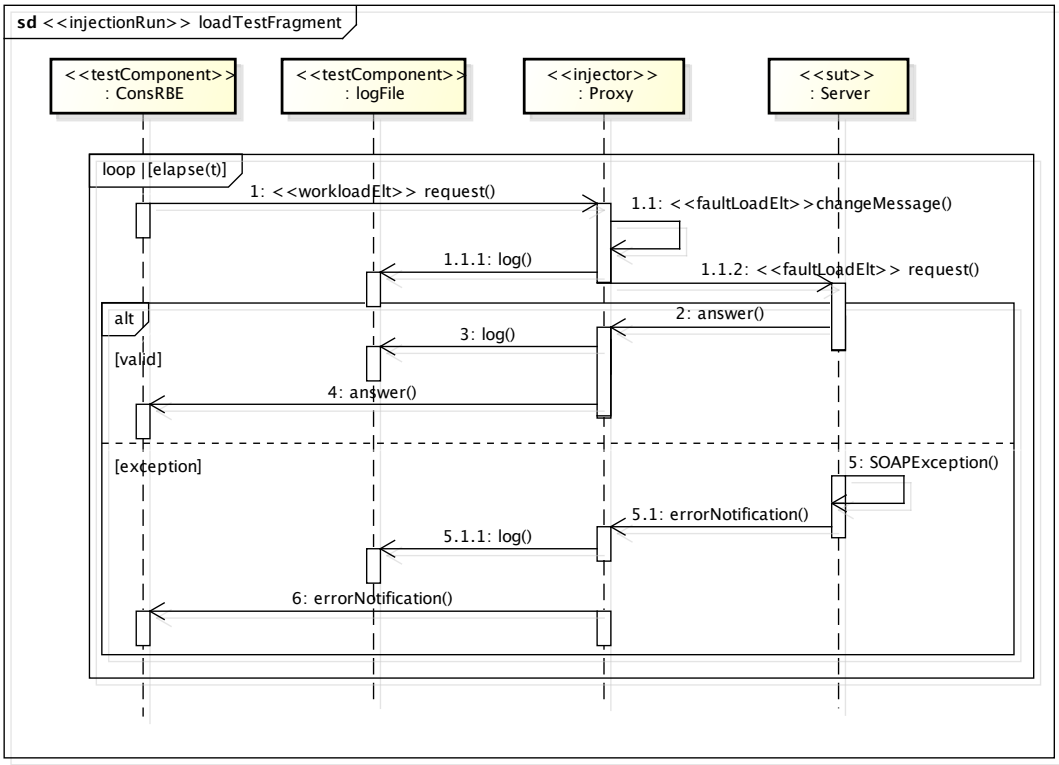

Fig. 9: Fragment of Fault Injection procedure

2007, pp. 86–100.

[10] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia, "Dependability analysis in the early phases of UML based system design," *Int. Journal of Computer Systems Science and Engineering*, vol. 16, pp. 265–275, 2001.

[11] S. Bernardi, J. Merseguer, and D. Petriu, "A dependability profile within MARTE," *Software and System Modeling*, vol. 10, no. 3, pp. 313–336, 2011.

[12] OMG, "UML profile for modelling quality of service and fault tolerance characteristics and mechanisms, v.1. formal/06-05-02," Object Management Group, 2006.

[13] M. Vieira, N. Laranjeiro, and H. Madeira, "Assessing robustness of web-services infrastructure," in *Proc. 37th IEEE-IFIP Int. Conf. on Dependable Systems and Networks - DSN*, 2007.

[14] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, pp. 11 – 33, 2004.

[15] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[16] V. Sieh, O. Tschche, and F. Balbach, "Comparing different models using verify," in *Proc. 6th Int. Working Conf. on Dependable Computing for Critical Applications - DCCA-6*, Germany, 1997, pp. 59–76.

[17] J. Duraes and H. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Trans. on Soft. Engineering*, vol. 32, pp. 849–867, 2006.

[18] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, and H. Martins, E.and Madeira, "Injection of faults at component interfaces and inside the component: Are they equivalent?" in *Proc. European Dependable Computing Conference - EDCC*, Portugal, 2006, pp. 53–64.

[19] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Experimental dependability evaluation of a fail-bounded jet engine control system for unmanned aerial vehicles," in *Proc. European Dependable Computing Conference*, Hungary, 2005, pp. 246–262.

[20] W. Bouricius, W. Carter, D. Jessep, P. Schneider, and A. Wadia, "Reliability modeling for fault-tolerant computers," *IEEE Trans. on Computers*, vol. 20, no. 11, pp. 306–311, 1971.

[21] J. Arlat, A. Costes, Y. Crouzet, L. J-C., and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transaction on Computers*, vol. 42, no. 8, pp. 913–23, 1993.

[22] P. Koopman and J. De Vale, "Comparing the robustness of POSIX operating systems," in *Proc. 29th Annual Int. Symp. on Fault-Tolerant Computing. Digest of Papers*, 1999, pp. 30–37.

[23] K. Kanoun and Y. Crouzet, "Dependability benchmarks for operating systems," *Int. Journal of Performability Engineering*, vol. 2, no. 3, pp. 275–287, 2006.

[24] N. Laranjeiro, M. Vieira, and H. Madeira, "Experimental robustness evaluation of JMS middleware," in *IEEE Int. Conf. on Services Computing*, vol. 1, 2008, pp. 119–126.

[25] D. Stott, G. Ries, M.-C. Hsueh, and R. Iyer, "Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection," *IEEE Trans. on Computers*, vol. 47, no. 1, pp. 108–119, 1998.

[26] D. Andres, J. Friginal, J.-C. Ruiz, and P. Gil, "An attack injection approach to evaluate the robustness of ad hoc networks," in *Proc. IEEE Pacific Rim Int. Symp. on Dependable Computing*, China, 2009.

[27] OMG, "UML profile for modeling and analysis of real-time and embedded systems (MARTE) version 1.0, formal/2009-11-02," Object Management Group, 2009.

[28] ——, "MOF meta Object Facility core v 2.4.1," Object Management Group, 2011.

[29] T. P. P. Council, "TPC benchmark app (application server) standard specification, version 1.1," 2005.