RobusTest: Towards a Framework for Automated Testing of Robustness in Software

Ali Shahrokni, Robert Feldt
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
ali.shahrokni, robert.feldt@chalmers.se

***Abstract*—Growing complexity of software systems and increasing demand for higher quality systems has resulted in more focus on software robustness in academia and research. By increasing the robustness of a software many failures which decrease the quality of the system can be avoided or masked. When it comes to specification, testing and assessing software robustness in an efficient manner the methods and techniques are not mature yet.**

This paper presents the idea of a framework RobusTest for testing robustness properties of a system with focus on timing issues. The test cases provided by the framework are formulated as properties with strong links to robustness requirements. These requirements are categorized into patterns as specified in the ROAST framework for specifying and eliciting robustness requirements. The properties are then used for automatically generating robustness test cases and assessing the results.

***Index Terms*—Robustness, real time systems, testing, timing**

I. Introduction

Robustness is an essential software quality attribute that is defined as [1]:

> The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.

Timing properties of software have a major role in determining the degree of robustness of the system. In our previous work [2] we focused on elicitation and specification of robustness requirements. In that study we have categorized requirements for developing a robust system in form of patterns. The motivation for using patterns is to capture the commonalities in structure and purpose of the requirements. The patterns can in general be divided into requirement patterns with the main focus to detect and solve robustness issues intrinsically and patterns that provide extrinsic architectural and design means to prevent robustness issues to surface. The patterns that provide intrinsic robustness are mainly to assure input stability or execution stability of the software. Input and events can be erroneous in two main manners, which can cause instability in systems, incorrect value or incorrect timing. The majority of academic research on robustness has so far been focused on stability of the system given erroneous input [3], [4], [5], [6], [7]. This paper discusses the stability in the presence of input and events with invalid timing. In ROAST, there are seven different patterns that focus on this problem area.

Robustness testing tools for generating random test data such as Ballista [3] and JCrasher [4] are the most well known methods of testing robustness of software systems. Using these frameworks help the user to assess how the system behaves in presence of input with invalid value. These frameworks are automated and the user has very little power to specify what data to test and what the expected result is. Instead, they use simple oracle frameworks such as CRASH [8], which is introduced later in this paper to determine whether the randomly generated input data results in failure in the system. Moreover, there is no or very little focus on the timing aspects of the input data in these framework.

Another framework which works on specifying and testing timing properties of a system is called Timed Input Output Automata (TIOA). With TIOA the user can model the interfaces of the system and specify the expected time intervals for the communications and between the different states of the system [9]. This model can then be used to automatically generate random test cases. To use TIOA the user often needs to create a sequential and large model of the system. Furthermore, when it comes to testing, TIOA mostly focuses on testing for timeout and has no or very little focus on other causes or patterns that can create robustness issues [9].

This paper presents the structure of RobusTest, which is a framework included in the ROAST framework for testing robustness requirements with timing focus. By writing testable properties, RobusTest automatically generates robustness test cases. If specified in sufficient detail these properties can be used as an oracle for the test cases. If there is no specification of the expected behavior, RobusTest oracle will assure that the test case will not put the system in a state with catastrophic consequences using the CRASH benchmarking framework. RobusTest not only provides to a large extent automatic testing but also a strong traceability between the generated test cases and the requirements through properties.

Section II discusses some of the concepts used to build the RobusTest framework. In Section III, we present the RobusTest patterns dealing with timing issues, test case generation, executor and oracle included in RobusTest. Finally, Section V discusses the current and future work planned for RobusTest.

 ISBN: 978-1-61208-168-7

II. BACKGROUND

In the first part of this section, the robustness benchmarking CRASH is introduced. The second part discusses the concept of property based testing and some of the techniques and tools available for this topic. Both these concepts are used in the framework RobusTest.

A. CRASH

The CRASH framework for benchmarking the robustness of operating systems (OS) was introduced by Koopman and is described in [8]. This framework acted as a simple oracle where the availability of functionality and the functions in the system rather than the correct functionality after the occurrence of a robustness issue can be measured. CRASH is used as the default oracle built in to the RobusTest framework. The CRASH framework is explained below, which will be implemented in our solution as an underlying layer to the framework. However, using RobusTest framework the expected functionality can be specified and benchmark on top of the extent of functionality.

- C Catastrophic (OS crashed multiple tasks affected)
- R Restart (task/process hangs, requiring restart)
- A Abort (task/process aborts, e.g., segmentation
- S Silent (no error code returned when one should be)
- H Hindering (incorrect error code returned)

Catastrophic class occurs when a fault in a part of the system under test (SUT) results in failure in other parts or even crash or hanging of the whole SUT. It usually requires hardware or software restart of the SUT. The Restart class occurs when one task hangs and can be resolved by killing or restarting that task. The Abort class occurs when a single task is abnormally terminated. The Silent class occurs when invalid parameters are submitted, but neither an error return code nor other task failure is generated. The Hindering type of failures is caused when the diagnosis is incorrect and could cause incorrect recovery.

B. Property based testing

A property is a statement which specifies how a system should or should not behave in a specific situation [10] in contrast to a test case that is set of executions done in a certain order. Using property based testing (PBT), high level properties of the system that should hold are stated and they are used to generate test cases in order to verify and validate a certain aspect or property of the system. In PBT a property is specified in a low level specification language. A PBT specification language should provide temporal and logical operators and location specifiers to the tester [10]. This specification written is then used to automatically generate test cases for that property. The expected behavior of the system is also specified in the property specification that can be used by the oracle to automatically analyze the results from the test execution.

Property-based testing intends to establish formal validation results through testing. "To validate that a program satisfies a property, the property must hold whenever the program is executed. Property-based testing assumes that the specified property captures everything of interest in the program, because the testing only validates that property" [10]. Notice that property based testing is in the same way as robustness testing a complementary to other types of verification and validation activities.

Fink et al. have used property based testing to identify security flaws and vulnerabilities in critical Unix programs such as *sendmail* [11], [12]. In the recent years, this type of testing has received increasing attention from the industry and research community. One example is the ProTest project[1] financed by the European Commission to improve methods and tools for property based testing. A well known tool for property testing is QuickCheck, which was initially developed for functional programming languages such as Haskell and Erlang but has now been developed for Java and other languages [13]. An example property written in QuickCheck for testing the functionality to reverse a list of integers can look like this [14]:

$$prop\_reverse() ->$$
$$?FORALL(L, list(int()),$$
$$reverse(reverse(L)) == L).$$

Another relevant study for this paper that uses property based testing for verification of the timing properties of an instant messaging server is presented in [14]. Using the Erlang language Hughes et al. generate test cases with a timing focus on an instant messaging server and compare the results of a property based approach with state-machine approach. However, this study has no focus on robustness testing and argues how property based testing can perform well for testing the timing aspects of a system. The timing parts here are mostly focused on timeout and not other timing aspects.

III. DESIGN OF ROBUSTEST

This section will discuss the design of the framework RobusTest for testing robustness with focus on timing properties. As discussed earlier there are seven patterns in ROAST with timing focus that are also used in RobusTest since RobusTest is a part of the ROAST framework. These are presented in this section and the first two are discussed in more detail. The other patterns are presented and discussed in less detail.

Figure 1 shows the overall structure of the RobusTest framework. The patterns discussed in this paper and [2] will give a structure to the requirements on the SUT that will be used by the testers to specify properties. These properties are then used by the test generator (TG) to generate test cases automatically. The number of test cases generated can be set manually by the tester. These test cases are then used by the test executor (TE) on the SUT. The results from execution of the tests are sent to the test oracle (TO) for assessment. Assessment is done based on the expected behavior in the properties, the results from running the test cases and the CRASH benchmarking framework.

[1] http://www.protest-project.eu/

 ISBN: 978-1-61208-168-7

In order to analyze the robustness of the SUT expected behavior and response should be specified. However, in some cases it might be enough to check whether running the test case has any critical effect on the SUT in form of crash or restart. The functionality to detect these kinds of failure is therefore built into the framework. Using the framework without specifying the expected behavior can in this way detect the most critical failures. This part of the framework is implemented using the CRASH benchmarking framework. CRASH is built in to the TO as the default oracle. However, CRASH can be supplanted by the expected behavior if provided in the property and it can even be disabled to generate some types of faults if that type of fault such as a system restart is an expected behavior of SUT.

In addition to the automated part of the oracle, it is possible to specify a concrete expected result for the test cases. The mechanism for generating test case and analyzing the result for the two first patterns is given below.

Another important aspect of RobusTest is the alignment of requirements and test cases. This is one of the main focuses of ROAST. Traceability between the requirements specified for the SUT based on the RobusTest patterns and the test cases generated by RobusTest is ensured through properties.

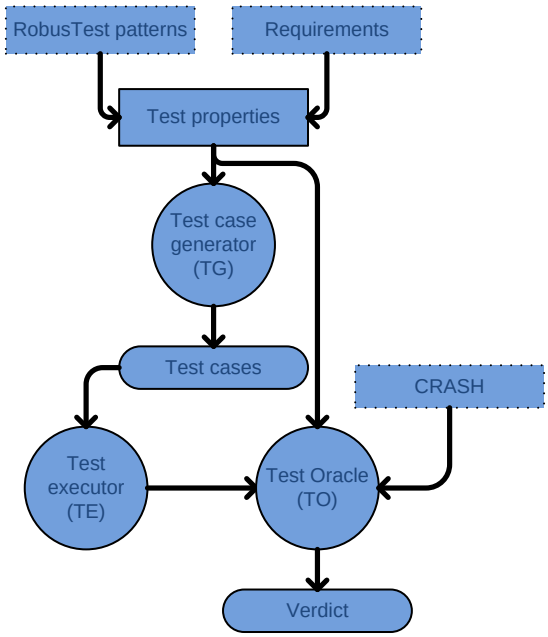

Fig. 1. RobusTest framework structure: The circles are parts of RobusTest and the rectangles are resources provided to or generated by RobusTest. The round rectangles are resources generated by RobusTest, the dotted rectangles are already available and the rectangle with a solid border should be specified by the tester.

A. Specified response to timeout and latency

This pattern specifies the expected behavior of the SUT in case an input or event is not received by an expected timeout deadline. To specify a property for this pattern the following factors can be specified: IUT, t_0, t_T, S_I, E, S_E The description of all the parameters in this section is given in Table I.

Given these parameters the timeout property can be specified. This specification can then be used to generate test cases that are run both before and after the timeout deadline to compare the behavior of the SUT in case of timeout with the case when the input is received on time.

S_I is used to specify what the test case needs to perform in order to have the SUT in the appropriate state for starting the test case. S_E is the expected SUT state after the test cases are run.

To generate test cases for this pattern not only the input arriving after the timeout deadline needs to be tested but in some cases test cases with input arriving before the deadline and very close to the deadline need to be generated to have a better understanding of the SUT's timing behavior. Test cases need to set the state of the SUT to S_I and simulate the input or event E. The SUT is then expected to be in state S_E after these actions.

To generate and analyze test cases for timeout the following algorithms are used. If the expected result S_E is not specified those steps with S_E will be neglected. The following is the description of three algorithms for three possible cases occurring when testing this pattern[2]

Input with timing before the timeout deadline:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t_T - \delta < t < t_T$ starting on t_0. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT (TE)
   - 4.1. CRASH $\rightarrow$ Fail (TO)
   - 4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   - 4.3. Valid output according to $S_E \rightarrow$ Pass (TO)

Input or event not received on deadline:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t > t_T$ starting on t_0. (TG)
3. On time t_T: (TE)
   - 3.1. CRASH $\rightarrow$ Fail (TO)
   - 3.2. If the behavior of the SUT is according to $S_E \rightarrow$ Pass (TO)

Input is sent after the deadline t_T:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t > t_T$ starting on t_0. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT. (TE)
   - 4.1. CRASH $\rightarrow$ Fail (TO)
   - 4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   - 4.3. Valid output according to $S_E \rightarrow$ Pass (TO)

The first algorithm analyzes the behavior of the SUT in cases where the input or event happens very close to the deadline. The purpose for this step is to ensure the correct behavior

[2]The letters in front of each step indicate what part of the framework is responsible for executing that step.

TABLE I
DESCRIPTION OF PARAMETERS THAT NEED TO BE SPECIFIED FOR TEST CASE GENERATION AND ANALYSIS

Parameter	Description
IUT	The set of interface(s) under test.
t_0	The reference time from when the timer should start counting. This is usually connected to an event in the form of an input received or an event in the execution environment.
t_T	The amount of time after the reference time until timeout occurs.
S_I	The initial state of the SUT at the reference time.
E	The expected input.
S_E	The expected behavior and response of the SUT. This might be as simple as the SUT not having any of the CRASH states. It can even be a specified expected behavior such as receiving a specific error message in case of timeout.
f	The maximum acceptable output or input frequency.
E_F	A follow up set of inputs that are dependent on E that will generate faults in the SUT if received before E.

of the SUT in general when the timeout is not expected to happen. The second algorithm assesses the behavior in case a deadline happens when the SUT is supposed to detect the deadline and take appropriate measures to ensure the rest of the functionality is not affected by the omitted input or event. The last algorithm is for generating an input or event after the deadline has been reached. Since the SUT does not have control on how the external parts behave and can not normally avoid them sending inputs or events, this part makes sure that receiving messages after deadline does not affect the correct functionality of the SUT.

A property for this pattern is specified in the following way[3]:

$$\forall t_T - \delta < t < t_T + \delta : setState(S_I, \{t_0\}), timeout \\ (Event(E, \{IUT\}), t) \rightarrow Expected(\{S_{Eb}\}, \{S_{Et}\}, \{S_{Ea}\})$$

where $setState(S_I, \{t_0\})$ specifies that the state of the SUT should be set to S_I at time t_0. $timeout(Event(E, \{IUT\}), t)$ specifies that the timeout happens at time t. $Expected(\{S_{Eb}\}, \{S_{Et}\}, \{S_{Ea}\})$ specifies that the expected behavior upon receiving the event E before timeout is S_{Eb}, after timeout S_{Et}, and the expected behavior in case timeout occurs is specified by S_{Et}.

B. Specified response to input with unexpected timing

This pattern represents cases where an input or event is received while it was not expected. The reason for too early input can be either a missing event that causes irregularities in the reception sequence or input that causes out of order events or inputs or the SUT not being ready to handle the event or input. To specify a property for this pattern the following factors are to be specified: IUT, t_0, S_I, E, S_E.

The difference between this pattern and the timeout pattern is that in this case there can be a need for more thorough understanding of the SUT as a whole. Modeling the SUT or at least parts of it is in some cases inevitable where we want automated generation of test cases for this pattern while in the case of timeout it can be enough to specify a property on a specific interface. The reason is that the initial state is more complex and simulating missing inputs is a more troublesome work than generating a timeout.

To generate test cases the SUT is configured to S_I after which the event or input E occurs. In the same manner as the previous pattern this needs to be tested on both sides of the deadline. S_E specifies in what state the SUT needs to be after this test in each case.

To generate and analyze test cases for inputs and events occurring with unexpected timing and specially too early inputs the following algorithms are built into RobusTest. Similar to the previous pattern, if the expected result S_E is not specified steps including S_E will be ignored by RobusTest and the default oracle (CRASH) is used.

Input with timing after t_0:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t_0 < t < t_0 + \delta$. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT. (TE)
   - 4.1. CRASH $\rightarrow$ Fail (TO)
   - 4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   - 4.3. Valid output according to $S_E \rightarrow$ Pass (TO)

Input or event has been received before the starting time t_0:

1. Set the SUT to state S_I. (TG)
2. Generate a random delay $t < t_0$. (TG)
3. Send a valid input according to the description in E to IUT. (TG)
4. Wait for output from IUT for a certain amount of time for output: (TE)
   - 4.1. CRASH $\rightarrow$ Fail (TO)
   - 4.2. Invalid output according to $S_E \rightarrow$ Fail (TO)
   - 4.3. Valid error message output according to $S_E \rightarrow$ Pass (TO)
   - 4.4. No output received $\rightarrow$ Pass (TO)
5. When IUT is ready to receive messages if the state of the SUT is incorrect $\rightarrow$ Fail

The first algorithm checks the behavior of the system for inputs received a short while after the SUT is ready to process inputs, while the second algorithm checks the behavior when

[3]Parameters inside {} are optional in RobusTest.

 ISBN: 978-1-61208-168-7

the SUT is not ready yet to receive any input. This way it is possible to check the behavior of the SUT in cases which should not happen and timings that are very close to the acceptable limit.

A property for this pattern is specified in the following way:

$$\forall t_0 - \delta < t < t_0 + \delta : setState(S_I, t), earlyEvent \\ (Event(E, \{IUT\}), t) \rightarrow Expected(\{S_{Eb}\}, \{S_{Ea}\})$$

where $setState(S_I, t)$ specifies that the state of the SUT should be set to S_I at a time before t. and the event E should be generated and sent to IUT at time t. $earlyEvent(Event(E, \{IUT\}), t)$ specifies that the event or input $Event(E, \{IUT\})$ is sent to the SUT at time t. The expected behavior in that case is S_{Eb} or S_{Ea} depending on whether t is before or after t_0.

C. High input frequency

This pattern tests the behavior of the system when the input frequency is high and higher than what the system or module can handle given the resources and processing power. These tests are specified generically and since the frequency can be dependent on what platform and hardware the SUT is running on, RobusTest will increase the frequency of the input gradually until it comes to a state where the SUT can not handle the work load. At that point the SUT is expected to behave in accordance with the specification without crashing. To specify a property for this pattern the following factors are to be specified: IUT, S_I, f, S_E.

Test cases for this pattern start with setting the initial state S_I to the SUT and then generating inputs and events with higher frequency than f to check the behavior of the SUT in that case.

D. Lost events

This pattern discusses robustness issues that occur when an event is expected but is missing. Although this pattern is not directly a timing pattern it usually has a close correlation to the timing issues as seen in the first two patterns discussed above. The following parameters are to be specified in order to generate test cases for this pattern: E, S_I, E_F, S_E.

The test cases in this pattern aim to test the robustness of the SUT when an important event is lost or ignored. In order to simulate this situation the event E is not created or created in an erroneous manner in the test cases.

E. High output frequency

This pattern discusses the robustness issues resulted from high output frequency of a module and its consequences for the whole system or other systems. In the same manner as high input frequency to specify a property for this pattern the following factors are to be specified: IUT, S_I, f, S_E.

The high input from one module or unit can lead to missing events and messages or overloading other part of the SUT that might lead to robustness issues. Since the framework focuses mostly on black box testing it is not always possible to generate tests for this pattern. Although if the structure of the SUT allows this, it can be simulated by limiting the output channels of the system or the input of the receiving unit.

F. Input before or during startup, after or during shut down

The focus of this pattern is to test the behavior of the system towards inputs received during startup or shut down. The test cases for this pattern can be generated in the same way as for input with unexpected timing. The main focus here is though to check whether inputs during startup and shut down can change the state of the SUT in a way that causes irregularities after the SUT has started properly.

G. Error recovery delays

This pattern focuses on the state where the SUT is recovering from an error or has degraded functionality. In the same way as the previous pattern inputs received during error recovery needs to be handled in a specific way. Although the SUT is running during these phase but the functionality is degraded and the transmitting parts and modules need to be aware of that and the received input needs to be handled properly according to the requirement specification for the SUT.

IV. Conclusion

This paper has discussed a framework called RobusTest for testing robustness properties of software systems. In the current version the framework mostly focuses on timing issues that lead to robustness vulnerabilities. Testing is done using properties that are written to specify an expected behavior from the system. These properties are then used to generate test cases automatically. The properties are even used to analyze the results from executing the test cases on the system and is in this way used as an oracle for the behavior of the system.

In this paper, seven properties from ROAST, which is a framework for specifying and testing robustness properties in software, are introduced. RobusTest is a part of ROAST with focus on the testing part. In the current study, the patterns with focus on timing issues were presented and the properties extracted from each pattern were discussed in more detail. After writing properties based on the patterns there is a clear link through those properties from the requirements elicited and specified by ROAST and the test cases generated by RobusTest.

V. Current and Future Work

Given the structure presented in this paper, the framework is being built for Java as an extension of JUnit. However, this framework can be implemented in any programming language and the Java framework is a case study for proof of concept. The idea is to extract commonalities for requirements in the same patterns and wrap them in RobusTest for testing those requirements. In this manner, there will be a common interface with built in functionality such as generation of test cases and automated CRASH oracle that can be used to test a specific type of requirement. This JUnit extension is then to be tested initially for protocol testing in a communication protocol with timing restrictions.

 ISBN: 978-1-61208-168-7

This paper aimed to present the idea and structure of the framework. However, a small evaluation of the concept was performed by testing parts of the framework on two simple programs. The generated test cases were able to identify the faults that were injected in the programs. However, since this evaluation was very small compared to the size of the framework, it is not possible to draw any conclusion regarding the validity of RobusTest. A more thorough evaluation on two large open source systems is currently in progress and will be published in our future publications.

Another important next step is to look at other patterns in ROAST that are not currently included in RobusTest. Testing for input with invalid value and testing unexpected conditions in the execution environment are to be added to RobusTest in order to have a complete structure and a clear link from the requirements in those patterns and the test cases.

REFERENCES

[1] *IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990*, 1990.

[2] A. Shahrokni and R. Feldt, "Towards a Framework for Specifying Software Robustness Requirements Based on Patterns," *Requirements Engineering: Foundation for Software Quality*, pp. 79–84, 2010.

[3] J. DeVale, P. Koopman, and D. Guttendorf, "The Ballista software robustness testing service," in *Testing Computer Software Conference*, 1999.

[4] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Software-Practice & Experience*, vol. 34, no. 11, pp. 1025–1050, 2004.

[5] A. K. Ghosh, M. Schmid, and V. Shah, "Testing the robustness of Windows NT software," in *Proceedings of the 9th International Symposium on Software Reliability Engineering, 4-7 Nov. 1998*, ser. Proceedings of the 9th International Symposium on Software Reliability Engineering (Cat. No.98TB100257). Los Alamitos, CA, USA: IEEE Computer Society, 1998, pp. 231–235.

[6] M. Dix and H. D. Hofmann, "Automated software robustness testing - static and adaptive test case design methods," in *Proceedings of the 28th Euromicro Conference, 4-6 Sept. 2002*, ser. Proceedings of the 28th Euromicro Conference. Los Alamitos, CA, USA: IEEE Computer Society, 2002, pp. 62–66.

[7] J. Fernandez, L. Mounier, and C. Pachon, "A model-based approach for robustness testing," *Testing of Communicating Systems*, pp. 333–348, 2005.

[8] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proceedings of the 16th Symposium on Reliable Distributed Systems, 1997.*, 1997, pp. 72–79.

[9] D. Clarke and I. Lee, "Automatic generation of tests for timing constraints from requirements," in *words*. Published by the IEEE Computer Society, 1997, p. 199.

[10] G. Fink and M. Bishop, "Property-based testing: a new approach to testing for assurance," *SIGSOFT Softw. Eng. Notes*, vol. 22, pp. 74–80, July 1997.

[11] G. Fink and K. Levitt, "Property-based testing of privileged programs," in *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*. IEEE, 1994, pp. 154–163.

[12] G. Fink, C. Ko, M. Archer, and K. Levitt, "Towards a property-based testing environment with applications to security-critical software," in *Proceedings of the 4th Irvine Software Symposium*, vol. 39, 1994, p. 48.

[13] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 35, pp. 268–279, September 2000.

[14] J. Hughes, U. Norell, and J. Sautret, "Using temporal relations to specify and test an instant messaging server," in *Proceedings of the 5th Workshop on Automation of Software Test*, ser. AST '10. New York, NY, USA: ACM, 2010, pp. 95–102.

 ISBN: 978-1-61208-168-7