Structural Unit Testing as a Service with PathCrawler-online.com

Nikolai Kosmatov Nicky Williams Bernard Botella Muriel Roger
CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette, France
E-mail: firstname.lastname@cea.fr

***Abstract*—PathCrawler is a concolic test generation tool for the structural unit testing of C code. This paper describes the PathCrawler-online.com testing service. The user submits C source files and test parameters, and the server generates test-cases and coverage information. The service is mostly used today as a novel alternative to providing a demonstration version for students and potential users to install. However, PathCrawler-online can also be seen as a prototype for Testing as a Service and a first step towards Software Testing in the Cloud. In this spirit, we discuss the issues raised by our two-year experience with PathCrawler-online.**

I. Introduction

Software testing accounts for up to 50% of the total cost of software development. Automatic testing tools can provide an efficient alternative to manual testing and reduce the cost of software testing. In particular, state-of-the-art tools can now automate the generation of test inputs to ensure structural coverage of source code.

Traditionally, the software developer buys a licence for an automatic test tool and installs and runs it on their own machine. The tool provider must port the tool to the different platforms likely to be used by software developers. Bugs or undesirable behavior must be reported by the user to the tool provider, who must then reproduce them, correct them and send updates to the user, who must then install them.

The paradigm of *Testing as a Service* (TaaS) brings several benefits, both to the user and to the tool provider (who becomes a test-service provider). Firstly, TaaS removes the need for users to install the testing tool. Instead, the test-service provider installs it on the servers which should vary less than the users development platforms. Only the user interface may need to be integrated into different software development environments, so the porting effort is considerably reduced for the test-service provider. Moreover, the test-service provider has a centralized record of the history of test sessions which revealed bugs in the tool, which ensures that they can easily be reproduced. The test-service provider can easily update the tool whenever necessary and this is completely transparent for the user. Finally, the user can test on demand so does not need to worry about amortizing the fixed capital costs of the software licence and extra hardware to run the tests.

This paper presents PathCrawler-online.com, the first online service for automatic structural unit testing of C programs, and discusses various issues encountered in our two-year experience of operation. Section II briefly presents the PathCrawler tool. Section III describes the PathCrawler-online testing service. Section IV reports on our experience feedback. Sections V and VI provide related work and conclusions.

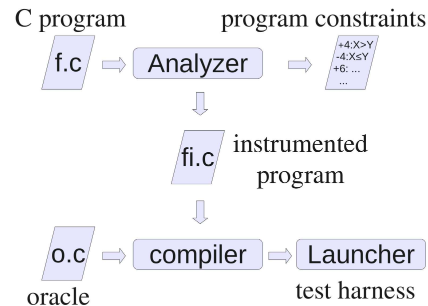

Figure 1. Stage 1 of the PathCrawler method

II. The PathCrawler tool

PathCrawler [1], [2], [3] is an automatic generator of test-case inputs to ensure structural coverage of C source code.

Structural unit testing is widely used in industrial verification processes of critical software. In critical systems processes where structural testing is required by the development norm, manually creating tests from the specification often fails to achieve complete satisfaction of the coverage criterion. In this case, automatic methods help to reach the objectives which are not covered and provide corresponding path conditions that may be used to refine the specification if needed. They may also determine whether the objectives which are not yet covered are really feasible.

However, even when the development process does not impose any structural testing activity, the use of an automatic structural test generation tool is a way to increase the quality of the software with a very low cost overhead. Indeed structural unit testing can accompany code development in order to detect bugs as early as possible. These may be functional errors revealed by an oracle which defines the specific properties of the tested software. But PathCrawler's exhaustive exploration of the source code can also be used to demonstrate the absence of certain runtime errors or anomalies that may indicate a potential bug (or cause future maintenance problems) in any program. The online tutorial[1] gives examples of how PathCrawler can be used with an oracle provided by the user, to search for runtime errors or

[1] http://pathcrawler-online.com/tutorial/tutorial_2012.pdf

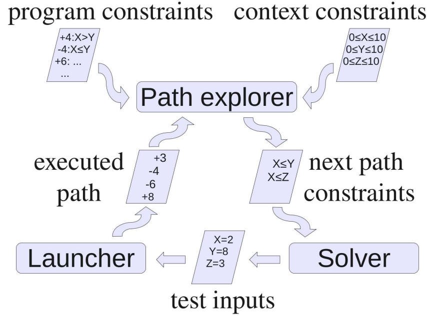

Figure 2. Stage 2 of the PathCrawler method

anomalies or even to *cross-check* the implementation against another implementation or specification in C.

PathCrawler is based on a method [2] often referred to in the literature as *concolic* or *dynamic symbolic execution*. It is based on the instrumentation and execution of the code under test and on constraint solving. Test generation is a cyclical process alternating execution of the previous test-case and the search for the following case. This has the incidental advantage that an oracle can be inserted into the loop to provide the results of the cases as they are generated.

This method contains two main stages. The first stage is illustrated by Figure 1. The user submits the complete ANSI C code of the function to be tested and all called functions in files designated in Figure 1 by `f.c`. The Analyzer module automatically instruments the program and extracts the semantics of each statement in the form of symbolic constraints. The instrumented version, denoted by `fi.c`, contains the original source code of `f.c` enriched with path tracing instructions. The compilation of this instrumented version provides an executable test harness called Launcher. The role of Launcher is to run the program under test with a given input and record the trace of the program path that was activated. After instrumentation, the user may modify the default test context. In particular, the user may provide an optional oracle program, denoted `o.c` in Figure 1, that checks if the result of the program executed by a given test case is correct or not. When provided, the oracle is loaded into Launcher, and called after the execution of a test case to produce a verdict on the results.

Figure 2 illustrates the second stage, which uses the Launcher and the intermediate program representation in constraints obtained at the first stage. At this stage, the three modules shown in Figure 2 are used one after another in a loop. At the beginning of each iteration, Path Explorer determines the next (partial) program path to be covered (for the first iteration, this is the empty path). Path Explorer sends to Solver the path constraints corresponding to the path to be covered. Solver is a finite-domain constraint solver. It generates test inputs which satisfy the constraints (and so activate the partial path) and sends them to Launcher. Launcher executes the program under test on these inputs and sends the trace of the executed program path to Path

Test session summary

General test session information

Function under test: Bsearch

Coverage criterion: all feasible paths

Termination status: normally

Test session statistics

Total test session duration:	1 sec.
Number of generated test-cases:	17
with verdict "success":	10
with verdict "failure":	7
with verdict "unknown":	0
violating user assertion:	0
Number of treated partial paths:	70
covered:	17
infeasible:	53
interrupted by timeout:	0

Figure 3. Summary of a test session with PathCrawler

Explorer, which starts the next iteration. Solver will fail to generate a test case if it can show that the program path to be covered is infeasible (i.e. the constraints are unsatisfiable) or if the constraint problem is so difficult that Solver does not find a solution in a given time, provoking a timeout. Indeed, constraint resolution (satisfiability) is NP-complete and it is difficult to predict the time needed to solve the path constraints. When Solver fails to generate a test case, the method skips the concrete execution step with Launcher and goes directly to Path Explorer for the next path choice. When Path Explorer has no more paths to cover, all paths are covered and test generation stops.

III. PathCrawler-Online testing service

In this section, we describe the inputs and generated results of the PathCrawler-online testing service.

A. Inputs

PathCrawler-online takes the following input data:

- an archive with complete, possibly multi-file, compilable C source code,
- name of the main file and the function under test,
- test parameters including a precondition and test generation strategy (coverage of all feasible paths or k-*path*: just those with a limited number of loop iterations),
- C source code of an oracle.

An online interface allows the user to customize test parameters and the oracle. Alternatively, the user can submit customized C oracle and test parameters in XML format within the submitted archive.

B. Outputs

During a test generation session, PathCrawler-online generates test-cases and coverage information for the submitted C code and parameters. Test session results include:

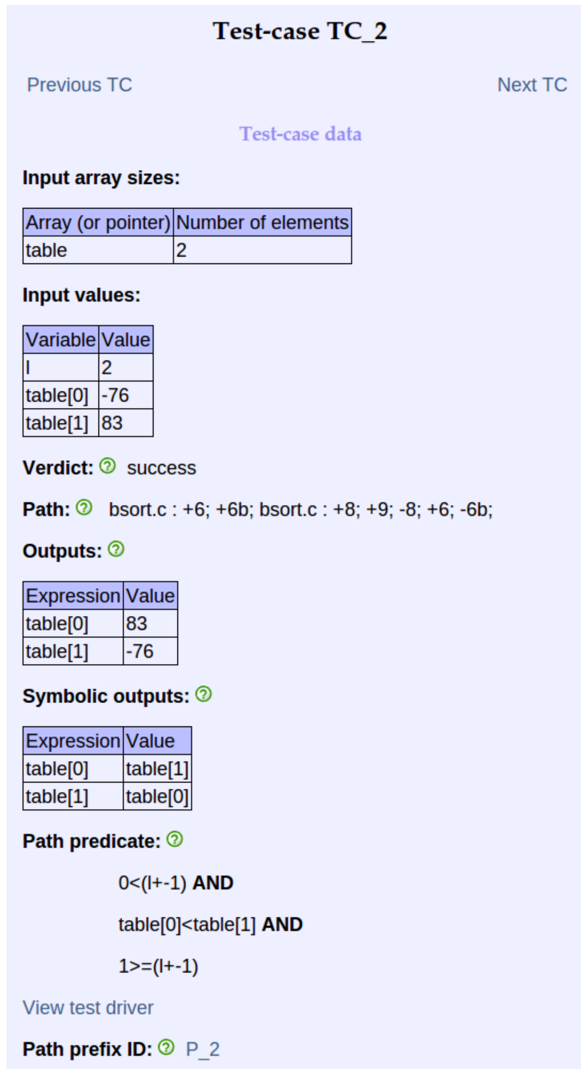

Figure 4. A test-case automatically generated by PathCrawler

- test session statistics and coverage information (cf Figure 3),
- test-cases (cf Figure 4),
- test drivers,
- explored path details.

The results are generated in XML and HTML formats that can be easily integrated into a user testing environment or visualized in a browser, while test drivers are ready-to-compile C files. The XML outputs are not available in the free evaluation version.

Test session statistics and path coverage information (cf Figure 3) include test session duration, the number of explored paths with the number of paths of each status (e.g. covered, infeasible, violating a user assume statement, interrupted by a timeout), the number of generated test-cases with the number of different oracle verdicts (e.g. success, failure, violating a user assertion, unknown). Branch coverage of generated test-cases is also computed (and will be part of the test session summary in the upcoming release).

A test-case (cf Figure 4) includes *input values* (for input array/pointer sizes and variable values), *output values* and the *oracle verdict*. Moreover, it provides the complete program *path* executed by the test-case, as well as *path prefix*, *path predicate* and *symbolic outputs*. *Explored paths* information provides the list of all program paths explored by PathCrawler, including infeasible ones, and their path predicates.

IV. Our experience of PathCrawler-online

A. PathCrawler-online as an evaluation and teaching aid

The primary motivation for developing PathCrawler-online was to provide a way for researchers, students and potential users from industry to try out and evaluate the tool and get a first experience of automatic structural unit testing. Indeed, we have successfully used it for classes on structural testing [4] and tutorials at several international events, such as TAP 2012 [5], TAROT 2012, QSIC 2012 [6] and ASE 2012. It is also used in courses taught in other countries.

However, PathCrawler-online can also be seen as a prototype for TaaS and a first step towards Software Testing in the Cloud. To develop this aspect, we start by discussing several issues raised by our experience in running PathCrawler-online.

B. Confidentiality of test artifacts

An online testing service requires the user to upload their implementation and an oracle. In the case of PathCrawler, the user uploads the source code, which is then compiled by the server. In the case of automatic test-case generation from models, we can suppose that the user would have to upload a model and an executable version of the implementation. In all cases, a major problem of confidentiality is apparent. Indeed, since PathCrawler-online came into service, we have had numerous comments from industrial partners that they could not possibly envisage uploading their source code onto our server. Many feel the same way about the detailed models often used in model-based testing. Along with the oracles, these artifacts represent an important part of the worth of the company. The confidentiality issue was also identified in other studies [7], [8], [9].

However, as we pointed out in the Introduction, one of the advantages of TaaS as exemplified by PathCrawler-online is that in the case of a bug, all the information necessary to reproduce it is present on the server. In many cases, the user does not have to intervene at all, abnormal behavior is automatically detected and the session data is saved for future analysis.

We see from the experience of PathCrawler-online that TaaS cannot succeed unless the protection of the test artifacts is sufficient to reassure potential users but that, if possible, this protection should allow the use of these artifacts by the service provider for debugging purposes only.

It would be quite possible to provide increased reassurance for clients by improving confidentiality in PathCrawler-online in the following ways:

- Cryptography: use of protected channels (such as https) to transfer the user's artifacts to the server.
- Anonymity: destruction of the identity of the client of each session in the records we save for debugging.
- Obfuscation: automatic irreversible renaming of variables, functions etc. (but without changing the control structure) in the user's source code in the records we save for debugging.

A legally binding undertaking not to disclose users' artifacts could also help.

C. *Execution of the implementation*

Among online services, test services such as PathCrawler-online present a particular danger: the potential corruption of the server due to the execution of unknown code submitted by the user (cf Section II). This may be an involuntary result of bugs in the user's implementation, or due to a malicious attempt to access data on the server or bring it down. PathCrawler-online is essentially protected in the four following ways from this danger (see also [10]).

- PathCrawler-online runs Stage 2 (Figure 2) in a virtual machine in order to ensure that it is isolated from other data and so that a program crash cannot impact the server. The virtual machine protects from unauthorized access to the files outside, but a limited information exchange with external files has to be allowed in some way in order to transfer into the virtual machine the files related to the desired test generation task and to extract the results. The privileges of the users executing the virtual machine and the program under test inside are carefully checked and restricted.
- Moreover, by restricting the resources used by an individual test session, PathCrawler-online prevents attacks based on saturation of the memory or fork bombs (without needing to completely forbid the fork operation).
- PathCrawler-online takes advantage of the necessary analysis of the source code to reject programs containing potentially dangerous instructions, such as assembly code.
- Finally, PathCrawler-online deactivates network access in the virtual machine. In our opinion, the number and variety of attacks using the Internet today would make it very difficult to provide a reliable public testing service similar to PathCrawler-online for software using the Internet.

The issues described above are related to those encountered in commercial PaaS clouds where the user code is executed in the cloud. For example, Google App Engine [11] also uses virtualization to secure execution, but its limitations are different from PathCrawler-online. Execution of C code is forbidden in Google App Engine, while PathCrawler is designed for testing C code. In Google App Engine, applications cannot write to the file system in any of the runtime environments, while PathCrawler-online allows read-write access to files inside the virtual machine because it is necessary e.g. to write test generation results. Google App Engine forbids creating sub-processes while PathCrawler-online allows a limited number of sub-processes to be created by the program under test. Google App Engine also restricts access to and from the Internet, whereas PathCrawler-online completely forbids it.

D. *Target Platform*

In order to provide a complete testing service, it is necessary to take into account the target platform (hardware, operating system, compiler,...) of the implementation. Indeed, the behavior of the tested program may vary slightly from platform to platform due to differences such as word length or endianness. If the program has been developed on a different platform to the target platform then these subtle differences can be a source of bugs which will not be detected unless testing is performed on the target platform.

PathCrawler-online does not currently propose testing on different target platforms. It is up to the user to recover the test inputs or drivers and re-run the tests on the target platform. However, testing on different target platforms could be proposed without changing the fundamental design of PathCrawler-online. The PathCrawler method described in Section II would have to be extended in three places.

Firstly, the semantics of the C instructions used in the Analyzer would have to take account of platform-specific features such as word length. It is already possible to configure the Analyzer to do this for several platforms.

Secondly, it would be necessary to select the appropriate compiler when compiling the Launcher.

Thirdly, the Launcher would need to be run on the target platform or a simulator. The architecture of PathCrawler-online allows the Path Explorer and Solver modules to be run on a different platform to the Launcher because the Launcher is a separate process.

E. *Pricing*

PathCrawler-online currently provides a restricted service for free. It may be desirable to maintain a level of use which is free, in order to tempt user into trying the service. In order to overcome the restrictions of the free service and take advantage of the cloud, the user would have to pay. This raises the question of the pricing strategy.

Because of the nature of structural test-case generation, the user who just pays for the CPU time consumed by the test session has no way of knowing in advance the cost of the test session. This is because the CPU time depends on the number of test-cases necessary to achieve the desired coverage, but also on the number of infeasible paths and the difficulty of the corresponding constraint problems.

Users may prefer to pay a fixed price per test. From their point of view, the more they pay, the more their implementation is tested. However, the supplier of the service must then estimate the risk that certain implementations contain a great many infeasible paths (which don't result in a test-case) or contain paths that pose a constraint problem which takes a long time to resolve.

The pricing of a similar test service is discussed in [12]. They propose a cost per marginal increase in coverage which would be higher as the coverage increases, reflecting the fact that the last percentage points of coverage often require more CPU time than the first. They also propose the alternative of a price per bug found but this has the disadvantage of being completely unpredictable for the test-service provider.

F. Performance

In order to avoid denial-of-service problems, PathCrawler-online restricts the resources which can be used during any one test session. These resources include disk space, memory and execution time. If PathCrawler-online were to be extended to Testing in the Cloud, then almost all restrictions could be lifted.

PathCrawler-online proposes a unit-test service so avoids the performance problems posed by testing large systems. However, structural coverage, and particularly path coverage, requires many test cases and their automatic generation can demand a lot of resources. Much research has been carried out in the optimization of automatic structural test-case generation. Sometimes path coverage is not possible and branch or condition coverage is preferred. However, a complementary angle of attack is to make use of the increased computing resources available in the cloud.

PathCrawler-online currently takes advantage of the multi-core architecture of the server to run several test sessions in parallel but each test-session is run sequentially. Deployment on the cloud offers the possibility of parallelization of each test session, to speed up testing, and the test-case generation technique used by PathCrawler seems particularly well adapted to parallelization.

Parallelization could be implemented at at least three levels:

- Concolic test generation is based on an alternation between execution of generated cases (Launcher) and the search for new cases (Path Explorer and Solver). The two processes currently run synchronously, one after the other, but they could be run in parallel.
- Concolic test generation performs an exploration of the tree of execution paths and this exploration could also be parallelized. This is the focus of the Cloud9 project, which is based on a generation method similar to PathCrawler's. Methods for load balancing, tree exploration strategies to speed up coverage and techniques to reduce redundant computation are described in [13].
- PathCrawler uses constraint solving over finite domains, which also offers interesting perspectives for parallelization.

Another potential benefit of TaaS described in [13] could also be transposed to PathCrawler. When many users make calls to the same libraries, the test results for these calls can be stored. In the case of PathCrawler, the previously discovered path predicates of the library functions could be used to avoid *path explosion*. This occurs when all combinations of paths through the tested function with those through called functions must be explored. It could be avoided by using the stored path predicates to predict when further exploration of the called function cannot increase coverage of the tested function.

V. Related work

Most current commercial use of software testing in the cloud [14] consists of providing test execution platforms for web, desktop or mobile applications, with some providers also allowing access to testing tools.

A non-commercial example of work with a similar goal (but not publicly available as an online service to our knowledge) is D-Cloud [15], [16]. This is a large-scale software testing environment which has been developed to ensure dependable distributed systems. D-Cloud can use computing resources provided by the cloud to execute several test-cases simultaneously, and thus accelerate the software testing process.

PathCrawler-online proposes a completely different service: automatic structural unit testing of C code based on the automatic generation of test-case inputs.

PathCrawler-online is similar to the service envisaged in the Cloud9 [13] project: a cloud-based testing service based on the KLEE testing tool. Cloud9 implements parallel symbolic execution for computer clusters operating on public cloud infrastructures such as Amazon EC2 and clusters running cloud software like Eucalyptus. The underlying test-generation technique is similar to that used by PathCrawler but Cloud9 is not focussed on unit testing and treats binary code rather than C source code. The Cloud9 software is publicly available but there does not seem to be a publicly available service based on the software.

The York Extensible Testing Infrastructure (YETI) [17] is an automated random testing tool for Java with the ability to test programs written in different programming languages. A cloud implementation of YETI was shown to significantly improve the performances and to solve potential security issues (by executing Java classes on clean virtual machines). However, there does not seem to be a publicly available TaaS based on YETI.

VI. Conclusion

We explained in the Introduction why automatic testing tools are good candidates for TaaS. We also explained in Section II how PathCrawler automates structural unit testing.

Year	Total test sessions	Examples	User code
2011	1237	52%	48%
2012	6869	32%	68%

Figure 5. Total number of test sessions and the rate of test sessions for predefined examples and for user submitted code

Unit testing is one of the three types of testing identified in [9] as suitable for the cloud. They argue that the benefits of testing in the cloud also depend on three characteristics of the application under test: test cases must be independent from one another, the operational environment of the test process must be well known and the application must have a programmatic interface so that testing can be automated. In the case of unit testing as performed by PathCrawler, these characteristics are indeed present. Unit testing requires the user to encapsulate each software unit, including all called functions, so that it can be run independently. To ensure the detection of functional defects, the user must also provide an oracle. Indeed, the user effort is shifted from inventing test-cases to formalizing the calling context and oracle of each tested unit.

The PathCrawler-online test service can be used to continually accompany the developer during code development. This is the programmer's sidekick rôle for TaaS described in [12]. Path coverage requires many tests, which may take a long time to construct. However, path coverage can ensure detection of bugs which might be missed by other criteria and it is the only test criterion which can be used to guarantee the absence of certain runtime errors or anomalies or do cross-checking. This is why we think that path testing is a good candidate for deployment on the cloud in order to gain access to increased resources. We explain in Section IV why the test-generation method used by PathCrawler should lend itself well to distribution on different machines.

To our knowledge, PathCrawler-online was the first test-case server of its kind freely available online. It has been in operation for two years now and we have observed a trend in its use that is illustrated by the statistics in Figure 5. Firstly, the number of users is increasing, and the number of test sessions has reached 700 per month. Secondly, early use was often restricted to running the predefined examples proposed by the server but recently this has changed. There has been a significant increase in the number of uploads of the user's own code, including some quite large programs.

In other words, we have the impression that the use of PathCrawler-online is evolving from an evaluation version and a teaching aid towards a test service. This use is clearly restricted today by concerns about the confidentiality of the source code, and by the limits on resources that we are obliged to impose as long as all sessions are run on one server. This is why we are studying how our experience of providing a limited testing service for free could help in the design of a commercial cloud-based deployment.

References

[1] N. Williams, B. Marre, and P. Mouy, "On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing," in *ASE 2004*, pp. 290–293.

[2] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: automatic generation of path tests by combining static and dynamic analysis," in *EDCC 2005*, pp. 281–292.

[3] B. Botella, M. Delahaye, S. Hong-Tuan-Ha, N. Kosmatov, P. Mouy, M. Roger, and N. Williams, "Automating structural testing of C programs: Experience with PathCrawler," in *AST 2009*, pp. 70–78.

[4] N. Kosmatov, N. Williams, B. Botella, M. Roger, and O. Chebaro, "A lesson on structural testing with pathcrawler-online.com," in *TAP 2012*, pp. 169–175.

[5] N. Kosmatov and N. Williams, "Tutorial on automated structural testing with PathCrawler. Extended abstract," in *TAP 2012*, p. 176.

[6] N. Williams and N. Kosmatov, "Structural testing with PathCrawler. Tutorial synopsis." in *QSIC 2012*, pp. 289–292.

[7] L. M. Riungu, O. Taipale, and K. Smolander, "Software testing as an online service: Observations from practice," in *STITC 2010*, pp. 418–423.

[8] ——, "Research issues for software testing in the cloud," in *CloudCom 2010*, pp. 557–564.

[9] T. Parveen and S. R. Tilley, "When to migrate software testing to the cloud?" in *STITC 2010*, pp. 424–427.

[10] N. Kosmatov, *Software Testing in the Cloud: Perspectives on an Emerging Discipline*. IGI Global, 2012, ch. XI: Concolic Test Generation and the Cloud: Deployment and Verification Perspectives, pp. 231–251.

[11] Google, "Google App Engine Documentation," 2012, https://developers.google.com/appengine/.

[12] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," in *SoCC 2010)*, pp. 155–160.

[13] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," *Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2009.

[14] L. Riungu-Kalliosaari, O. Taipale, and K. Smolander, "Testing in the Coud: Exploring the Practice," *IEEE Software*, vol. 29, no. 2, pp. 46–51, 2012.

[15] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-Cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology," in *CCGrid 2010*, pp. 631–636.

[16] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato, "Large-scale software testing environment using cloud computing technology for dependable parallel and distributed systems," in *STITC 2010*, pp. 428–433.

[17] M. Oriol and F. Ullah, "Yeti on the cloud," in *STITC 2010*, pp. 434–437.