Stuck-At-Faults Tester as a Web-Service

N.A. Shalyapina <nat.shalyapina@gmail.com>
A.A. Zaytsev <z.sania@mail.ru>
S.V. Batratskiy <pride080993@gmail.com>
M.L. Gromov <maxim.leo.gromov@gmail.com>
Tomsk State University,
634050, Russia, Tomsk, Lenin av., 36

Abstract. In this paper, we tell about a web-service we would like to develop. There are two goals we aim at, when developing this service. The first one is to give researchers a platform, where they could conduct preliminary experiments with different methods of test generation for digital circuits, in order to check different ideas. The second one is an opportunity to share implementations of new developed methods "on-the-fly". The web-service development procedure was splitted into three stages: the architecture design, a light version implementation and the actual implementation. This paper tells about first two stages. There are two types of web-service architectures – with monolithic kernel and with microkernel – and our architecture has the properties of both types. The intention was to have monolithic kernel, since the desired functionality is not that hard to implement. However, the property of being extensible by implementations of new methods implies that part of the functions (namely the methods implementations) should be designed as separate sub-services. The light version implementation was done for the only method: method of fault domain enumeration for the stuck-at-faults fault model. It proved that the designed architecture is viable. However, some issues with the architecture were discovered. A mechanism of on-the-fly deployment of a new method is unclear, since it is not obvious, how to satisfy possible dependences of the implementation. Also, the architecture does not follow the classical web-service design: the service has states, that should not be, if a service is intended to be the classical one. The resolution of these issues is left for the future.

Keywords. Stuck-At-Faults, Combinational Logical Circuit, Web Service, Test.

DOI: 10.15514/ISPRAS-2018-30(1)-3

For citation: Shalyapina N.A., Zaytsev A.A., Batratskiy S.V. Gromov M.L. Stuck-At-Faults Tester as a Web-Service. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 1, 2018, pp. 41-54. DOI: 10.15514/ISPRAS-2018-30(1)-3

1. Introduction

The modern world is hard to imagine without a variety of digital circuits. Even if a device does an analogue job, say, play music, most probably somewhere inside this device there is a digital circuit. Like any other utilitarian objects, each digital circuit is expected to perform particular work. Therefore, a problem, if this work is *done correctly*, is an actual problem. This problem appears in almost all stages of a circuit's life cycle: development, production and use.

There are many aspects of *doing a work right* by a circuit: correct functions are implemented, the circuit works quickly enough to be used, the circuit consumes limited amount of energy, etc. For example, when it comes to the correct implementation of desired functions, they usually talk about *functional testing* of a circuit [1, 2].

There are many different methods for test generation [1], and every day new ones appear (see, for example, [2]). In addition, new fault models for circuits are being developed, that can describe faults, which occur in real life, in some sense better. In such a situation, at least two questions arise. Is a new method better or worse than already existing methods? Can existing methods detect faults, introduced by a new fault model? Experiments can answer these questions. To conduct at least the preliminary experiments, a researcher should have a "rapid" access to already existing methods of test generation for digital circuits, as well as the opportunity to share with other researchers his or her new method.

Apparently, a web-service can provide such an access. However, for the best of our knowledge, a web-service that provides resources to construct tests for digital circuits using different methods does not exist. Therefore, in our research group under the leadership of Nina Yevtushenko, within the RSF project № 16-49-03012, it was decided to implement such a service.

We splitted the development of the service into several stages. The first stage is to design architecture of the service. The second one is to implement a light version of the service, to see on practice, what hardships we shall meet during the main implementation, what decisions we shall need to do and what white spots there are in the architecture. And the final stage is the actual implementation of the service.

This paper describes the first two stages of the development: architecture design and a light version implementation of the service. For the light version of the service, we have chosen a classical fault model – stuck-at-faults [1] – and a classical method of test derivation – fault domain enumeration [1]. The stuck-at-faults fault model supposes that the only faults that can occur in the circuits are shorts to the ground or to the power wire. The fault domain enumeration is the method when every possible implementation of the circuit (defined by the fault model) is enumerated and input stimuli are searched, which can show the difference between the right circuit (the specification) and the enumerated one.

The rest of the paper is organized as follows. In Section 2, the description of the designed architecture of the web-service is given. Section 3 is devoted to the light version of the service implementation; it includes the short description of the enumeration of the fault domain method for the single stuck-at-faults fault domain. Section 4 concludes the paper. In Section 5, we provide acknowledgements for the paper support.

2. The Architecture of the Web-Service for Tests Derivation

When designing a web-service one should make two main choices: *monolithic kernel* or *microkernel* and *SOAP* [4] or *REST* [5].

2.1. Monolithic Kernel vs Microkernel

A monolithic kernel supposes, that all logics of a service are implemented as a single application, which has outside only data sources (databases). This approach promises fast and easy deployment procedure together with better performance.

On the other hand, microkernel has always been considered as a more flexible solution: each function of the big service is implemented as a stand-alone micro-service. This approach provides natural mechanisms of functionality extensions and updates (including security updates).

For our case, we have chosen somewhat hybrid approach (Fig. 1). We cannot use a monolithic kernel, since we would like to implement a possibility of on-the-fly functionality extension (adding new test generation methods on-the-fly). However, the microkernel approach is not an option either. First, rapid analysis showed, that apart of test generation methods implementations, the rest of the functionality is not that large, to be splitted into different micro-services. Second, we would not like to provide a separate database or an HTTP-server for each of micro-services.

2.2. SOAP or REST

SOAP (*Simple Object Access Protocol*) [4] and REST (*Representational State Transfer*) [5] are two possible ways, how to organize interaction between a server and a client of a web-service.

SOAP, being a protocol, works on top of HTTP and specifies how calls of remote functions should be done and how the result should be returned. It is based on XML and is able to process very complicated messages (for example, when a function returns an object, containing lists of other objects etc.). But of course, for such a flexibility there is a price: increased traffic and a need of an XML parser at both ends of the communication channel.

In the same time REST is not a protocol, it is basically a prescription, how to perform HTTP requests and how these requests should be understood at the server side. It does not require such amount of traffic or special parser as SOAP does, and it is much easier to implement. However, it is usually considered as less safe and it definitely is not that flexible as SOAP is (example with an object containing lists of objects is very tricky for REST).

At present, we have chosen REST as a way of server-client communication, but when the service grows, we suppose to introduce SOAP.

2.3. The Architecture

When all the choices are done, we are ready to present the architecture of our web-service (Fig. 1). The service consists of the following parts. *The engine*, which implements the main functionality of the service. *HTTP server*, which provides HTTP transport and the rest of HTTP functionality. A PHP program that provides *Web API* of the service. A small *Simple PHP client*, available via an Internet browser. *The database*, which keeps information about users (username, password), running jobs (process ID, start date-time, stop date-time) and existing implementations of test generation methods (ID, path-to-run). And of course a bunch of test generation methods implementations, available as *Tool scripts*.

The functionality of each element is clear from its description above, but the service engine. The engine can be implemented either as a stand-alone program or as a PHP program, combining Web API. The functions, which we consider the basic ones for our service, are as follows.

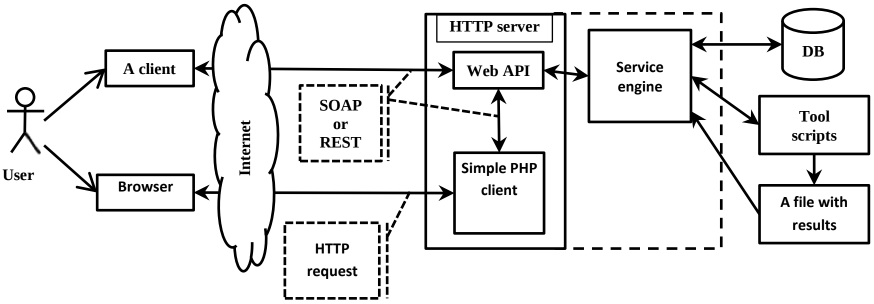

Fig. 1. The web-service architecture

1. *The authentication*. A user is asked to provide a username and a password. This function is always available; the rest functions are available only for successfully authenticated users.
2. *A job request*. A user send a request to run specific tool to build a test.
3. *A request for a list of running jobs*. A test generation is a long-term job, it takes a lot of time. For that reason, a user should have a mechanism to see, which of his/her job requests are finished, and which are not.
4. *A request for a list of available methods*. Since it is supposed, that the service will be expanded on-the-fly, the number of method may change and a user should be able to see the change.
5. *A request for a generated test*. When a job is done and the test is ready, a user should be able to get the result.

6. *A request for a new method deployment*. A user provides a program, which implements a method for a test generation he or she would like to share and the service deploys it at its side.
7. *Jobs sanitization*. A test generation is a long-term job; it can take days or even more. We suppose, that the service is not the production one rather than for a research. We limited the time of the job to complete to 24 hours. Also, a host, which runs the server, may lose the power supply. In this case, all running jobs are lost, but no records are done about this fact in the database, and the lost jobs will be marked as "in progress" forever. The jobs sanitization mechanism resolves both issues. It searches for the jobs, which are running for too long, and stops them, and looks for the lost jobs and makes certain record about this fact to the database.

Among all the functions, the deployment function seems to be the trickiest function of the service engine. The problem is that a researcher's implementation of a new method may have some dependences which may not be satisfied on the host machine, and for now we do not see, how to overcome this issue.

2.4. The Web-Service Which is not A Web-Server

One can notice that the service we describe here is not a classical web-service. It happens because a test generation procedure is a long-term process. When a user requests to build a test, the server-side may not be blocked by the request, it should stay responsive for this particular user and other users. Therefore, the server-side changes its state from "ready for requests" to "ready for requests and running one job in the background". Then "… two jobs in the background", then "… two jobs in the background, but another ones, and those two are ready". And so on. The classical approach supposes that the server-side is stateless [6].

However, apart from the described issue, the rest of our design follows the classical web-service design and therefore we tend to call it a web-service. In the future, we would like to rethink on this issue.

3. Light Version of the Web-Service

In this Section, we describe an implementation of a light version of the web-service. First, we describe the simplifications made. Then we describe the stuck-at-fault fault model and the fault domain enumeration method, which we have chosen for the implementation. And at last we give some implementation details.

3.1. Simplifications

We decided to implement only a part of the service engine functions, namely: "The authentication", "A job request", "A request for a list of running jobs", "A request for a generated test", and partly "Jobs sanitization" (only the time to complete restriction).

The web API is abandon. The engine and the simple PHP client should be implemented as one PHP program. However, the "Jobs sanitization" and "A job request" are shared between the engine and the tool script. And the tool script is a shell-script which implements the fault domain enumeration method. In the light version it is the only method implemented, that is why instead of several tool scripts we have only one.

According to the selected engine functions, the database needs only two tables (entities): the user information table and the job information table.

3.2. The Fault Model and the Method

3.2.1. Logical Circuits

There are two main classes of digital circuits: combinational circuits and sequential circuits. The first ones are usually referred as memoryless circuit without feedbacks and the second ones are the rest circuits [1, 7]. A digital circuit can be seen from different perspectives (behavioural, structural and physical) and at different level of abstraction (system, register, logical, schematic) [1, 7]. From the point of view of the service that we design, it does not matter, which kind of a digital circuit or from what perspective and at what level is given, everything depends on a test generation method implementation (a tool script) being called, whether it can or not to handle the call. For the light version of the service, we have chosen *combinational circuits*, considered from the *behavioural* and *structural* perspectives at the *logical* level, tested against the *stuck-at-faults* fault model using the *fault domain enumeration method*.

3.2.2. Behaviour and Structure of a Digital Circuit

Behavioural model at the logical level of a combinational digital circuit with n input pins and m output pins is defined as an *ordered system of m Boolean functions,* mapping Boolean vectors of the length n into the Boolean set [1, 2, 7]: $f_i(x_1,.., x_n) \in \{\mathbb{B}^n \to \mathbb{B}\}$, where $\mathbb{B} = \{0, 1\}$ is the Boolean set, $i \in \{1, \ldots, m\}$. Later on in this paper, we shall omit the word "ordered", but will always mean that, if the opposite is not specially denoted.

Two ordered system of Boolean functions $f_i(x_1,.., x_n)$ and $f'_i(x_1,.., x_n)$, $i \in \{1, \ldots, m\}$, are equivalent, if for each $i \in \{1, \ldots, m\}$ it holds, that $f_i = f'_i$ (corresponding Boolean functions are equivalent).

A structural model at the logical level of a combinational digital circuit is called a *logical combinational circuit*. *A logical combinational circuit (or just a circuit)* for a combinational digital circuit with n input pins and m output pins is a directed acyclic graph [2] $G = \langle V, E\rangle$, where V – is non-empty, finite set of *nodes*, $E \subseteq V \times V$ – is a set of ordered pairs $\langle v_1, v_2\rangle$ called *edges*, and every node $v \in V$ is either *an input pole* or *a gate* or *an output pole*. Poles correspond to the pins of the digital combinational circuit. There should be exactly n input poles and m output poles. An input pole has no ingoing edges, and at least one outgoing edge. An output pole has no outgoing edges and exactly one ingoing edge. A gate has at least one ingoing and at least one outgoing edge and is associated with some Boolean function g: $\mathbb{B}^k \to \mathbb{B}$, where arity k equals to the number of ingoing edges for this gate.

A logical circuit with n input poles and m output poles maps Boolean vectors of length n into m Boolean values. This mapping is computed as follows. Let there is a vector $\langle \alpha_1 \ldots \alpha_n \rangle \in \mathbb{B}^n$ (*an input vector*). Before start, if there are any mappings of the circuit's edges and vertices, drop these mappings. When a node is mapped to some Boolean value, its outgoing edges are also mapped to this very value. For each $j = 1,\ldots,n$ the input pole j is mapped to the value α_j. When every ingoing edge of a gate is mapped to some value γ_l, the node becomes mapped to the value $g(\gamma_1, \ldots, \gamma_k)$, where g is the function associated with the gate. Due to our definition of the logical circuit, each output pole i eventually is mapped to some Boolean value β_i. So, at some point we are able to collect *an output vector*

$$\langle \beta_1 \ldots \beta_m \rangle = \langle f_1(\alpha_1, \ldots, \alpha_n), \ldots, f_m(\alpha_1, \ldots, \alpha_n) \rangle \in \mathbb{B}^m.$$

The fact, that the circuit G maps vector $\langle \alpha_1 \ldots \alpha_n \rangle$ to the vector $\langle \beta_1 \ldots \beta_m \rangle$ we shall denote as $G(\langle \alpha_1 \ldots \alpha_n \rangle) = \langle \beta_1 \ldots \beta_m \rangle$.

Therefore, every logical circuit is associated with a system of Boolean functions $f_i(x_1,.., x_n)$, $i \in \{1, \ldots, m\}$. This system is a behavioural model of the underlying digital circuit. Apparently, that every logical circuit has the only corresponding system of Boolean functions, and we say that the logical circuit *implements* the system of the Boolean functions. However, every system of Boolean functions may have several corresponding logical circuits (structural implementations).

3.2.3. Tests and Fault Models

Two logical circuits G and F are *equivalent* if they implement the same system of Boolean functions. This fact we shall note as $G \cong F$. The non-equivalence of two circuits G and F we shall denote as $G \ncong F$.

Let a logical circuit be given. They say, that there is *stuck-at-zero (stuck-at-one) fault* [1] at the edge $e = \langle v_1, v_2 \rangle$ if this edge is constantly mapped to *zero* (*one*), regardless from the any other mappings. Stuck-at-zero (stuck-at-one) fault means, that node v_2 is virtually associated with Boolean function $g(x_1, \ldots, x_{l-1}, 0, x_{l+1},\ldots, x_n)$ (with function $g(x_1, \ldots, x_{l-1}, 1, x_{l+1},\ldots, x_n)$), where l is the index of the edge e and $g(x_1, \ldots, x_n)$ is the Boolean function actually associated with the node v_2. One can consider the case, when several edges are faulty in the circuit, but for the purpose of this paper, we consider only single stuck-at-faults.

A fault model is a triple $\langle S, \cong, \Omega \rangle$, where S – is a logical circuit (usually called *a specification*), describing a correct behaviour of a system; Ω – is a set (usually called *a fault domain*), containing logical circuits, which are considered as possible (correct and incorrect) implementations of the system of Boolean functions defined by S; and $\cong$ – is the equivalence relation. Every circuit from Ω has the same numbers of input and output poles as S does.

A test case t for the fault model $\langle S, \cong, \Omega\rangle$ is a Boolean vector $\langle \alpha_1\dots\alpha_n\rangle \in \mathbb{B}^n$, where n – is the number of the input poles of the circuit S. *A test T* for the fault model $\langle S, \cong, \Omega\rangle$ is a finite set of test cases for the fault model $\langle S, \cong, \Omega\rangle$. A test T for the fault model $\langle S, \cong, \Omega\rangle$ is called *sound* if for each circuit $G \in \Omega$ such that $G \cong S$ and for each $t \in T$ it holds, that $G(t) = S(t)$. A test T for the fault model $\langle S, \cong, \Omega\rangle$ is called *complete* if for each circuit $G \in \Omega$ such that $G \ncong S$ there exists $t \in T$ such that $G(t) \neq S(t)$. A test T for the fault model $\langle S, \cong, \Omega\rangle$ is called *exhaustive* if it is sound and complete.

In this work, we consider a fault domain, which contains S and every possible circuit got from S by introduction every possible single stuck-at-zero or stuck-at-one fault. The fault model with this fault domain is called *the stuck-at-faults fault model*. Since S is finite graph with finite number of edges, the fault domain in this case is finite. Namely, the number of circuits in fault domain equals to $2\cdot|E|$, where E is the set of edges.

3.2.4. Miter

Given two (ordered) systems of Boolean functions $f_i(x_1,.., x_n)$ and $f'_i(x_1,.., x_n)$, $i \in \{1, \dots, m\}$. A *miter* for them is the following function

$$f_M = (f_1(\boldsymbol{x}) \oplus f'_1(\boldsymbol{x})) \vee \dots \vee (f_m(\boldsymbol{x}) \oplus f'_m(\boldsymbol{x})),$$

where $\boldsymbol{x} = \langle x_1 \dots, x_n\rangle$ is the vector of arguments. Note, that the order of the functions in the systems is important.

Proposition. *Two systems of Boolean functions* $f_i(x_1,.., x_n)$ and $f'_i(x_1,.., x_n)$, $i \in \{1, \dots, m\}$ *are not equivalent if and only if there exists such a vector* $\boldsymbol{\alpha} \in \mathbb{B}^n$ *that* $f_M(\boldsymbol{\alpha}) = 1$, *where* f_M *– is the miter for the give systems*.

If there exists such a vector $\boldsymbol{\alpha} \in \mathbb{B}^n$ that $f_M(\boldsymbol{\alpha}) = 1$, then the miter is called *satisfiable* and $\boldsymbol{\alpha}$ is called *satisfying vector*. In contrary, if for any $\boldsymbol{\alpha} \in \mathbb{B}^n$ $f_M(\boldsymbol{\alpha}) = 0$, then the miter is called *unsatisfiable*.

Of course, a miter can be built for logical combinational circuits as well. A miter for logical circuits G and G' – is a circuit M, which we get by matching input poles of G to corresponding input poles of G' (v_1 to v'_1, v_2 to v'_2 etc.), so we have n input poles. Then we combine every output pole u_i of G with corresponding output pole u'_i of G' by XOR gate, and then combine all outgoing edges of those XOR gates by disjunction gate. And at last outgoing edge of the disjunction gate comes to only output pole u of M. Now, if we compute mapping, done by the structural miter we shall get the same function f_M.

3.2.5. The Method of Fault Domain Enumeration

The idea of the method is quite simple. Let the stuck-at-faults fault model $\langle S, \cong, \Omega\rangle$ be given. Since for the case of stuck-at-faults fault model the set Ω is finite, and its cardinality is polynomial on the edges number of S, we can enumerate it. At the beginning, a test under construction T is empty. Get the next circuit G from Ω.

Check, whether T contains such a vector $\boldsymbol{t}$ that $G(\boldsymbol{t}) \neq S(\boldsymbol{t})$. If it does, take the next circuit from Ω and repeat. If it does not, build a miter for G and S and check, whether it is unsatisfiable. If it is not (i.e., satisfiable), then take a counter example for unsatisfiability as a test case and put it in T.

Proposition. *The above procedure provides an exhaustive test.*

There are some notes to the procedure above. First, it does not guarantee that the resulting test is minimal on number of test cases. But for the goals of this work we do not aim at getting optimal tests. Second, the problem of unsatisfiability check is not an easy task by itself [8]. It is known to be NP-complete and may bring a lot of headache in general case. Fortunately, this problem is quite popular among researches and many powerful tools were developed to solve it, which perform well in most real life cases. For example, see works on MiniSAT [9].

3.3. Implementation Details

3.3.1. Operating System, HTTP Server, DBMS, and the Engine

The available to us host machine, where we can deploy the service, runs under FreeBSD operating system. It has MySQL database management system and Apache HTTP server with the support of PHP installed. Therefore, we had no need of choosing the right environment for our web-service. The triple Apache+MySQL+PHP proved to be the reliable basis for any project of any level of complexity.

The engine of the light version of the web-service was implemented in PHP. As it was noted earlier, we abandon the web API, and the small PHP client was implemented together with the engine as one program.

3.3.2. The Method Implementation

To implement the method of fault domain enumeration for the stuck-at-faults fault model as a tool, we use the tool called ABC [10]. This tool has built-in instruments for a miter construction and checking the miter for being unsatisfiable. If the miter is not unsatisfiable, ABC provides a counter example.

The only thing we implemented in addition to ABC is an instrument for faulty circuit's enumeration. This instrument is implemented as a stand-alone program, which takes a logical circuit (specification) and a folder to store the result as an input and saves files with faulty logical circuits in the designated folder.

Since the chosen fault model describes faults in the structural terms (in terms of logical circuits), the method implementation requires digital circuits under test be given as logical circuits. ABC understands logical circuits in ISCAS'89 format (also known as `.bench` format). This format simply describes directed graph. For example, a code like this

```
INPUT(x1)
INPUT(x2)
OUTPUT(y)
y = AND(x1, x2)
```

states, that there are two input poles, named `x1` and `x2`, one output pole, named `y`, and one gate associated with the conjunction (AND), but without a name. Let call this gate *g*. Then the edges are ⟨`x1`, *g*⟩, ⟨`x2`, *g*⟩ and ⟨*g*, `y`⟩.

To finish the method implementation, the faults enumerator and ABC should be called in the right order with the right parameters. This is done by a shell-script, which in terms of the architecture of the web-service we call *the tool script*.

3.3.3. Jobs sanitization

The most of the functions implemented in the engine of the light version of the web-service are quite straightforward and we shall not describe them. However, the implemented part of the “Jobs sanitization” (namely, the time to complete restriction) and “A job request” functions are worthy of mention and description.

When the engine gets the request from a user to start a new job of test derivation, it runs the script and delegates all interactions with the database to the script. The script makes all necessary records in the database: the date-time of the script start, the ID of the user, who requested this job, process ID of the job. When the script finishes the task, it again makes necessary records about that: the task was normally finished, when the task was finished.

The implementation of the time to complete restriction is done with the use of `timeout` utility built into FreeBSD [11]. This utility starts the specified program, waits for specified amount of time and if the program is still running, it sends to the program the termination signal. The script catches this signal and before termination, it makes a record into the database, that certain task was artificially stopped. That is the engine does not run the script directly, it runs the `timeout` utility, specifying that the script should be run and should finish within 24 hours. The rest is done by the `timeout` utility.

This approach to implementation of the “A job request” and “Jobs sanitization” functions helped to guarantee the responsiveness of the web-service whilst the requested jobs are run in the background and successfully sanitized.

4. Conclusion

In this paper, we presented architecture of the web-service we would like to develop. The main goal of the web-service is to give for researchers a platform, where they can do preliminary rapid experiments with different test generation methods for digital circuits. And, as the second feature of the web-service, researchers can share an implementation of a new method they developed.

The architecture has properties of both a microkernel service and a monolithic kernel services. Also, the analysis of the architecture shows, that in its current state it is not the classical architecture of a web-service, because web-services are supposed to be stateless, and our service by design has states. This issue needs some further research.

Another issue that needs to be resolved in the future is the deployment of new methods implementations on the service. For now, it is not clear, how to guarantee possible dependences.

As the second stage of the development of the web-service, we implemented the light version of the service. We took only minimal necessary functionality, and the only test generation method – the fault domain enumeration method for the stuck-at-faults fault model for combinational logical circuits. This implementation proved that the designed architecture is viable. In addition, the implementation showed, that some functions of the service engine (like the “Jobs sanitization”), probably, are better be moved away from the engine.

Acknowledgements

This work is partially supported by the grant of the Russian Science Foundation № 16-49-03012. The authors would like to thank Andrey Laputenko for presenting the result of the paper at the ESRCT 2017 workshop. In addition, the authors would like to thank the reviewer of the paper, whose constructive comments helped to make this paper better.

References

[1]. Skobcov Yu.A., Skobcov V.Yu. Logical modeling and testing of digital devices. Doneck: IAMM NAS of Ukraine, DonNTU, 2005. 436 p. (in Russian)
[2]. Zakrevskij A.D., Pottosin Yu. V., Cheremisinova L.D. Fundamentals of logic design. Minsk: UIIP NAN of Belorus, 2006. 254 p. (in Russian)
[3]. Chernov A.V., Sergeeva E.A. Autocorrelational testing of digital combinational circuits. Sovremennye problemy nauki i obrazovaniya [Modern problems of science and education], 2013, № 6 (in Russian)
[4]. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). https://www.w3.org/TR/soap12/, 05.02.2018.
[5]. Wilde E., Pautasso C. REST: From Research to Practice. Springer Science & Business Media, 2011. 528 p.
[6]. Fielding R.T. Architectural Styles and the Design of Network-based Software Architectures. Chapter 5. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 05.02.2018.
[7]. Harris D., Harris S. Digital Design and Computer Architecture. Morgan Kaufmann, 2012, 712 p.
[8]. Silva L.G. e, Silveira L.M. and Marques-Silva J.P. Algorithms for Solving Boolean Satisfiability in Combinational Circuits. In Proceedings of DATE’99, 1999, pp. 526-530.
[9]. Niklas Eén, Niklas Sörensson. The MiniSAT. http://minisat.se/Main.html, 16.10.2017.
[10]. ABC: A System for Sequential Synthesis and Verification. https://people.eecs.berkeley.edu/~alanmi/abc/, 16.10.2017.
[11]. FreeBSD Manual Pages. timeout. https://www.freebsd.org/cgi/man.cgi?query=timeout&sektion=1, 05.02.2018.

Тесты на константные неисправности как веб-сервис

Н.А. Шаляпина <nat.shalyapina@gmail.com>
А.А. Зайцев <z.sania@mail.ru>
С.В. Батрацкий <pride080993@gmail.com>
М.Л. Громов <maxim.leo.gromov@gmail.com>
Томский государственный университет,
634050, Россия, Томск, пр. Ленина, 36.

Аннотация. В этой статье рассказывается о разрабатываемом нами веб-сервисе. Разрабатывая этот сервис, мы преследуем две цели. Первая – предложить исследователям платформу, где они могли бы проводить предварительные эксперименты с различными методами генерации тестов для цифровых схем, для проверки различных идей. Вторая – возможность «на лету» поделиться реализациями новых методов. Процедура разработки веб-сервиса была разделена на три этапа: дизайн архитектуры, реализация облегчённой версии и фактическая реализация. В этой статье рассказывается о первых двух этапах. Есть два типа архитектур веб-сервисов – с монолитным ядром и микроядром – и наша архитектура обладает свойствами обоих типов. Мы стремились к тому, чтобы получить монолитное ядро, поскольку желаемая функциональность не так уж трудно реализовать. Однако расширяемость реализациями новых методов подразумевает, что часть функций (а именно, реализации методов) должны быть разработаны как отдельные под-сервисы. Реализация легкой версии была выполнена для единственного метода: метода перебора области неисправности для модели константных неисправностей. Она показал, что разработанная архитектура жизнеспособна. Однако были обнаружены некоторые проблемы с ней. Механизм развертывания добавляемого «на лету» метода неясен, так как неясно, как удовлетворить возможные зависимости реализации. Кроме того, архитектура не соответствует классическому дизайну веб-сервиса: у сервиса есть состояния, которых не должно быть, если сервис классифицирован как классический. Решение этих вопросов остается на будущее.

Ключевые слова. Константные неисправности, комбинационные схемы, Web сервис, проверяющие тесты.

DOI: 10.15514/ISPRAS-2018-30(1)-3

Для цитирования: Шаляпина Н.А., Зайцев А.А., Батрацкий С.В., Громов М.Л. Тесты на константные неисправности как веб-сервис. Труды ИСП РАН, том 30, вып. 1, 2018 г., стр. 41-54. DOI: 10.15514/ISPRAS-2018-30(1)-3

Список литературы

[12]. Скобцов Ю.А., Скобцов В.Ю. Логическое моделирование и тестирование цифровых устройств. Донецк, ИПММ НАН Украины, ДонНТУ, 2005,436 с.
[13]. Закревский А.Д., Поттосин Ю.В., Черемисинова Л.Д. Fundamentals of logic design. Минск, Национальная академия наук Беларуси, 2006, 254 с.

[14]. Чернов А.В., Сергеева Е.А. Автокорреляционное тестирование цифровых комбинационных схем. Современные проблемы науки и образования, 2013, № 6
[15]. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). https://www.w3.org/TR/soap12/, 05.02.2018.
[16]. Wilde E., Pautasso C. REST: From Research to Practice. Springer Science & Business Media, 2011. 528 p.
[17]. Fielding R.T. Architectural Styles and the Design of Network-based Software Architectures. Chapter 5. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 05.02.2018.
[18]. Harris D., Harris S. Digital Design and Computer Architecture. Morgan Kaufmann, 2012, 712 p.
[19]. Silva L.G. e, Silveira L.M. and Marques-Silva J.P. Algorithms for Solving Boolean Satisfiability in Combinational Circuits. In Proceedings of DATE’99, 1999, pp. 526-530.
[20]. Niklas Eén, Niklas Sörensson. The MiniSAT. http://minisat.se/Main.html, 16.10.2017.
[21]. ABC: A System for Sequential Synthesis and Verification. https://people.eecs.berkeley.edu/~alanmi/abc/, 16.10.2017.
[22]. FreeBSD Manual Pages. timeout. https://www.freebsd.org/cgi/man.cgi?query=timeout&sektion=1, 05.02.2018.