# A Systematic Mapping about Testing of Functional Programs

**Alexandre Ponce de Oliveira[1 2], Paulo Sérgio Lopes de Souza[1], Simone R. Senger de Souza[1], Júlio Cezar Estrella[1], Sarita Mazzini Bruschi[1]**

1: Universidade de São Paulo, ICMC, São Carlos, SP, Brazil
2: Faculdade de Tecnologia de Lins, Lins, SP, Brazil

**Abstract -** *Functional languages, like Erlang, Haskell and Scala allow the development of real-time and fault-tolerant parallel programs. In general, these programs are used in critical systems such as telephone switching networks and must provide high quality, reliability and efficiency. In this context, validation, verification and testing activities are necessary and contribute to improving the quality of functional programs. This paper presents a systematic mapping concerning the testing of functional programs, considering also their parallel/concurrent aspects. The paper describes the three stages used during the systematic mapping: planning, execution and presentation of results. The mapping was able to identify only twenty-two relevant studies. In these studies, fourteen considered test models, three used data flow testing, twelve used/proposed testing tools and five considered concurrent/parallel aspects of such programs. The results indicate that there are few researchers working on testing of functional programs and that few studies are concentrated almost exclusively in the Erlang language. The main testing technique found in the papers is the structural one; however, it does not properly consider the software testing methodology already established for the imperative programming. Indeed, the results show gaps in the area of testing of functional programs, even for Erlang, the most considered language by the studies. These gaps are presented and discussed at the end of this paper.*

**Keywords:** Testing, functional programs, Erlang, testing criteria, test models.

## 1 Introduction

Nowadays functional programs are an aim of research in universities with distinct examples of research and applications [12]. Parallel and soft-real time features are key aspects related to functional applications, which stimulate the interest for new research.

Functional languages can be used also to build programs utilizing expressions as mathematical functions, avoiding both mutable data and changes in the state of the program that do not depend on the function inputs. The program behavior can be easier to predict when using this paradigm, which motivates research on functional languages. Some examples of functional languages are: Lisp [24], Haskell [37], Scala [37] and Erlang [3] [4].

The functional applications are often critical and failures affect their quality, reliability and efficiency. In this sense, the testing of functional applications is essential to prevent potential failures and to ensure that all features are according to what is expected [6].

Software testing activity aims to find unrevealed defects that are responsible for errors during the execution of programs [25]. A number of studies have been conducted in sequential and concurrent software testing, investigating models, criteria and tools for testing.

Considering the context of concurrent programs, for example, Taylor et al. [38] proposes to apply coverage criteria for concurrent programs. Yang et al. [52] adapts All-Du-path testing criterion for concurrent programs.

Souza et al. proposes structural coverage criteria for C/MPI [33] [35], C/Pthreads [32], BPEL [11] and Java [34]. However, this scenario is not true for the testing of functional programs, since it is not trivial to find studies already published in the context of functional programs (sequential or concurrent).

Functional programs present concurrent aspects and therefore these aspects should be properly explored during the testing activity. In order to contribute to this scenario, it is important to consider state-of-the-art research on functional program testing. We could not find a literature review available in this context, which motivated this work. Considering this scenario, a systematic mapping process was used to collect, guide new research and analyze the papers already published for the testing of functional programs. A systematic mapping identifies, in the literature, what type of studies can be considered in a systematic review, pointing out mainly where those studies have been published and their main results.

A systematic mapping allows a wider view of primary studies, which can reveal the evidences of research [27]. A systematic mapping process is capable to contribute with new research insights in a particular area, providing an initial overview. The systematic review, on the other hand, tries to identify, evaluate and interpret all the available works, relevant for a specific research question [7].

This paper identifies, through a systematic mapping, studies related to the testing of functional programs, classifying and analyzing relevant papers in this context. The eligible papers were classified under three main features: *a)* work that proposes novel models of testing to functional paradigms; *b)* work that presents testing criteria related to this subject; and *c)* work that presents a software tool to support the testing activity. This classification facilitates the analysis of the selected papers.

The main results indicate that there is little research on testing of functional programs. These studies are focused, almost exclusively, on the Erlang language, using the structural testing technique. However, they do not properly consider the software testing methodologies already established for the imperative programming. It is important to consider this previous research, because the knowledge produced for imperative programming can guide the definition of new approaches for new contexts. Indeed, the results show gaps in the area of testing of functional programs, even for Erlang, the most considered language by the studies.

This paper is structured as follows: Section 2 presents some of the main features of functional languages that make the testing of functional programs different from the testing of imperative programs; Section 3 includes details of the systematic mapping planning; Section 4 presents the execution of the systematic mapping planned; The results are discussed in Section 5 and in Section 6 the main conclusions are drawn.

## 2 Functional Programs

Functional languages are based on mathematical functions. An important feature of mathematical functions is the use of recursion and conditional expressions to control the order in which pattern matching is evaluated. The variables in functional language are immutable, so once a value is assigned, it cannot be changed; this feature does not generate side effects to the functions [29].

Functional languages have no side effects (or state change), because they define the same value given a same parameter (referential transparency). Functional languages also use higher-order functions; which are functions that receive functions as parameters and can generate a function as a result [43].

A function definition, in functional languages, uses pattern matching to select a guard among different cases and to extract components from complex data structures. Erlang [3] works in that way, combining high level data with sequences of bits, to enable functions of protocol handling.

Concurrency is a fundamental and native concept of some functional languages, such as Erlang. Those languages do not provide threads to share memory, thus each process runs in its own memory space and has its own heap and stack. These processes in Erlang employ the Communicating Sequential Processes (CSP) model [17]. Hoare [17] described how sets of concurrent processes could be used to model applications. Erlang explores this idea in a functional framework and uses asynchronous message passing instead of the synchronous message passing of CSP. Each process has a mailbox to store incoming messages, which are selectively obtained [8].

Some functional applications may run transparently in a distributed environment. In Erlang, a VM (Virtual Machine) instance is called *node*. One or more computers can run multiple nodes independently from the hardware architecture and even operating system. Processes can be created in remote nodes, because processes can be registered in Erlang VM.

Fault Tolerance is a necessary resource for concurrency applications, in this context. Erlang has libraries that support supervisors, worker processes, exception detection and recovery mechanisms. Thus, processes create links to each other to receive notifications as messages. This is used, for example, when a process finishes [23].

## 3 Systematic Mapping Planning

This systematic mapping was performed according to the process defined by Kitchenham and Charters [18] and Petersen et al. [27]. This process consists of three stages: a) planning – definition of a protocol specifying the plan that the systematic mapping will follow; b) execution – the execution of the protocol planned; and c) presentation of the results [7].

Primarily, our main objective with the systematic mapping was to identify studies that explore the testing of concurrent aspects of functional programs. However, we found few studies in this more restrict context and therefore we decided to make this systematic mapping broader, in order to find a wider range of publications about functional software testing as a whole. Considering this scenario, three research questions were defined and used to conduct the systematic mapping carried out in this paper:

*Question 1 (Q1): What aspects related to the testing of functional languages have been identified?* Our interest here is to identify the main features in the functional paradigm that make the test activity more complex in this context.

*Question 2 (Q2): How is the testing activity of the functional programs conducted?* The aim is to find studies that apply testing methodologies and to establish which/how testing criteria are used.

*Question 3 (Q3): Are there testing tools for functional programs?* Identifying testing tools that support the testing activity is important due to the complexity of the testing activity and the difficulty to apply it manually.

### 3.1 Search String and Source Selection

The search string was defined as follows: first, the main search keywords were established based on our research questions. We considered terms such as functional language, software testing and testing tools. The languages Erlang, Haskell and Lisp have been inserted in our search string because they are the most used functional languages for both academic and industrial purposes. However, it must be observed that the string did not restrict the search just for these three languages. Next, a set of relevant synonyms for the search keywords was identified, based on the terminology used in a set of relevant contributions in the area of software testing and functional language. Thus, the main keywords were combined with the chosen synonymous using the Boolean operators AND and OR. The search string used in the systematic mapping is:

[("functional language" or "erlang" or "lisp" or "haskell") AND ("software testing" or "structural testing" or "mutation testing" or "functional testing" or "blackbox" or "whitebox" or "tools" or "test" or "criteria" or "coverage")]

In Table 1 the digital libraries selected to conduct the systematic mapping are presented. These libraries were chosen because they present the most relevant sources in software engineering.

**Table 1. Selected Digital Libraries.**

| Digital Library | Link |
|---|---|
| **ACM** | http://dl.acm.org/ |
| **IEEE Xplore** | http://ieeexplore.ieee.org/Xplore/home.jsp |
| **SCOPUS** | http://www.scopus.com/ |

### 3.2 Studies Selection

The following inclusion criteria (IC) were defined in order to obtain primary studies that could provide answers to the research questions. It is important to observe that just one valid inclusion criterion is enough to include a primary study in the next step (eliminate primary studies).

*IC1:* Primary studies presenting testing models for applications written in functional languages;
*IC2:* Primary studies proposing tools and research for the context of functional language;
*IC3:* Primary studies applying case studies in the context of functional program testing.

The following exclusion criteria (EC) were defined to eliminate primary studies when they are not related to the research questions:

*EC1:* Primary studies presenting testing approaches not related to functional languages;
*EC2:* Primary studies presenting approaches related to hardware testing;
*EC3:* Primary studies presenting tutorials about software testing or functional languages.

### 3.3 Data extraction

A form was filled with the extracted data. This form was used to record information obtained from primary studies, as described in Kitchenham and Charters [18]. The data extraction provides information such as: extraction source, title, year and authors. The procedure to extract the data was carried out after the studies. A summary was written for each examined study, in order to facilitate the documentation of the responses for the research questions.

## 4 Systematic Mapping Execution

The systematic mapping was carried out with the support of the tool StArt (State of the Art through Systematic Review) [30]. Despite its name, related to systematic review, this tool offers facilities to support all the activities of the systematic mappings, including planning, execution and summarization.

The studies were selected in September, 2014 and there were three different stages, as described in the sequence. Initially, 556 studies were retrieved.

In Stage 1, duplicate studies were identified and eliminated (done automatically by the StArt tool). Furthermore, we eliminated non relevant data, such as conference proceedings, abstracts and unavailable papers. After this stage, only 44 studies remained.

In Stage 2, we applied the inclusion and exclusion criteria based on title, abstract and keywords. Moreover, we read the conclusion and the introduction sections of each study in order to apply the inclusion and exclusion criteria. After this stage, only 22 studies remained.

At the final phase (Stage 3), the studies were analyzed completely. In this phase we selected 17 studies. According to our preliminary studies, five other studies were included: [47], [48], [49], [50] and [51]. Such studies were not indexed by digital libraries but were published in local workshops. Thus, 22 studies were selected.

Table 2 shows the number of studies selected at each stage, considering the total studies retrieved from the digital library. All the results of the search procedure were documented and are available[1]. If necessary, the search procedure can be repeated considering, for example, a different period of time.

## 5 Systematic Mapping Results

This section presents the mapping results, grouping the selected studies according to the research question. The aims of the studies are described as follows.

*Q1. What aspects related to the testing of functional languages have been identified?*

Widera [51] explains that generating a control flow graph (GFC) for functional programs is more complex than for traditional programs due to the existence of higher-order functions. The difference in the control structures of functional languages in relation to imperative languages also makes the application of the coverage criteria more complex, in the functional context.

[1] http://labes.icmc.usp.br/~alexandre/mapping.pdf, 2014.

**Table 2. Number of Studies Selected During the Search Procedure**

| Digital Library | Return | Stage 1 | | Stage 2 | | Stage 3 | |
|---|---|---|---|---|---|---|---|
| | | Included | Excluded | Included | Excluded | Included | Excluded |
| **ACM** | 72 | 19 | 53 | 8 | 11 | 6 | 2 |
| **IEEE** | 171 | 4 | 167 | 1 | 3 | 1 | - |
| **SCOPUS** | 315 | 23 | 292 | 15 | 8 | 10 | 5 |
| **Total** | 556 | 44 | 512 | 22 | 22 | 17 | 7 |

An example of this occurs when higher-order functions can receive and send functions as parameters. This dynamic creation of functions makes the control flow unpredictable and must be considered during the testing activity.

In this same context, Tóth and Bozo [41] cite that the aim of a Data Flow Graph (DFG) is to determine how far a variable definition can reach. This is because variables are immutable in functional languages. In the context of data flow, it is important to analyze a value from its first definition to its last use [50].

Considering the objective of this research question, we identified two main aspects of functions programs that impact the testing activity: higher-order functions and immutable feature of variables. Higher-order functions influence the control flow, which requires a proper analysis of the data flow. The immutable feature of variables brings the necessity of a variable to be copied to another one after its use, so it is important to identify this sequence of copies from its first definition up to its last use. All the studies were related to Erlang language (although, the authors argue that the studies could be extended to consider other functional languages, such as Haskell).

*Q2. How is the testing activity of the functional programs conducted?*

Tóth and Bozo [41] presented the Semantic Program Graph (SPG), a model to represent the semantic and syntactic information from Erlang programs. SPG is the basis to construct another three graphs: Data Flow Graph (DFG), Control Flow Graph (CFG) and Dependency Graph (DG).

The DG can be used to extract parts of the source code and then identify components that can be parallelized efficiently with inexpensive synchronization. Graphs are integrated in the RefactorErl software, which analyzes the source code and extracts parts of the Erlang code.

Toth et al. [39] investigated the use of SPG during the regression testing aiming to reduce the number of test cases that must be considered to rerun. A behavioral dependency graph is specified and used to represent test cases affected by changes in the program´s behavior, due to modifications. In a similar way, Tóth and Bozo [40] investigate the use of a dependency control graph to support the selection of effective test cases during the regression testing for Erlang programs.

Silva et al. [31] specified a graph called the Erlang Dependency Graph (EDG), which shows the dependencies of data and control in function calls. The authors propose a testing tool, named Slicerl to extract relevant parts of the Erlang program based on the proposed model. Guo et al. [14] defined a model, named Complete Functional Binary Tree (CFBT), which transforms each Erlang function into a tree structure. Each node of the tree corresponds to a predicate of the original function and the objective is to represent all predicates in order to apply coverage criteria based on the CFBT.

Five selected studies, described below in this (Q2) research question, did not consider concurrent aspects of the functional programs although all of them considered Erlang. These studies explore the definition of test models and they do not specify testing criteria.

Three studies discussed in the previous research question (Q1) also contribute to the definition of models and criteria for testing of functional programs. Widera [44] describes a test model to include a subset of Erlang functions and proposes a GFC for this model. This model covers only sequential Erlang programs. In Widera [45] the model is extended to include higher-order functions. Widera [46] complements the model to include concurrency aspects of Erlang programs.

In the context of testing criteria, four studies were retrieved. Widera [47] proposes a set of coverage criteria based on data flow testing adapted to functional programs. These criteria are based on associations of definition and use of variables (du-pair) that is a triple (v, d, u). In this triple, v is a variable, d is a definition of v, u is a use of v and there is a path w from d to u such that v is not redefined on w. Widera [48] introduces the du-chain concept, which is a sequence p1;….; pk of du-pairs, such that the definition of p1 and the use of pk denote the same value. Based on this concept and considering a flow graph G, a set of five testing criteria was defined: a-aware (aliasing aware), s-aware (structure aware), r-aware (result aware), f-aware (freeze aware) and m-aware (message aware).

Tasharofi et al. [36] presents a scalable and automatic approach to test non-deterministic behavior of actor-based Scala programs [1]. This approach uses three schedule coverage criteria for actor programs, an algorithm to generate feasible schedules to increase the coverage and a technique for deterministic execution. Le et al. [19] presents new mutation operators for functional constructs and also describes MuCheck, a mutation testing tool for Haskell programs.

To summarize, we observed contributions that explore mainly the structural testing for functional languages. These papers present propositions to represent and to extract relevant information for testing of functional programs.

*Q3. Are there testing tools for functional programs?* Widera [49] considers data flow tracing of Erlang codes and describes the properties and implementation of an interpreter prototype for GFC. The interpreter instruments the source code with the aim to evaluate parts of the GFC that are covered by the test cases. The study does not evaluate the coverage criteria; it only makes a comparison of the runtime of small code examples with and without the interpreter.

Nagy and Vig [26] present a survey about the main testing tools used by developers of Erlang systems. The survey is focused on model-based testing and Test-Driven Development (TDD). The tools mentioned by the developers were Dialyzer, EUnit, Wrangler and RefactorErl and QuickCheck, which was proposed by Claessen and Hughes [10]. The survey specifies that the tools used by the developers do not present information about the coverage of test cases and that it is also difficult to know what was really tested into the program. These aspects encourage the improvement of tools available for concurrent functional programs testing.

Christakis and Sagonas [9] present a technique for detecting errors related to message passing in Erlang, considering the dynamic process of creation and communication based on asynchronous message passing. Static analysis is used to build a communication graph from a Dialyzer tool. This graph is traversed during the testing activity, to obtain data about the message passing coverage.

Arts et al. [5] presented a testing tool called Quviq QuickCheck to analyze properties in Erlang programs. This tool uses a model to represent data type information from specification and during the testing; it can be evaluated whether the data types of the program meet its specification.

Wrangler and RefactorErl tools aim to support the refactoring of Erlang programs, the aim of which is to detect a similar code. Taking this into account, Li and Thompson [20] used the Quviq QuickCheck testing tool to automate the refactoring performed by the Wrangler tool. Li and Thompson [39] and [41] proposed a technique to detect syntactically identical codes, which was developed and integrated into the Wrangler testing tool. The authors used both syntactic analysis and code decomposition to remove duplicated code and thus reduce code maintenance.

Gotovos et al. [13] developed the Concuerror testing tool to assist the TDD process. This tool uses test sets to detect errors related to concurrency, such as deadlocks and race conditions in Erlang programs.

Therefore, six studies [9], [5], [20], [21], [22] and [13] are related to model testing, refactoring and TDD. Two studies [9] and [13] explore concurrency aspects.

## 5.1 Other Results

Figure 1 shows the number of selected studies by year. The result of the mapping showed studies only from the last 11 years, while 2011 had the highest score with four selected studies.

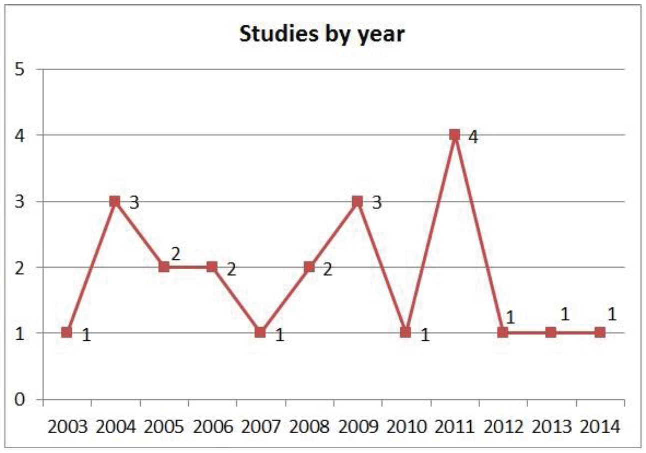

**Figure 1. Numbers of studies by year.**

Figure 2 groups studies by country, considering the authors´ affiliation. The results show that the University of Hagen in Germany has 8 studies, i.e., 36% of the selected studies. An important feature of these studies is that only four were conducted in partnership with universities in different countries. Two studies were conducted by universities in Greece and Sweden, one study was carried out by universities in Sweden and Spain and one study between universities in the USA and Switzerland.

Figure 3 shows the percentage of selected studies by research question. According to the result, 50% of the studies are related to Q2, which refers to a testing specification models and testing criteria. Q3 is related to testing tools, and 27% of the studies are in this context. Only 9% of the studies specify the challenges of testing activity for functional languages (Q1). Finally, 9% of the studies are in the context of Q1 and Q3 together and 50% of the studies between Q2 and Q3.

## 6 Concluding Remarks

A systematic mapping conducted to find studies on software testing for functional languages was presented in this paper. These studies provide an overview for the testing of functional languages, revealing the state of the art in terms of knowledge production in this area. These studies point out new research insights and can be used to guide further contributions in this context.

Some studies ([36], [46], [47], [48] and [50]) present the definition of data flow testing for functional programs in Erlang, exploring the definition-use of variables. In this group, five studies ([9], [13], [36], [46] and [48]) investigated the concurrency and parallel aspects existent in functional programs.

The selected studies proposing testing tools for functional programs, consider mainly structural aspects of such programs.

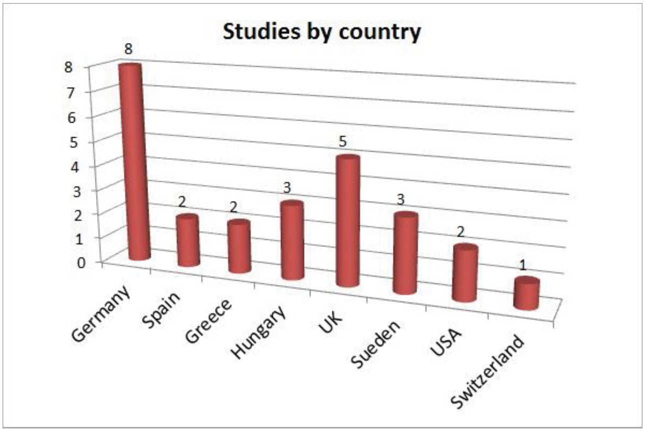

**Figure 2. Numbers of studies by country.**

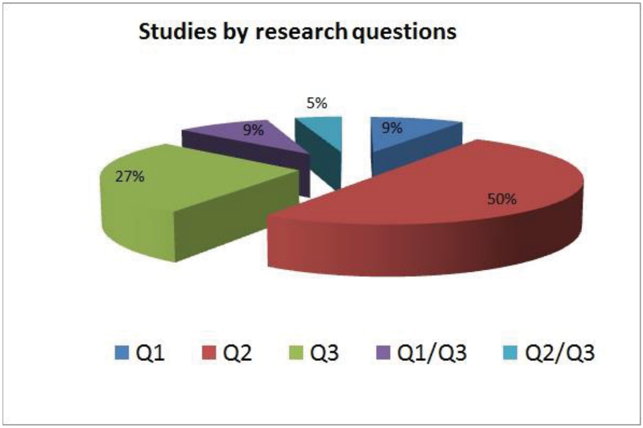

**Figure 3. Numbers of studies by research questions.**

However, in general, these tools do not apply properly the testing techniques; they do not explore the testing process, such as: generation of test cases and testing activity evaluation.

In summary, 63% of the studies present test models for Erlang programs; 45% of the studies applied a case study to evaluate a testing tool; 18% of all papers define testing criteria exploring sequential aspects of the programs and 9% investigate concurrent aspects of the programs to define testing criteria.

Furthermore, this mapping contributed to indicate lack of research exploring how to derive tests from functional programs and how to extract relevant information from these programs, in order to guide the testing activity. Also, there is a lack of experimental studies to analyze tools and testing criteria.

These results indicate a gap in research related to coverage testing applied to functional programs, mainly related to concurrent aspects of these programs. Considering this gap, we are investigating the definition of structural testing, exploring the same aspects in Souza et al. [33] in this context. We intend to define the coverage testing able to explore intrinsic aspects of this program, for instance: synchronization, communication, parallelism and concurrency considering message passing and other language features such as: higher order functions and functions call.

## References

[1] Agha, G. Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, USA, 1986.

[2] Almasi, G.; Gottlieb, A. Highly parallel computing. The Benjamin/Cummings series in computer science and engineering. Benjamin/Cummings Pub. Co., 1994.

[3] Armstrong, J., Virding, R., Wikström, C., and Williams, M. Concurrent Programming in Erlang. Prentice Hall Europe, Herfordshire, Great Britain, second edition, 1996.

[4] Armstrong, J. Concurrency Oriented Programming in Erlang. Invited talk, FFG. 2003.

[5] Arts, T.; Castro, L.M.; Hughes, J. Testing Erlang Data Types with Quviq QuickCheck. In: Proceedings of the ACM SIGPLAN Workshop on Erlang, ACM Press , 2008.

[6] Balakrishnan, A. and Anand, N. (2009). Development of an automated testing software for real time systems. In Industrial and Information Systems (ICIIS), 2009 International Conference on, pages 193 - 198.

[7] Biolchini, J.C.A.; Mian, P. G.; Natali, A. C. C.; Conte, T.U.; Travassos, G. H. Scientific research ontology to support systematic review in software engineering. Advanced Engineering Informatics, p.133-151, 2007.

[8] Cesarini, F. and Thompson, S. Erlang Programming - A Concurrent Approach to Software Development. O'Reilly Media, 2009. 496p.

[9] Christakis, M.; Sagonas, K. Detection of asynchronous message passing errors using static analysis. In: Proceedings of the 13th international conference on Practical aspects of declarative languages, PADL'11, p.5-18, Austin, USA, January 24-25, 2011.

[10] Claessen, K.; Hughes, J. QuickCheck: a lightweight tool for random testing of Haskell programs, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, p.268-279, September 2000.

[11] Endo, A. T.; Simão, A. S.; Souza, S. R. S.; Souza, P. S. L. Web services composition testing: A strategy based on structural testing of parallel programs. In: TaicPart: Testing Academic & Industrial Conference - Practice and Research Techniques, Windsor, 2008, pp. 3–12.

[12] Erlang FAQ. Who uses Erlang for product development? http://www.erlang.org/faq/introduction.html, 2014.

[13] Gotovos, A.; Christakis, M.; Sagonas, K. Test-driven development of concurrent programs using concuerror. In Proceedings of the 10th ACM SIGPLAN workshop on Erlang (Erlang '11). ACM, New York, USA, 2011.

[14] Guo, Q.; Derrick, J.; Walkinshaw, N. Applying Testability Transformations to Achieve Structural Coverage of Erlang Programs. In Proceedings of the 21st International Conference on Testing of Software and Communication Systems and 9th International Workshop FATES, Eindhoven, Netherlands, November 2-4, 2009.

[15] Grama, A; Gupta, A; Karypis, G; Kumar, V. Introduction to Parallel Computing. 2nd Ed. Addison Wesley, 2003.

[16] Hansen, M. R.; Rischel, H. Functional Programming Using F#. Cambridge University Press, 2013.

[17] Hoare, C.A.R. Communicating Sequential Processes. Prentice Hall, Upper Saddle River, NJ, 1985.

[18] Kitchenham, B.; Charters, S. Guidelines for performing systematic literature reviews in software engineering. Technical Report. EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[19] Le, D.; Alipour, M. A.; Gopinath,R.; Groce, A. MuCheck: an extensible tool for mutation testing of haskell programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). ACM, New York, NY, USA, p. 429-432. 2014.

[20] Li, H; Thompson, S. Testing Erlang Refactorings with QuickCheck. In the 19th International Symposium on Implementation and Application of Functional Languages, IFL 2007, LNCS, pages 182-196, Freiburg, Germany.

[21] Li, H; Thompson, S. Clone detection and removal for Erlang/OTP within a refactoring environment. In Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09). ACM, New York, USA. 2009.

[22] Li, H; Thompson, S. Incremental clone detection and elimination for erlang programs. In Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software (FASE'11/ETAPS'11). Springer-Verlag, Berlin, Heidelberg, 2011.

[23] Logan, M., Merritt, E., and Carlsson, R. Erlang and OTP in Action. Manning Publications. 2010.

[24] McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; Levin, Michael I. Lisp 1.5 Programmer´s Manual. Cambridge, Massachusetts: The MIT Press, 1962.

[25] Myers, G. J. The Art of Software Testing. 2 ed. John Wiley & Sons, 2004.

[26] Nagy, T., Víg, A.N. Erlang testing and tools survey. Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, September 27-27, 2008, Victoria, BC, Canada.

[27] Petersen, K.; Feldt, R.; Mujtaba, S. and Mattsson, M. Systematic mapping studies in software engineering. In Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE'08), 2008. British Computer Society, Swinton, UK, 68-77.

[28] Rauber, T.; Rünger, G. Parallel programming: for multicore and cluster systems. Springer, 2010.

[29] Sebesta, R. W. Concepts of Programming Languages. 10. ed. Pearson. 2012.

[30] StArt. State of the Art through Systematic Review. http://lapes.dc.ufscar.br/tools/start_tool, 2012.

[31] Silva J.; Tamarit, S; Tomás, C. System dependence graphs in sequential erlang. In Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering (FASE'12). p.486-500, Tallinn, Estonia, Springer-Verlag, 2012.

[32] Sarmanho, F.; Souza, P. S. L.; Souza, S. R.; Simao, A. S. Structural testing for semaphore-based multithread programs. In: ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–346.

[33] Souza, S. R. S.; Vergilio, S. R.; Souza, P. S. L.; Simão, A. S.; Hausen, A. C. Structural testing criteria for message-passing parallel programs. Concurrency and Computation: Practice and Experience, p. 1893–1916, 2008.

[34] Souza, P. S. L.; Souza, S. R. S.; Rocha, M. G.; Prado, R. R.; Batista, R. N. Data flow testing in concurrent programs with message-passing and shared-memory paradigms. In: ICCS - International Conference on Computational Science, Barcelona, Espanha, 2013b, pp. 149–158.

[35] Souza, P. S. L.; Souza, S. R. S.; Zaluska, E. Structural testing for message-passing concurrent programs: an-extended test model. Concurrency and Computation, v. 26, n. 1, pp. 21–50, 2014.

[36] Tasharofi, S.; Pradel, M.; Lin, Y. and Johnson, R. Bita: Coverage-guided, automatic testing of actor programs. In 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), 2013.

[37] Tate, Bruce A. Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages. Pragmatic Bookshelf, 2010.

[38] Taylor, R. N.; Levine, D. L. and Kelly. C. D. Structural testing of concurrent programs. IEEE Tr Softw Eng, 18(3):206–215, 1992.

[39] Tóth, M.; Bozó, I.; Horváth, Z.; Lövei,L.; Tejfel, M.; Kozsik, T. Impact Analysis of Erlang Programs Using Behaviour Dependency Graphs. Proceedings of the 3th Conference on Central European Functional Programming School, p.372-390, Komarno, Slovakia, May 25-30, 2009.

[40] Tóth, M. and Bozó I. Building dependency graph for slicing erlang programs. Conference of PhD Students in Computer Science, Periodica polytechnica, 2010.

[41] Tóth, M.; Bozó, I. Static analysis of complex software systems implemented in erlang, Proceedings of the 4th Conference on Central European Functional Programming School. Budapest, Hungary, June 14-24, 2011.

[42] Trobec, R.; Vajteršic, M.; Zinterhof, P. Parallel computing: Numerics, applications, and trends. Parallel Computing: Numerics, Applications, and Trends. Springer, 2009.

[43] Watt, D. A. Programming Languages: Concepts and Paradigms. Prentice Hall International Series in Conputer Science, 1990.

[44] Widera, M. Flow graphs for testing sequential Erlang programs. In Proceedings of the 3rd ACM SIGPLAN Erlang Workshop. ACM Press, 2004.

[45] Widera, M. Towards flow graph directed testing of functional programs. In Draft Proceedings of the 15th International Workshop on the Implementation of Functional Languages, IFL, 2003.

[46] Widera, M. Concurrent Erlang flow graphs. In Proceedings of the Erlang/OTP User Conference 2005, Stockholm, 2005.

[47] Widera, M. Data flow coverage for testing Erlang programs. In Marko van Eekelen, editor, Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP'05), September 2005.

[48] Widera, M. Data flow considerations for source code directed testing of functional programs. In H.-W. Loidl, editor, Draft Proceedings of the Fifth Symposium on Trends in Functional Programming, Nov. 2004.

[49] Widera, M. Flow graph interpretation for source code directed testing of functional programs. In Implementation an Application of Functional Languages, 16th International Workshop, IFL'04. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universit☐at zu Kiel, 2004.

[50] Widera, M. Adapting structural testing to functional programming. In International Conference on Software Engineering Research and Practice (SERP 06), 86-92. CSREA Press, 2006.

[51] Widera, M. Why Testing Matters in Functional Programming. 7th Symposium on Trends in Functional Programming, University of Nottingham, TFP, 2006.

[52] Yang, C.-S. D.; Souter, A. L. and Pollock, L. L. All-du-path coverage for parallel programs. In ISSTA, pages 153–162, 1998.