6th International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST 2015)

Proceedings

Tanja Vos, Sigrid Eldh, and Wishnu Prasetya

August 30-31, 2015
Bergamo, Italy

The Association for Computing Machinery, Inc.
2 Penn Plaza, Suite 701
New York, NY 10121-0701

ACM ISBN: 978-1-4503-3813-4

Additional copies may be ordered prepaid from:

	Phone: 1-800-342-6626
ACM Order Department	(U.S.A. and Canada)
P.O. BOX 11405	+1-212-626-0500
Church Street Station	(All other countries)
New York, NY 10286-1405	Fax: +1-212-944-1318
	E-mail: acmhelp@acm.org

Production: Conference Publishing Consulting
D-94034 Passau, Germany, info@conference-publishing.com

Message from the Chairs

Welcome to the **Workshop on Automated Software Testing (A-TEST)**, in Bergamo, Italy, 30th -31st August 2015. This workshop is co-located with 10th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), 2015.

Software testing is at the moment the most important and mostly used quality assurance technique applied in industry. Considering the activities that make up the testing life-cycle, test case design, selection and evaluation is the activity that determines the quality and effectiveness of the whole testing process. These are, however, the most difficult, time-consuming and error-prone activities during testing. Much of these activities are still carried out manually, and the quality of the resulting tests is sometimes low since they fail to find important errors in the system.

A-TEST workshop aims to provide a venue for researchers as well as the industry to exchange and discuss trending views, ideas, state of the art work in progress, and scientific results on topics such as:

- Techniques and tools for automating test case design and selection, e.g. model-based, combinatorial, search-based, or symbolic approaches.
- Test cases optimization.
- Test cases evaluation and metrics.
- Test cases design, selection, and evaluation in emerging test domains, e.g. Graphical User Interface, Social Network, Cloud, Big Data, Games, or Security.
- Case studies that have evaluated an existing technique or tool on real systems, not only toy problems, to show the quality of the resulting test cases compared to other approaches.

The workshop welcomes submissions in the form of full paper (10 pages) describing original and completed research, either empirical or theoretical, or an industrial case study; as well as submissions in the form of work-in-progress paper (4 pages) that describes novel, interesting, and highly potential work in progress, but not necessarily reaching its full completion.

There were 12 submissions, each is peer-evaluated by at least three reviewers, with at least one reviewer who are highly confident in the subject of the paper. Six full papers and one work-in-progress paper are accepted for presentation in the workshop.

We are grateful to all authors for their submissions to A-TEST 2015, and to the Program Committee members for their valuable time and effort in reviewing the submitted papers. Thank you, and we hope that you enjoy this year's workshop.

(Tanja Vos, Sigrid Eldh, Wishnu Prasetya. and Anna Esparcia)

A-TEST 2015 Organization

Organizing Committee

Contents

Model-Driven Test Case Design for Model-to-Model Semantics Preservation

Christopher Gerking
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
christopher.gerking@upb.de

Jan Ladleif
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
jladleif@mail.upb.de

Wilhelm Schäfer
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
wilhelm@upb.de

ABSTRACT

Model transformations used in model-driven software development need to be semantics-preserving, i.e., the meaning of a model must not be distorted by the transformation. Testing whether a transformation preserves the dynamic semantics of a model requires oracles such as model checkers, which explore the runtime statespace of models. The high amount of repetitive code to integrate heterogeneous transformation engines and test oracles makes the design of semantics preservation tests a tedious task. In this paper, we apply the approach of model-driven testing to the domain of model transformation. We present a visual domain-specific language for the design of model transformation tests, which reduces test cases to their essential components. Our language enables an immediate execution of test cases with precise validation feedback. We evaluate our approach in terms of a case study based on the MechatronicUML modeling language for the software development of cyber-physical systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.9 [**Software Engineering**]: Management—*software quality assurance*; I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*model validation and analysis*

General Terms

Design

Keywords

Model transformation, test case design, semantics preservation

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804323

1. INTRODUCTION

Model transformations are an integral part of contemporary model-driven software development (MDSD) processes. They play the vital role of bridging the gap between platform-independent models and concrete execution platforms, generating executable software systems from abstract specifications. In order to ensure that a generated software system meets its specified requirements, model transformations in MDSD need to be *semantics-preserving*, i.e., the meaning of a model must not be distorted (only refined) by the transformation. However, proof techniques for semantics preservation during model transformations are still in the early stages of development [10]. Therefore, viable quality assurance for model transformations reduces mainly to testing approaches [2].

Test cases for model-to-model semantics preservation are characterized by a heterogeneous infrastructure in terms of tools, technologies, and methods involved. On the one hand, a variety of special-purpose model transformation languages exists [3], and requires to invoke specific transformation engines during the execution of test cases. On the other hand, different requirements in terms of testing precision give rise to various kinds of test oracles [16], which are consulted during the execution of test cases in order to assess the correctness of the transformation result. For example, validating the output model against syntactic constraints could be a sufficient oracle, when the primary goal is to exclude semantic invalidity. In contrast, model comparison [13] represents an established and more restrictive test oracle, demanding syntactic equality between the output model and a carefully selected, manually certified reference model. As syntactic equality implies semantic equivalence, model comparison is sufficient to test semantics preservation between output and reference model.

However, syntactic equality is far from a necessary condition for semantics preservation, because two disparate models can still be equivalent with respect to a selection of representative semantic properties. Therefore, model comparison often appears as a too strict oracle for the preservation of semantic properties during model-to-model transformations. Especially when the model transformation is work in progress, the structure of the output model might change frequently and cause *false positive* test failures. MDSD suffers from this problem in particular, as it involves behavioral models with intrinsic dynamic semantics, which define the runtime execution behavior of a software system. Static reasoning at syntax level is inappropriate to argue about these dynamic semantic properties. Hence, testing semantics preservation efficiently requires a test oracle that operates beyond the syntax level, and analyzes output models in terms of their dynamic semantics. Thus, oracles need to explore the runtime statespace of the models involved, using dedicated tools for model simulation or model checking [7].

Test cases for model-to-model semantics preservation require a considerable amount of repetitive *glue code*, usually written in a general-purpose programming language that supports the integration of heterogeneous technologies. For example, a test case could invoke a specific transformation engine to transform an input model into an output model, before invoking a specific model checker to analyse the output model for certain semantic properties. Integrating such heterogeneous technologies makes the test case design for model transformations a tedious task, because the repetitive integration code is irrelevant to the essential logic of the test cases.

In this paper, we apply the approach of *model-driven testing* to the domain of model transformation. We present a visual domain-specific language (DSL) for the design of model transformation tests, which abstracts from irrelevant details and reduces test cases to their essential components. Our DSL supports the visual flow-based modeling of test cases, and enables to specify the flow of models between different components, while abstracting from the concrete execution order. The approach also enables an immediate test execution with precise visual validation feedback. We evaluate our approach in terms of a case study based on the MechatronicUML modeling language for the software development of cyber-physical systems [4].

In summary, the contribution of this paper is (i) a language concept for the model-driven design of test cases for semantics preservation during model-to-model transformations, and (ii) a visual DSL as an application of our concept, enabling the design and execution of semantics preservation test cases in the context of MechatronicUML.

Our paper is structured as follows: Section 2 describes our model-driven testing approach for model transformations. In Section 3, we demonstrate our approach in terms of a visual DSL for the design and execution of test cases. We discuss related work in Section 4, before concluding in Section 5.

2. MODEL-DRIVEN TESTING OF MODEL TRANSFORMATIONS

Common to the typical components of model transformation test cases (e.g., loading test models, invoking transformation engines, or consulting oracles) is their usage of models as inputs or outputs. Hence, regardless of the high amount of repetitive *glue code* that is usually required to integrate heterogeneous components, they share a common type of interface in terms of models. Based on the observation of models as primary interfaces between components, we abstract from their technological distinctions and apply the approach of *model-driven testing* to the domain of model transformation. In Section 2.1, we present a formal design approach for model transformation test cases in terms of a domain-specific modeling language. Based on this modeling approach, Section 2.2 describes the execution of test cases and how to determine the result of an execution.

2.1 Test Case Design

The core elements of a test case are its components, which we model using nodes: Each node represents one specific action, such as loading a model or verifying assertions. To accomplish its task, a node exhibits individual input ports from which it receives data. After its execution, it may issue results to its individual set of output ports. These, in turn, can be connected to input ports of other nodes, yielding a dataflow network. As we specify test cases, each node may also fail: If a node observes unexpected behavior or finds its assertions are incorrect, it issues a relevant error message and triggers the whole test case's failure. Formally, such a system can be summarized as follows:

Definition 1. A *test case* C is a 7-tuple $(V, I, O, D, opt, exe, L)$ with

- the set of nodes V,
- the set of all input ports I,
- the set of all output ports O,
- the dataflow relation $D \subseteq O \times I$,
- a function $opt : I \rightarrow \{true, false\}$ determining if an input is optional,
- a function $exe : V \rightarrow \{true, false\}$ determining if the execution of that node was successful, and
- a function $L : (V \cup I \cup O) \rightarrow \Sigma$ assigning labels to nodes and ports, with Σ the set of labels.

For a node $v \in V$ let $I(v) \subseteq I$ be its disjoint set of input ports and $O(v) \subseteq O$ its disjoint set of output ports.

The above definition describes all possible test cases, but also includes invalid ones that cannot be executed. For example, one could define an input port that is part of more than one node. To amend this, we introduce the notion of *valid* test cases:

Definition 2. A test case $C = (V, I, O, D, opt, exe, L)$ is valid iff

- every input port (analog for output ports) is assigned to exactly one node:
  $\forall i \in I (\exists!\ v \in V\ (i \in I(v)))$,
- the input ports (analog for output ports) have unique labels:
  $\forall v \in V\ (i_1, i_2 \in I(v) \Rightarrow (i_1 = i_2 \vee L(i_1) \neq L(i_2))$,
- each input port is connected to at most one output port:
  $\forall i \in I\ \neg(\exists o_1, o_2 \in O\ ((o_1, i) \in D \wedge (o_2, i) \in D))$,
- every non-optional input port is connected to at least one output port:
  $\forall i \in I\ (opt(i) = false\ \Rightarrow\ \exists o \in O : (o, i) \in D)$ and
- the dataflow relation D is acyclic, i.e., you cannot arrive at the same node using dataflows after leaving it through an output port.

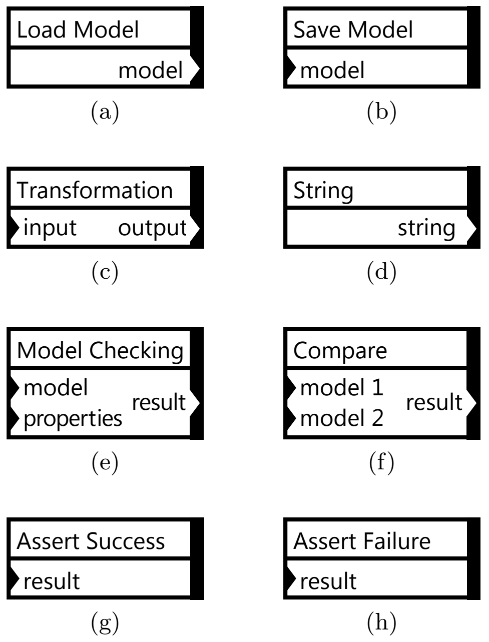


Figure 1: Different Types of Nodes Used to Test Model Transformations

The number and type of ports as well as the concrete nature of the function $exe(v)$ of a node $v \in V$ depend on its *type*. Figure 1 shows a variety of node types that we conceived as part of a visual DSL to test model transformations: Loading (a) and saving (b) models is essential, as is executing model transformations (c). A way to specify and output arbitrary strings (d) is required too, mainly to parametrize model-to-model transformations. Also, a model checker should be available (e) and two models should be comparable on a syntactical level (f). Lastly, the results of the model checking and comparison can either be asserted to be a success (g) or a failure (h).

2.2 Execution of Test Cases

Ultimately, all nodes of a test case should be executed. The order of the nodes' execution is not an arbitrary decision, however. One has to take into account the dependencies that are implied by the dataflow relation D.

Definition 3. A node $v_2 \in V$ directly depends on $v_1 \in V$ ($v_1 \rightsquigarrow v_2$) iff a dataflow $(o, i) \in D$ with $o \in O(v_1)$ and $i \in I(v_2)$ exists. The transitive closure $\rightsquigarrow^*$ of $\rightsquigarrow$ contains all dependencies.

A dependency $v_1 \rightsquigarrow^* v_2$ implies that v_1 has to be executed before v_2. This is the case whenever a node directly or indirectly requires the output of another one for its own computations. A correct order of execution needs to respect all dependencies imposed by $\rightsquigarrow^*$ and actually always exists for valid test cases:

THEOREM 1. *For every valid test case $C = (V, I, O, D, opt, exe, L)$ there exists a topological sorting, i.e., a bijective mapping $ord : V \to \{1, ..., n\}, n = |V|$, such that*

$$v_1 \rightsquigarrow^* v_2 \Rightarrow ord(v_1) < ord(v_2)\ \forall v_1, v_2 \in V\ .$$

PROOF. The dependency relation $\rightsquigarrow^*$ defines a strict partial order over V: It is irreflexive because we required acyclicity in Definition 2, and transitive because it is defined as a transitive closure (see Definition 3). Thus, the implied graph $G = (V, \rightsquigarrow^*)$ is a directed acyclic graph (DAG). For every DAG a topological sorting of its nodes exists, which in particular yields a topological sorting for every valid test case. □

There are various canonical algorithms to calculate such a topological sorting, e.g., by Kahn [11]. Once a topological sorting has been acquired, a test case can be executed in its entirety. Every node has to finish successfully in order for the whole test case to be regarded a success:

Definition 4. A test case $C = (V, I, O, D, opt, exe, L)$ with a topological sort ord is successful iff

$$\bigwedge_{j=1}^{n} exe(ord^{-1}(j)) = true\ .$$

3. CASE STUDY

The goal of this case study is to demonstrate that our DSL enables an effective test case design and execution for model-to-model semantics preservation. For evaluating our approach, we consider the MECHATRONICUML domain-specific modeling language [4], which targets the model-driven software development for cyber-physical systems. MECHATRONICUML supports modeling of behavioral contracts for real-time coordination by means of Real-Time Statecharts, a combination of UML statemachines and timed automata. One of the key features of MECHATRONICUML is the verification of these contracts against temporal logic safety properties (e.g., deadlock freedom) by means of model checking. To this end, MECHATRONICUML provides a model-to-model transformation which translates Real-Time Statecharts into timed automata that can be analyzed by the model checker UPPAAL [5]. In order to ensure reliable results, the transformation needs to preserve the semantics of the input Real-Time Statecharts, i.e., the output timed automata need to be semantically equivalent. In Section 3.1, we describe our prototypical implementation of a domain-specific testing language in the context of MECHATRONICUML, using Eclipse and a variety of its tools. Afterwards, in Section 3.2, we evaluate our approach by testing the model transformation from MECHATRONICUML to UPPAAL for semantics preservation.

3.1 Implementation

The architectural basis is laid out by two separate metamodels which we model using the Eclipse Modeling Framework (EMF, [19]). Figure 2 shows an overview of our architecture. As EMF is a commonly used standard framework for model-driven software development it makes our test case models easily usable, allowing for a straightforward integration with existing software. The execution logic is added to the metamodel by taking advantage of the EMF Validation Framework[1]: We supply our own strategy to calculate a topological sorting (see Theorem 1), which the EMF Validation Framework uses to execute our nodes in the correct order. Furthermore, a graphical editor implementing our

[1]`https://projects.eclipse.org/projects/modeling.emf.validation`

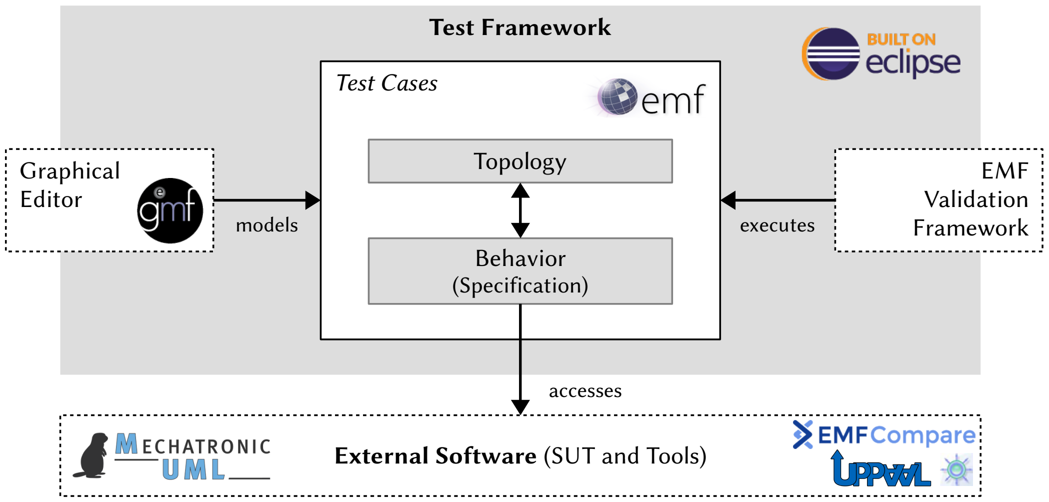


Figure 2: Relationship Between Components in our Framework's Architecture

concrete syntax (see Figure 1) is realized using the Graphical Modeling Framework[2] (GMF).

One metamodel (labeled *Topology*) contains all the topological aspects of the test cases, while the other (labeled *Behavior*) uses a strategy pattern to easily define and implement new node types. They are intertwined such that each topological node has access to its particular set of behavioral instructions. The EMF Validation Framework accesses the topological level to calculate the topological sorting, and afterwards the behavioral level to execute the test case.

The nodes themselves may access any external tool that they need to perform their computations. In our case, we implement the node types given in Figure 1. For this we employ QVTo, an Eclipse integration of the QVT Operational Mappings model transformation standard [18], UPPAAL as an external model checker, and EMF Compare [6] to compare arbitrary EMF models. Additionally, our implementation is tailored for use with the MECHATRONICUML tool suite. It supports the specification of temporal logic properties using a domain-specific variant of the Timed Computation Tree Logic [1], called MTCTL, as well as the transformation of MECHATRONICUML models to UPPAAL-compatible timed automata in order to conduct model checking.

3.2 Evaluation

The evaluation of our approach is based on the guidelines for case studies by Kitchenham et al. [12]. We consider four different MECHATRONICUML software models of interconnected transportation systems (e.g., autonomous cars, trains, or miniature robots). Our models include an overall amount of thirteen contracts for real-time coordination behavior such as overtaking or collision avoidance. All contracts are equipped with temporal logic verification properties expressed using MTCTL. According to the MECHATRONICUML semantics, all the attached properties hold on the given models. Our expectation is that the transformation from MECHATRONICUML to UPPAAL preserves these semantics. Thus, the evaluation hypothesis for our evaluation is that our approach allows to design test cases which transform the given MECHATRONICUML models to UPPAAL, and then check for semantics preservation by model checking the given properties on the output timed automata. To this end, we also prepare an erroneous variant of our model transformation, which deliberately introduces semantic distortions between input and output model. We regard our hypothesis as fulfilled if the execution of our test cases clearly separates the semantics-preserving model transformation from the semantics-distorting variant.

Figure 3 illustrates the pattern that we used to design our test cases. Using a *Load Model* node, we first load one of the exemplary MECHATRONICUML models which already includes a temporal logic property (meaning that once a certain state `x` becomes active, the state `y` will invariably be reached). A *String* node is used to specify the name of the particular coordination contract to transform in a particular test case. Both nodes act as inputs to a third node of type *Transformation*, which represents the execution of our model transformation from MECHATRONICUML to UPPAAL using the QVTo engine. The two outputs of the transformation (a network of UPPAAL timed automata, and the translated TCTL properties) connect to a node of type UPPAAL, which invokes UPPAAL's command line verification tool. Finally, we use an *Assert Success* node to express that the expected model checking result is *true* for all verified properties.

According to the above pattern, we design one test case for each of the thirteen contracts to test. Initially, the *Transformation* nodes in all our test cases refer to the semantics-preserving variant of our model transformation from MECHATRONICUML to UPPAAL. After the test case design, we run all our test cases using our integration with the EMF Validation Framework described in Section 3.1. We observe that our tests run successfully, as the final *Assert Success* node in each of our test cases can be executed without any deviations from our specified expectations. In the next step, we redesign all of our test cases to refer to the semantics-distorting variant of our model transformation. Again, we execute all of our test cases and observe the test results. All of our thirteen test cases fail after switching to the semantics-distorting model transformation, because at least one of the specified MTCTL properties can not be ver-

[2]`http://www.eclipse.org/modeling/gmp/`

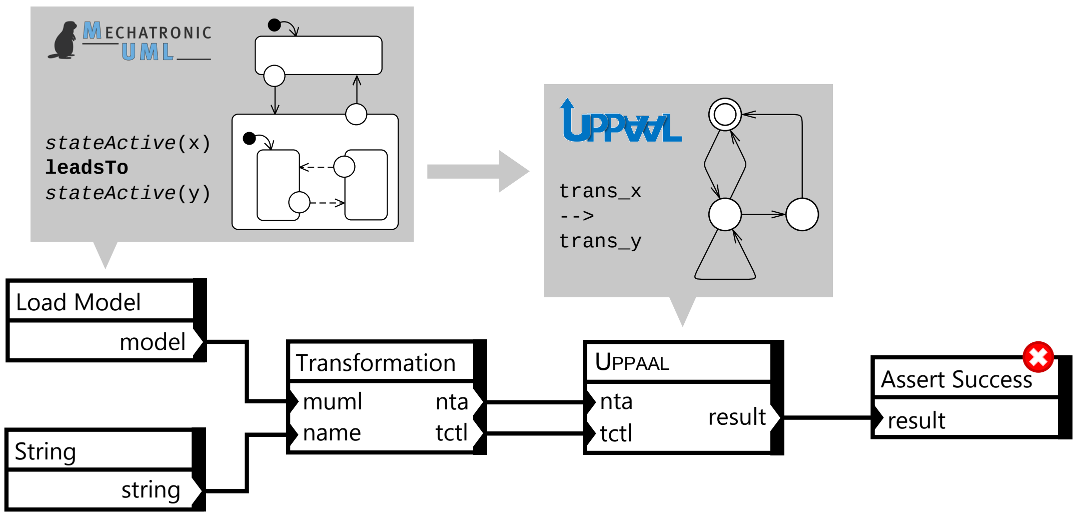


Figure 3: Failing Test for Semantics Preservation of a Model-to-Model Transformation

ified successfully by the Uppaal model checking. Figure 3 depicts a failing execution of an exemplary test case. The graphical editor gives a precise feedback, by marking the *Assert Success* node as the point of failure.

In summary, our case study successfully separates the semantics-preserving model transformation from its semantics-distorting variant. We therefore regard our evaluation hypothesis as fulfilled, and thus conclude that our approach enables the effective testing of model-to-model semantics preservation.

4. RELATED WORK

In this section, we discuss related work in terms of existing frameworks for model transformation tests, as well as alternative testing techniques for semantics preservation.

Küster et al. [14] describe four test design techniques for the incremental development of model transformations, and discuss their integration into a test framework. Whereas our DSL covers several of these techniques (such as integrity testing against the syntactic constraints induced by the target metamodel, or model comparison against reference outcomes), none of the mentioned techniques explicitly addresses testing at the dynamic semantics level.

García-Domínguez et al. [8] present the EUnit framework for testing of model management tasks such as model transformations. Similar to our approach, they enable modeling of executable test cases by means of dataflow networks. In contrast to our approach, the Epsilon Object Language used for the textual specification of test cases is less abstract than our visual DSL. Although the presented framework is highly extensible, the authors do not explicitly address testing at the dynamic semantics level using model checking or comparable techniques.

Model transformation contracts [9] represent a contrary approach for testing model transformation outputs at syntax level. In general, a contract consists of syntax constraints over the input/output models, whereas one single constraint may also refer to both models and describe a certain relation between model elements. Thus, a contract may restrict the output model's syntax depending on specific syntactic characteristics of the input model, or vice versa. If the specified characteristics are stable (i.e., remain unaffected by changes to a transformation which is work in progress), contracts can reduce the number of false positive test failures in comparison to plain model comparison approaches [13]. Therefore, we regard the specification of contracts as a promising extension to our DSL for test case design. In particular, contracts specified in the scope of a trace model [15] could help to define more precise contracts by referring explicitly to the relations between particular input/output elements recorded during a transformation.

Varró and Pataricza [20] explicitly address the testing of dynamic semantics preservation by means of model checking. In contrast to our approach, they propose a model checking for both input and output models in order to compare the results. Whereas our DSL basically supports this design technique, model checking an input model given in terms of a DSL requires its dynamic semantics to be fully formalized and operational, which is usually a barrier to a successful implementation of the proposed technique. In comparison, we focus on model checking only the output model.

Narayanan and Karsai [17] analyze the semantic equivalence of particular input/output models with respect to a given property. To this end, they check the bisimilarity between particular runtime snapshots, and therefore require an exploration of the runtime statespace for both models. In contrast, whereas our approach explores the statespace of the output model as well, it increases the applicability by employing a general-purpose model checking tool for this task.

5. CONCLUSION AND FUTURE WORK

In this paper, we propose a model-driven design approach for semantics preservation tests in the scope of model-to-model transformations. We provide a concept for a domain-specific modeling language, which abstracts from the repetitive code required to integrate different technologies for loading test models, invoking transformation engines, or consulting oracles. Our modeling approach also enables the test execution with appropriate validation feedback.

Our case study reports the successful implementation of the aforementioned language concept in terms of a test design language in the context of MechatronicUML, a domain-specific modeling language for the software development of cyber-physical systems. We design a range of test cases including the transformation from MechatronicUML to timed automata, and the verification of particular temporal logic properties on these automata, using the Uppaal model checker as test oracle. The execution of these test cases successfully separates the semantics-preserving model transformation from a semantics-distorting variant.

Design engineers for test cases benefit from our approach, as they require less effort to create executable test cases that integrate different technologies. Both our language concept and implementation are highly extensible in terms of different transformation engines/languages, or alternative tools used as oracles.

Future work on our approach encompasses the integration of alternative testing techniques to our DSL. As discussed in Section 4, model transformation contracts [9] represent a promising approach towards testing model transformations by specifying syntactic relations between input/output models. Especially promising is the approach of specifying such contracts in the scope of a trace model [15], which explicitly relates particular input/output elements and therefore enables a more precise contract definition. Additionally, future work includes design support for parametrized test cases, differing only in terms of their particular input data. Our evaluation demonstrated that test case design often reduces to a common pattern, such that designers highly benefit from a parametrized approach.

6. ACKNOWLEDGMENTS

We thank Stefan Dziwok for providing test models for our case study. Christopher Gerking is member of the PhD program "Design of Flexible Work Environments – Human-Centric Use of Cyber-Physical Systems in Industry 4.0", supported by the federal state of North Rhine-Westphalia.

7. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[2] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model transformation testing challenges. In H. Eichler and T. Ritter, editors, *Proceedings of the ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing.* Fraunhofer IRB, 2006.

[3] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. Le Traon, and J. Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6):139–143, 2010.

[4] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann. The MechatronicUML method: Model-driven software engineering of self-adaptive mechatronic systems. In P. Jalote, L. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering (ICSE Companion 2014)*, pages 614–615, New York, 2014. ACM.

[5] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal – a tool suite for automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 232–243, Berlin/Heidelberg, 1996. Springer.

[6] C. Brun and A. Pierantonio. Model differences in the Eclipse Modelling Framework. *CEPIS Upgrade*, 9(2):29–34, Apr. 2008.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge/London, 2000.

[8] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo. EUnit: A unit testing framework for model management tasks. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011*, volume 6981 of *LNCS*, pages 395–409, Berlin/Heidelberg, 2011. Springer.

[9] M. Gogolla and A. Vallecillo. *Tract*able model transformation testing. In R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige, editors, *Modelling Foundations and Applications, 7th European Conference, ECMFA 2011*, volume 6698 of *LNCS*, pages 221–235, Berlin/Heidelberg, 2011. Springer.

[10] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Showing full semantics preservation in model transformation – a comparison of techniques. In D. Méry and S. Merz, editors, *Integrated Formal Methods, 8th International Conference, IFM 2010*, volume 6396 of *LNCS*, pages 183–198, Berlin/Heidelberg, 2010. Springer.

[11] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, Nov. 1962.

[12] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995.

[13] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: A foundation for model composition and model transformation testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, pages 13–20, New York, 2006. ACM.

[14] J. M. Küster, T. Gschwind, and O. Zimmermann. Incremental development of model transformation chains using automated testing. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*, volume 5795 of *LNCS*, pages 733–747, Berlin/Heidelberg, 2009. Springer.

[15] N. D. Matragkas, D. S. Kolovos, R. F. Paige, and A. Zolotas. A traceability-driven approach to model transformation testing. In B. Baudry, J. Dingel, L. Lucio, and H. Vangheluwe, editors, *Proceedings of the Second Workshop on the Analysis of Model Transformations, (AMT 2013)*, volume 1077 of *CEUR Workshop Proceedings*, 2013.

[16] J.-M. Mottu, B. Baudry, and Y. Le Traon. Model transformation testing: oracle issue. In *2008 IEEE International Conference on Software Testing, Verification and Validation Workshop (ICSTW'08)*, pages 105–112. IEEE, 2008.

[17] A. Narayanan and G. Karsai. Towards verifying model transformations. *Electronic Notes in Theoretical Computer Science*, 211:191–200, Apr. 2008.

[18] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Number formal/15-02-01. 2015.

[19] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2nd edition, 2008.

[20] D. Varró and A. Pataricza. Automated formal verification of model transformations. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors, *2rd International Workshop on Critical Systems Development with UML (CSD-UML 2003)*, pages 63–78, 2003.

A Test Model for Graph Database Applications: An MDA-Based Approach

Raquel Blanco
Department of Computing
University of Oviedo
Gijón, Spain
rblanco@uniovi.es

Javier Tuya
Department of Computing
University of Oviedo
Gijón, Spain
tuya@uniovi.es

ABSTRACT
NoSQL databases have given rise to new testing challenges due to the fact that they use data models and access modes to the data that differ from the relational databases. Testing relational database applications has attracted the interest of many researchers; but this is still not the case with NoSQL database applications. The approach presented in this paper defines a test model for graph database applications that takes into account the data model of this technology and the system specification. To automate the derivation of the test cases and the evaluation of their adequacy we propose a framework that places model-based testing into the model-driven architecture context.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords
Graph database testing, model-based testing, MDA, specification-based testing.

1. INTRODUCTION
Databases are probably the most important asset of an organization, and constitute the core of most software systems. Nowadays, many organizations need to store a vast amount of information, and they are increasingly turning to NoSQL databases to manipulate this large amount of data with higher performance [21].

There are numerous NoSQL technologies (currently 150) [28], which are classified into four popular types according to their data model [26]: key-value, document-based, column-family and graph databases. These types of database have something in common: they do not require a schema that restricts the data that can be stored.

Testing NoSQL database applications is a crucial and a challenging process for several reasons. On the one hand, NoSQL technologies do not work with SQL and each one uses its own APIs and tailored query languages, which are not as widely known as SQL by the developers. Moreover, the programming of complex queries can be difficult [21]. In particular, queries of graph database technologies can be especially verbose and difficult to write, understand and maintain [2]. Due to these difficulties, faults can appear in the code that accesses the database.

On the other hand, despite the fact that NoSQL databases do not require a schema, the applications usually have an underlying conceptual model that represents the data stored (henceforth *conceptual data model*). As there are no constraints that restrict their storage, the physical database could contain data that do not satisfy the conceptual data model. These data can produce application malfunctions and/or incorrect outputs to the user.

To test database applications, many approaches have been developed, such as [7], [9], [13], [15], [24]. However, as these works rely on SQL statements and/or the existence of an explicit database schema, they cannot be applied to testing NoSQL database applications. So, it is necessary to develop new testing approaches for this type of applications, which take into account the new data models and specific characteristics of each NoSQL technology.

The scope of this paper is the development of an approach to test graph database applications that considers the data model characteristics of this technology. Data are stored in nodes and relationships among nodes, and both nodes and relationships can contain properties. The graph databases are gaining in popularity and thousands of organizations use them in applications such as social recommendations, logistics, fraud detection, identity and access management, etc. [27]. To achieve this goal, we propose a model-based testing approach in the context of model-driven architecture, so that we benefit from the support of automation of both paradigms.

The main contributions of this work are:

- The definition of a framework that integrates model-based testing (MBT) into the model-driven architecture (MDA) paradigm.
- The definition of a test model for graph database applications that relies on both the underlying conceptual data model and the system specification.

The remainder of this paper is organized as follow: Section 2 presents the related work. Sections 3 and 4 describe the architecture of our MBT/MDA framework and the test model, respectively. Section 5 presents a case study. The paper ends with conclusions and future work.

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804324

2. RELATED WORK

2.1 Testing Database Applications

Several approaches in the literature address the problem of testing database applications. To guide the generation of test inputs and evaluate their adequacy, several criteria have been developed. Works that define program-based adequacy criteria range from criteria for procedural code that take into account the SQL queries [10], to criteria specially designed to deal with the SQL statements [14], [15], [32], [33], [35] and tools to automate the criteria [16], [31], [39]. Other works define specification-based adequacy criteria, such as [4]. The generation of test inputs has been addressed in several works: [3], [23], [36] generate test databases and [7], [24] both test database and program inputs.

With regard to testing the database schema, works are focused on defining adequacy criteria [25], [37], generating data to test the schema constraints [17] or prioritizing the test cases when the database schema changes [12], [13].

As stated before, these works depend on SQL code and/or explicit relational database schemas, while our approach is totally independent. The closest works to ours are those of [17], [37] as they use the database schema to generate test cases. These works are focused on testing the database schema. However, our approach uses a conceptual data model as the basis for designing the test model according to the system specification.

2.2 Model-Based Testing and Model-Driven Architecture

Model-based testing (MBT) has been used in several database testing works, such as [4], [9], [11], [18]. In MBT, the system is modelled to identify the important aspect to be tested regarding the expected system behaviour, obtaining a test model. Next, a test selection criterion is chosen to derive the abstract test cases, which are then concretized by means of a test generation technology and translated into executable test cases that can be run against the software under test (SUT) [34].

On the other hand, MBT can be placed into the MDA context, obtaining the abstraction levels PIT (Platform Independent Test) and PST (Platform Specific Test) [8]. The PIT level contains the test models that are platform independent, whereas at the PST level the test models contain information about the specific underlying platform.

Works in the MBT/MDA context are mainly focused on transforming the system model at the PIM level into the test model at PIT level [1], [5], [6], [19], [22], and defining transformations from the PIT level to the PST level and/or the test code [1], [20], [22], [38]. However, it is important to have some independence between the system models and the test models, because mistakes in the system models can be propagated to the code and the tests and, therefore, they are impossible to detect [30], [34]. In our approach the test model is designed by the testers, instead of being generated from a system model.

3. THE MBT/MBA FRAMEWORK

The architecture of the MBT/MDA framework we propose is depicted in Figure 1. At the PIT level, we have identified two important viewpoints: PITM (*Platform Independent Test Model*) and PITGM (*Platform Independent Test Generation Model*).

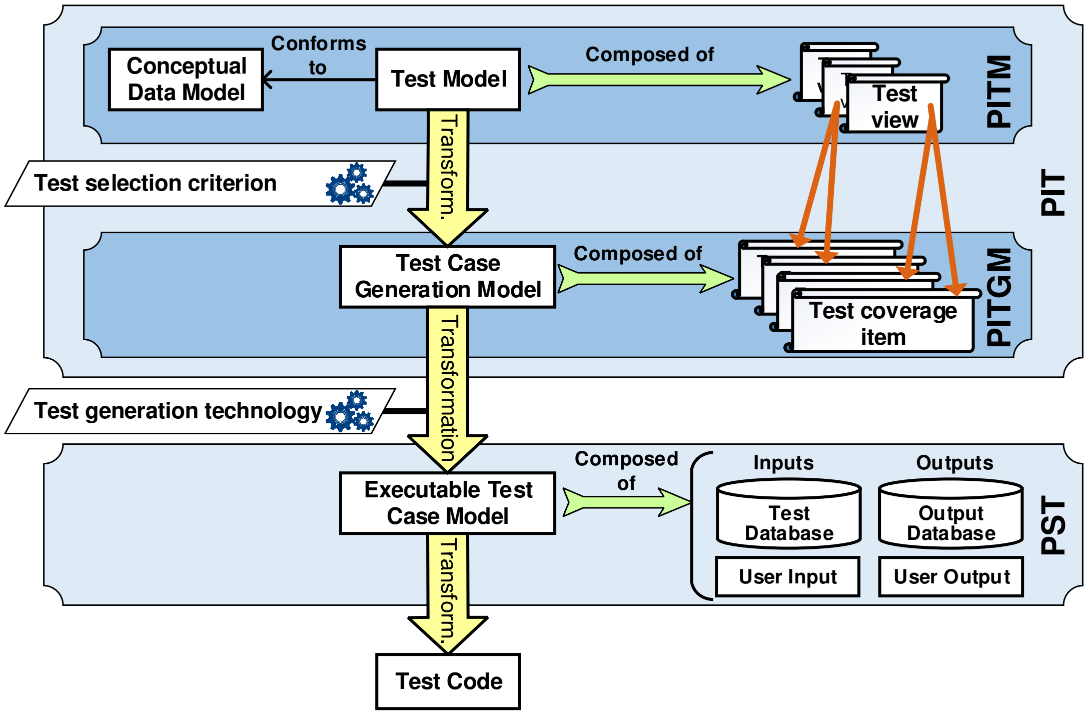


Figure 1. Architecture of the MBT/MDA framework

The PITM level is focused on the definition of the testing objectives, according to the system specification of the SUT. Here, the test model is designed as a composition of one or more important features of the SUT to be tested, called *test views*. Related to the scope of testing database applications, the conceptual data model of the database plays an important role, so the test model must conform to it in order to specify the test views correctly. If the conceptual data model is not explicitly stated (the NoSQL databases are schema-optional), the tester prepares this model as part of the testing process.

The PITGM level is centred on the definition of the test case generation model that is formed by the specific items that must be tested, which are called *test coverage items*. In the context of MBT, the test generation model represents a model of the abstract test cases. The mapping between the test model and the test generation model is performed by transformations that are guided by the test selection criterion chosen, which leads the test coverage items. In this mapping, a test view can give rise to several test coverage items. From a PITM, several PITGM can be automatically derived by appropriate transformations.

The PST level contains the executable test case model, which is obtained by means of transformations from the test case generation model and depends on the specific graph database management system used. These transformations are guided by some test generation technology that concretizes the test inputs, formed by the state of the database before the execution of the test case (test database) and the values supplied by the user (user input); and the expected outputs, formed by the state of the database after the execution of the test case (output database) and the values shown to the user (user output). Finally, the executable test cases can be transformed into an executable test code.

An important benefit of the MDA paradigm consists in reaching a high level of automation by defining transformations among models. In our framework, the tester specifies the test model and, after that, the processes of deriving the test case generation model, the executable test cases and the test code can be carried out automatically.

The elaboration of a test database with meaningful data is a determining factor, as these data are transformed to produce the test output and the test database has to represent the situations of interest to be tested, so the SUT can exercise them. This paper is focused on the definition of test views for unit testing, which are specially tailored for managing the database of graph database applications.

4. TEST VIEWS FOR GRAPH DATABASE APPLICATIONS

Consider, for example, a database application ("illness risk") which determines the level of risk of suffering an illness according to different factors such as the severity of previous episodes suffered by the person (which is classified in three levels), the existence of previous episodes of the illness in his/her family, etc.. The conceptual data model of the database is depicted in Figure 2.

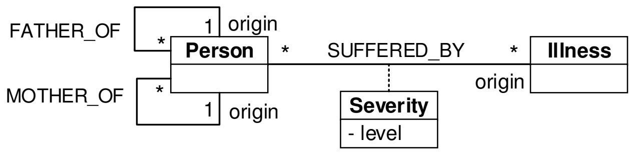
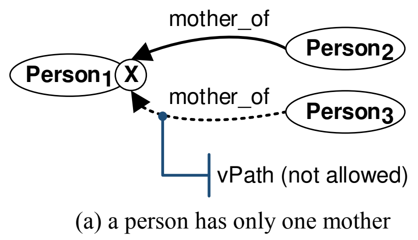


Figure 2. Conceptual data model of the "illness risk" application

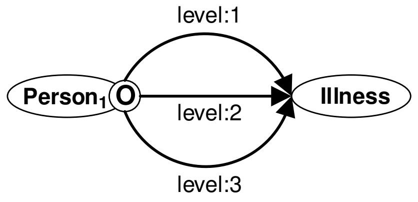
Some interesting features to test are situations in which: (a) a person has only one mother; (b) a person can suffer several episodes of the same illness with different severity levels; (c) an illness can be suffered by several people of the same family.

Our approach allows the tester to define test views based on the system specification, which indicate interesting nodes and relationships of the test database to test the application behaviour. Figure 3 depicts the test views that correspond to the aforementioned features to test. The elements that compose a test view are also identified:

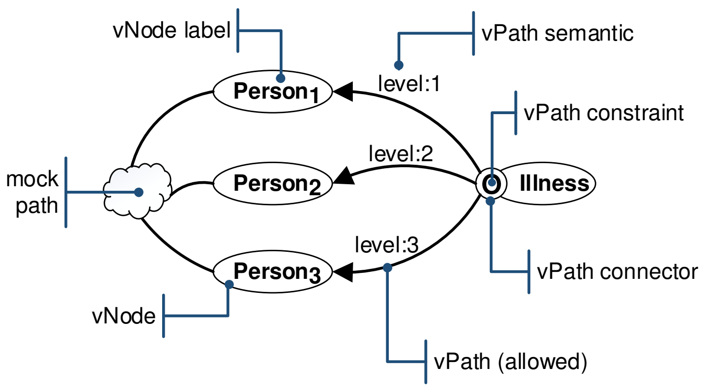
- *View node or vNode*: a type of node of the database. The *vNode label* indicates the class that represents the vNode in the conceptual data model. A type of node can be unique in a test view, generating only one vNode (like the vNode "Illness"), or have several instances, giving rise to several vNodes denoted by *class*$_i$, (the subscript represents the number of the instance of this vNode). For example, the vNodes "Person$_1$", "Person$_2$" and "Person$_3$" are three different instances of the same type of node "Person".
- *View path* or *vPath*: a directed path that relates two vNodes according to a specific semantic derived from the relationships of the conceptual data model, which is indicated by the label *vPath semantic*. There are two types of vPaths: allowed and not allowed, which specify that a vPath can appear or cannot appear in a database, respectively.
- *Mock path*: a not completely defined path that relates two or more vNodes. The testing objective is not focused on any specific path that relates these vNodes, but it is focused on its existence.
- *vPath constraint*: a restriction over a group of vPaths, which constraints whether each one can, cannot or must appear at the same time in the database. There are three types of vPath constraints: *XOR* (represented by "X") indicates that only one allowed vPath must appear in the database; *OR* (represented by "O") indicates that several allowed vPaths can appear at the same time in the database; *AND* (represented by "+") indicates that all allowed vPaths constrained must appear at the same time in the database.
- *vPath connector* (*connector*, for short): joins a group of vPaths that are restricted by the same vPath constraint. A connector can join vPaths that start in the same vNode or vPaths that end in the same vNode.



(a) a person has only one mother



(b) a person suffers several episodes of the same illness



(c) an illness can be suffered by several people of the same family

Figure 3. Examples of test views

The test view of Figure 3(a) indicates that the vPath between "Person$_2$" and "Person$_1$" must appear in the database, whereas the vPath between "Person$_3$" and "Person$_1$" cannot appear. Figure 3(b) indicates several vPaths that represent different severity levels of an illness. One or more of these vPaths can appear in the database between an instance of "Person" and an instance of "Illness". Finally, the test view of Figure 3(c) indicates that three different people have suffered an illness with different severity levels (vPaths from "Illness" to "Person$_1$", "Person$_2$" and "Person$_3$"). One or more of these vPaths can appear in the database. The mock path indicates that there can be family relationships between "Person$_1$", "Person$_2$" and "Person$_3$", but these relationships are not exactly defined.

After defining the test views, transformations guided by some test selection criterion can derive automatically the test coverage items. These test coverage items can be automatically mapped to executable test cases by means of transformations guided by a test generation technique.

Our approach allows the tester to define several types of test views, however, due to the lack of space we only present three examples.

5. CASE STUDY

To illustrate how our approach can be applied, a real-world example of a graph database application, called "authorization and access control" [29], has been used. This application represents the business of an international communications services company, which offers its customer organizations the ability of self-service their accounts. Organization administrators can add and remove services on behalf of their employees. To ensure that resources are only seen and changed by the entitled users, a complex access control system has been designed, considering different types of permissions and hierarchy structures among organizations. The conceptual data model of the database is depicted in Figure 4.

Administrators are assigned to one or several groups, and these groups have several permissions against the organizational structure. Each organization can be the parent of several organizations, with their own employees and accounts to manage. The permissions defined among groups and organizations are: (1) *allowed_inherit* allows administrators within the group to manage the accounts of both the organization and its children; (2) *allowed_do_not_inherit* allows the administrator with the group to manage the organization, but not its children; (3) *denied* forbids administrators with a group to manage the organization and its children. The access control system also establishes a permission precedence, because an administrator can be a member of two groups within different permissions against the same organizations. So, the permission *denied* takes precedence over *allowed_inherit*, and *allowed_do_not_inherit* prevails over *denied*.

The system specification defines three queries to find all accessible accounts for an administrator (shown in Figure 5), to determine whether an administrator has access to an account and to find all administrators for an account.

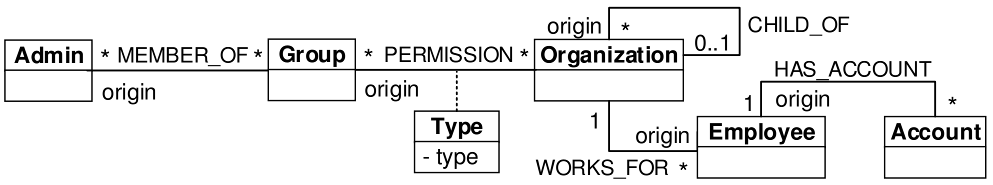


Figure 4. Conceptual data model of the "authorization and access control" application

```
START admin=node:administrator(name={administratorName})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->()
        <-[:CHILD_OF*0..3]-(company)<-[:WORKS_FOR]-(employee)
        -[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()<-[:CHILD_OF*0..3]-(company))
RETURN employee.name AS employee, account.name AS account
UNION
START admin=node:administrator(name={administratorName})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->()
        <-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
RETURN employee.name AS employee, account.name AS account
```

Figure 5. Cypher query for finding all accessible accounts for an administrator

First, we designed several test views, according to the system specification. One of them can be seen in Figure 6: a group can have different permissions against different organizations, which have a hierarchical structure. The "void" permission indicates that the group does not have an explicit permission against "Organization$_4$". The objective of this test view is to test the inheritance of the different types of permissions.

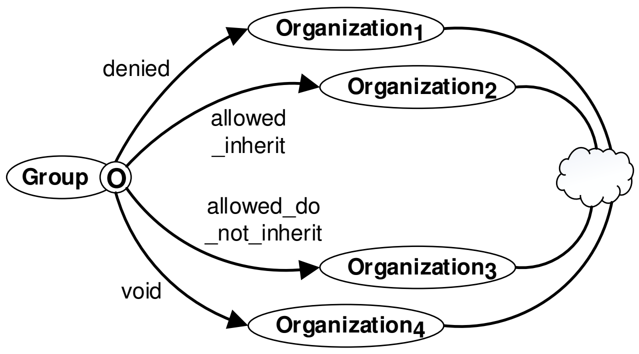


Figure 6. Test view of the "authorization and access control" application

Then, we transformed the test views into the test coverage items using a script that implements a combinatorial technique based on permutations without repetition. For example, for the test view of Figure 6 the script carried out permutations without repetition over the vNodes related by the mock path to generate different hierarchical orders between them. As a result, 24 test coverage items were generated automatically. Two of these test coverage items are shown in Figure 7. Note that the mock paths are now directed paths to indicate the particular hierarchical structure represented by the test coverage item.

From the test coverage items, we generated the test database, considering the specific graph database (Neo4j in our case [27]).

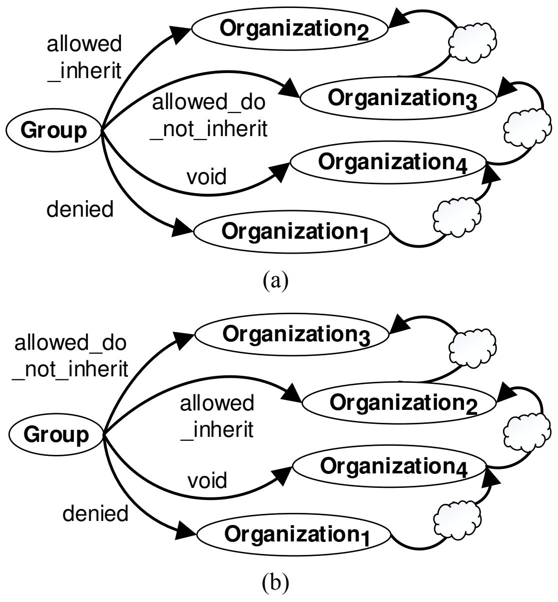


Figure 7. Test coverage items of the "authorization and access control" application

Figure 8 shows the nodes and relationships that were introduced into the test database to cover the test coverage items of Figure 7. The nodes "G1", "O1", "O2", "O3" and "O4" (and their relationships) cover the test coverage item of Figure 7(a), while the nodes "G1", "O5", "O6", "O7" and "O8" (and their relationships) cover the test coverage item of Figure 7(b). The other nodes and relationships were used to conform to the conceptual data model. Finally, we generated the test code that was executed against the SUT using the languages Cypher and Java. At present, the test database and the test code are generated by hand, however both tasks will be automated in the future.

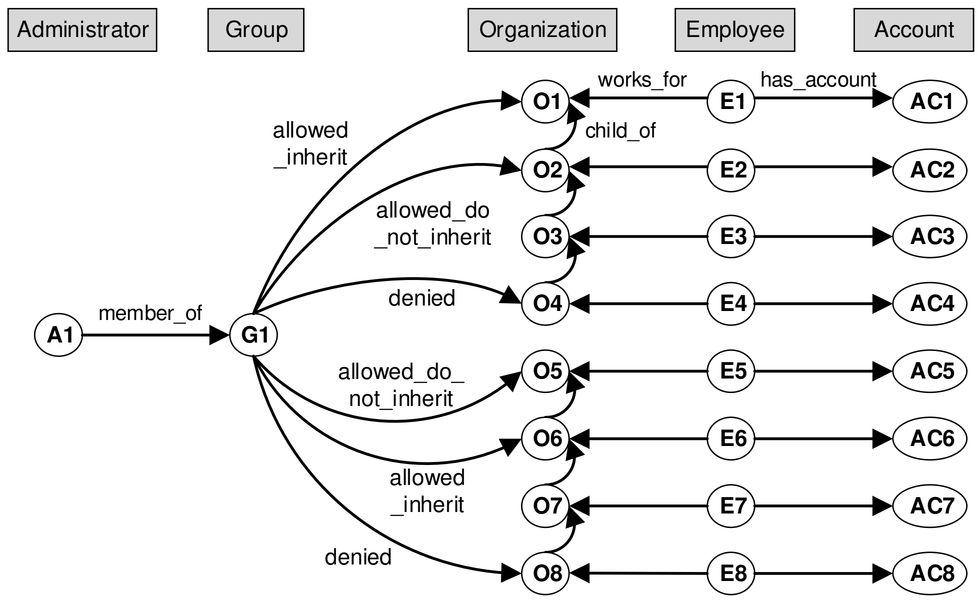


Figure 8. Extract of the test database of the "authorization and access control" application

The execution of the test cases, which take as input the test database generated, reported that "A1" has access to the accounts "AC1", "AC2", AC3", "AC5", "AC6" and "AC7", but should "A1" have access to the accounts "AC3", "AC6" and "AC7"? We do not know because the system specification does not indicate the preference between the *allowed_inherit* and the *allowed_do_not_inherit* permissions. So, the test cases detected a fault. If the observed output is equal to the expected output, the specification has a fault because it is incomplete. If the observed output is not equal to the expected output, both the specification and the implementation have a fault.

6. CONCLUSIONS AND FUTURE WORK

We have presented an approach to test graph database applications. This approach defines a test model taking into account the conceptual data model of the SUT and the system specification. The test model is composed of several test views that represent the important features of the SUT to be tested. To automate the generation of test cases from the test model we have proposed a framework that places MBT in the MDA context.

The results of the case study show that the test cases obtained from the test model reported that an administrator had access to some resources that could be forbidden (the system specification is not complete). An incomplete specification can cause defects in the applications, as developers could make erroneous assumptions about what the system must do; the increase of costs, since new code could be developed, and of course tested, when the omission is detected; and even the dissatisfaction of the customers, as the system does not meet their needs.

Future work includes several avenues. On the one hand, the definition of test selection criteria that consider the characteristics of the test views to derive the test coverage items and the development of techniques to generate executable test cases for graph database applications. Furthermore, the elaboration of the test views could be partially automated to represent different strategies and patterns of features that should be tested. At present, the generation of test coverage items has been automated, however other aspects can be automated, such as the transformations between the other models. As part of future work, we will define transformations between models that allow automating the process and we will develop a tool implementing the framework proposed.

7. ACKNOWLEDGMENTS

This work was supported by projects TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Science and Technology, and GRUPIN14-007, funded by the Principality of Asturias (Spain) and ERDF funds.

8. REFERENCES

[1] Alves, E.L.G., Machado, P.D.L., Ramalho, F. 2014. Automatic generation of built-in contract test drivers. *Software and Systems Modelling*, 13(3), 1141-1165.

[2] Barmpis, K., Kolovos, D.S. 2014. Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models. *Journal of Object Technology*, 13(2), pp 3:1-26.

[3] Binnig, C., Kossmann, D., Lo, E. 2008. MultiRQP - Generating test databases for the functional testing of OLTP applications. In *Proceedings of the 1st International Workshop on Testing Database Systems*.

[4] Blanco, R., Tuya, J., Seco, R.V. 2012. Test adequacy evaluation for the user-database interaction: a specification-based approach. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pp. 71-80.

[5] Busch, M., Chaparadza, R., Dai, Z., Hoffmann, A., Lacmene, L, Ngwangwen, T., Ndem, G., Ogawa, H., Serbanescu, D., Schieferdecker, I., Zander-Nowicka,J. 2006. *Model transformers for test generation from system models*. Technical report, Fraunhofer FOKUS,Germany and Hitachi Central Research Laboratory Ltd., Japan.

[6] Ciccozzi, F., Cicchetti, A., Siljamäki, T, Kavadiya, J. 2010. Automating test cases generation: from xtuml system models to qml test models. In *Proceedings of International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pp. 9–16.

[7] Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., Weyuker, E.J. 2004. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability,* 14(1), 17-44.

[8] Dai, Z.R. 2004. Model-driven testing with UML 2.0. In *Proceedings of the Second European Workshop on Model Driven Architecture*, pp. 179-187.

[9] de la Riva, C., Suárez-Cabal, M.J., Tuya, J. 2010. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th International Workshop on Automation of Software Test*, pp. 67-74.

[10] Emmi, M., Majumdar, R., Sen, K. 2007. Dynamic Test input generation of database applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 151-162.

[11] Fujiwara, S., Munakata, K., Maeda, Y., Katayama, A., Uehara, T. 2011. Test data generation for web application using a UML class diagram with OCL constraints. *Innovations in Systems and Software Engineering*, 7(4), 275-282.

[12] Gardikiotis, S.K., Malevris, N. 2009. A Two-folded Impact Analysis of Schema Changes on Database Applications. *International Journal of Automation and Computing*, 6(2) 109-123.

[13] Garg, D., Datta A. 2012. Test Case Priorization due to Database Changes in Web Applications. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pp. 726-730.

[14] Halfond, W.G.J., Orso, A. 2006. Command-form coverage for testing database applications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp. 69-80.

[15] Kapfhammer, G.M., Soffa, M.L. 2003. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering*, pp. 98-107.

[16] Kapfhammer, G.M., Soffa M.L. 2008. Database-aware test coverage monitoring. In *Proceedings of the 1st India Software Engineering Conference*, pp. 77-86

[17] Kapfhammer, G.M., McMinn, P., Wright, C.J. 2013. Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, pp. 31-40.

[18] Khalek, S.A., Elkarablieh, B., Laleye, Y.O., Khurshid, A. 2008. Query-aware test Generation using a relational constraint solver. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 238-247.

[19] Lamancha, B.P., Reales, P., García, I., M. Polo, Piattini, M. 2009. Automated Model-based Testing using the UML Testing Profile and QVT. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pp. 1-10.

[20] Lamancha, B.P, Reales P., Polo M., Caivano, D. 2011. Model-driven testing: transformations from test models to test code. In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 121-130.

[21] Leavitt, N. 2010. Will NoSQL databases live up to their promise? *IEEE Computer*, 43(2) 12-14.

[22] Liu, Y., Li, Y., Wang, P. 2010. Design and implementation of automatic generation of test cases based on model driven architecture. In *Proceedings of the 2nd International Conference on Information Technology and Computer Science*, pp. 344-347.

[23] Lo, E., Binnig, C., Kossmann, D., Özsu, M.T., Hon, W.K. 2010. A framework for testing DBMS features. *The VLDB Journal*, 19(2), pp. 203-230.

[24] Marcozzi, M., Vanhoof, W., Hainaut, J.L. 2013. A relational symbolic execution algorithm for constraint-based testing of database programs. In *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation*, pp. 179-188.

[25] McMinn, P., Wright, C.J., Kapfhammer, G.M. 2015. An Analysis of the Effectiveness of Different Coverage Criteria for Testing Relational Database Schema Integrity Constraints. Technical Report CS-15-01, University of Sheffield.

[26] Moniruzzaman, A.B.M., Hossain, S.H. 2013. NoSQL Database: New Era of Databases for Big data Analytics-Classification, Characteristics and Comparison. *International Journal of Database Theory and Application*, 6(4) 1-14.

[27] Neo4J, http://neo4j.com

[28] NoSQL databases, http://nosql-database.org

[29] Robinson, I., Webber, J., Eifrem, E. 2013. *Graph databases*. O'Reilly.

[30] Schieferdecker, I. 2012. Model-based testing. *IEEE Software* 29, 14-18.

[31] Tuya, J., Suárez-Cabal M.J., de la Riva, C. 2006. SQLMutation: a tool to generate mutants of SQL database queries. In *Proceedings of the Second Workshop on Mutation Analysis*.

[32] Tuya, T., Suárez-Cabal, M.J., de la Riva, C.2007. Mutating database queries. *Information and Software Technology*, 49(4) 398-417.

[33] Tuya, T., Suárez-Cabal, M.J., de la Riva, C. 2010. Full predicate coverage for testing SQL database queries. *Software Testing Verification and Reliability*, 20(3) 237-288.

[34] Utting, M., Pretschner, A., Legeard, B. 2012. A taxonomy of model-based testing approach. *Software Testing, Verification and Reliability*, 22(5) 297-312.

[35] Willmor, D., Embury, S.M. 2005. Exploring test adequacy for database systems. In *Proceedings of the 3rd UK Software Testing Research Group*, pp. 123-133.

[36] Willmor, D., Embury, S.M. 2006. Testing the implementation of business rules using intensional database

tests. In *Proceedings of Testing: Academic & Industrial Conference on Practice and Research Techniques*, pp. 115-126.

[37] Wright, C.J., Kapfhammer, G.M., McMinn, P. 2013. Efficient Mutation Analysis Of Relational Database Structure Using Mutant Schemata And Parallelisation. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops*, pp. 63-72.

[38] Zander, J., Dai, Z.R., Schieferdecker, I., Din, G. 2005. From U2TP models to executable tests with TTCN-3 - an approach to model driven testing. In *Testing of Communicating Systems*. LNCS 3502, pp.289-303.

[39] Zhou, C., Frankl, P. 2011. JDAMA: Java Database Application Mutation Analyzer. *Software Testing, Verification and Reliability*, 21(3), 241-263.

EvoSE: Evolutionary Symbolic Execution

Mauro Baluda
Fraunhofer SIT
Darmstadt, Germany
mauro.baluda@sit.fraunhofer.de

ABSTRACT

Search Based Software Testing (SBST) and Symbolic Execution (SE) have emerged as the most effective among the fully automated test input generation techniques. However, none of the two techniques satisfactorily solves the problem of generating test cases that exercise specific code elements, as it is required for example in security vulnerability testing.

This paper proposes EvoSE, an approach that combines the strengths of SBST and SE. EvoSE implements an evolutionary algorithm that searches the program control flow graph for symbolic paths that traverse the minimum number of unsatisfiable branch conditions. Preliminary evaluation shows that EvoSE outperforms state-of-the-art SE search strategies when targeting specific code elements.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Verification

Keywords

Test automation, search-based software testing, symbolic execution

1. INTRODUCTION

Test automation has the potential to drastically reduce the cost of quality assurance in software development. With this motivation, a great amount of research has been devoted in particular to the problem of generating test input values that thoroughly exercise a software system. Search Based Software Testing (SBST) and Symbolic Execution (SE) are two of the most popular approaches to generate test suites automatically [1].

The problem of finding an input that exercise a specific code element and the dual problem of proving that a code element is infeasible are well known undecidable problems. To avoid wasting resources towards infeasible goals, most recent SBST and SE approaches aim to maximize code coverage measures instead of focusing on specific code elements [4, 7]. EvoSE instead aims to generate test cases targeting specific code elements as this is required, for example, in security testing where one wishes to exploit a presumable vulnerability or test the validity of an assertion.

A number of goal oriented SE search strategies have been proposed to address the problems of testing software patches, improve code coverage and exploiting security vulnerabilities [12, 13, 14]. These approaches employ ad hoc heuristic to rank the program branches that should be expanded first in the next SE iterations with the goal of reaching quickly the target code elements. EvoSE instead implements a metaheuristic exploration of the program execution space thus overcoming the well known limitations of deterministic best-first search algorithms.

EvoSE follows the recent line of work that combines SBST and SE approaches to benefit from their complementary strengths and weaknesses [17, 3, 8, 5]. However, instead of searching in the numeric space of program inputs, EvoSE considers the combinatorial space of the program execution paths. Metaheuristic algorithms have been successfully applied in the context of combinatorial problems like the traveling salesman problem and software model checking [11, 9]. To the best of our knowledge, EvoSE is the first approach that investigates the use of evolutionary algorithms to guide the exploration of symbolic execution paths.

2. THE EVOSE APPROACH

EvoSE is a novel test generation approach that combines SBST and SE aiming to exercise specific program elements. The main departure from existing SBST approaches is the identification of a different search space. While classic SBST techniques perform a search in the input space of a program, EvoSE considers the space of the program execution paths that may lead to the target program element.

Classic SBST techniques try to minimize a fitness function that measures the distance between the concrete execution and the target code element, this is because most of the program inputs produce executions that do not reach the target code element. Instead EvoSE considers the program execution paths that reach the target code element in the program Control Flow Graph (CFG). The CFG is an approximation of the program behavior and also includes infeasible paths, that is, paths that cannot be executed under any program

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804325

input. In essence, *EvoSE implements a search in the CFG for feasible program paths* that reach the target code element.

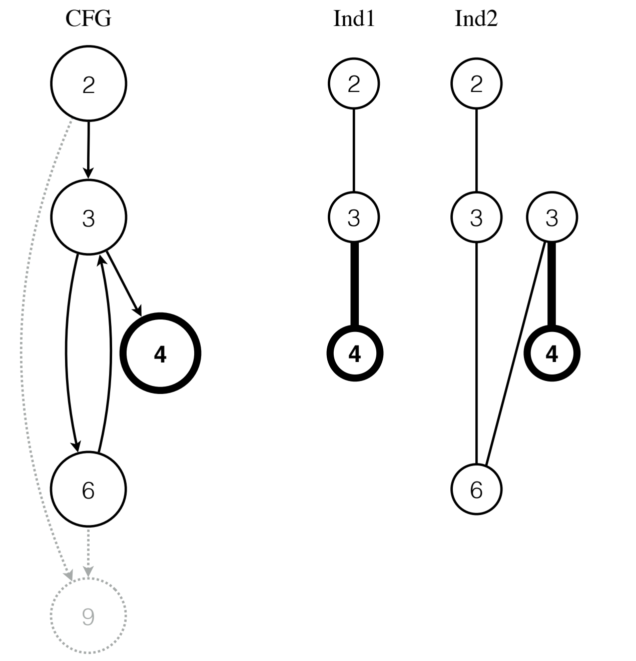
Evolutionary algorithms come in different forms, we designed EvoSE as a *memetic algorithm*, a metaheuristic algorithm that combines a classic Genetic Algorithm (GA) with systematic local optimization. In the following we describe the EvoSE design and in particular we focus on the problem representation, the fitness function, the crossover and mutation operators, and finally on the systematic local search algorithm. The description as well as the current implementation are limited to the intraprocedural case.

2.1 Problem Representation

EvoSE searches for feasible paths in the program CFG. The individuals that compose the evolving population are represented by variable-length lists of branches that are contiguous in the CFG graph. Each list begins with the program entry point and ends with the target program element. The initial population of candidate solutions is seeded performing random walks in the CFG.

```
1:  int s=read();
2:  for(int i=0; i < s; i++) {
3:     if(i > 100) {
4:        exit(ERROR);
5:     } else {
6:        ...
7:     }
8:  }
9:  exit();
```

(a) Example program



(b) CFG of the program in Figure 1a and two individuals of the GA population Ind1 and Ind2

PC1: `i=0 && i<s &&` **`i>100`**
PC2: `i=0 && i<s && i<=100 && i+1<s &&` **`i+1>100`**

(c) Path conditions of the GA individuals in Figure 1b

Figure 1: Example EvoSE problem

As an example, consider the code in Figure 1a and the corresponding CFG in Figure 1b. The CFG nodes are labeled by the corresponding line numbers in the code, dashed lines represents CFG portions that can be ignored as they cannot be part of a path reaching the target node 4. In this program, the only feasible path that reaches the target is one that enters the loop 100 times. The two infeasible paths encoded by the individuals Ind1 and Ind2, constitute the initial population of the GA, they enter the loop one time and two times respectively.

2.2 Fitness Function

It is well known that metaheuristic search techniques perform better when the search landscape is smooth, that is, when elements that are nearby in the search space have similar fitness values [6]. For this reason, state of the art SBST techniques employ fitness functions that combine two different metrics: *approach level* and *branch distance*. The *approach level* is a discrete distance value that is computed by counting the nodes that separate the exercised path and the target element in the program CFG. The *branch distance* is a smooth measure that, considering the branch with the smallest *approach level*, tells how far the execution went from taking the opposite side of the conditional (the interested reader can refer to [15] for more details). Similarly, the fitness function adopted in EvoSE is not a binary value that tells if the considered execution path is feasible or not, but is instead a continuous value obtained by performing a Dynamic Symbolic Execution (DSE) of the program along that path.

Classic SE allows to map a certain program execution path to a program input, if it exists, that produces an execution following the same path. This can be achieved automatically by solving the conditions that appear on the path with the help of an SMT solver. EvoSE uses a MaxSMT solver and can therefore obtain a smoother measure of how far the path is from being feasible. Given a list of constraints, MaxSMT solvers identify the largest subset of elementary constraints that is satisfiable. The fitness function is computed dynamically by executing the program along the desired path using as input the solution provided by the MaxSMT.

EvoSE guarantees that the dynamic symbolic execution follows the desired path by forcing the concrete evaluation of branch conditions in the style of execution hijacking [16]. The symbolic evaluation of branch conditions instead is computed as usual. EvoSE defines a *graded feasibility* measure by counting the number of branches that are not in the subset identified by MaxSMT, this measure is used to direct the search towards maximally satisfiable paths and eventually identify feasible execution paths. *Graded feasibility* can be regarded as a replacement for the *approach level* metric. EvoSE computes the classic *branch distance* measure for each unsatisfiable branch along the execution and normalizes it according to the formula proposed by Arcuri in [2].

In summary, given an execution path p, the fitness value $f(p)$ that needs to be minimized is the sum of the path's graded feasibility $grad\_feas(p)$ and the normalized branch distance $b\_dist(p, b_i)$ for all the branch conditions b_i that are not satisfied by the MaxSMT solution $maxsmt(p)$ to the symbolic constraints collected for path p:

$$f(p) = grad\_feas(p) + \sum\nolimits_{b_i \notin maxsmt(p)} norm(b\_dist(p, b_i)) \tag{1}$$

Consider the GA individuals Ind1 and Ind2 in Figure 1b. Both their respective path conditions PC1 and PC2 reported in Figure 1b are infeasible and their MaxSMT solutions contains all but the last branch condition (represented with a bold line). Both the individuals have therefore *graded feasibility* 1. The *branch distance* however is smalled for P2 because the value of i is 0 and thus the expression $i + 1$ is closer to 100 compared to i. This matches our intuition that an execution that gets closer to executing the loop 100 times should be favored in the population evolution.

The fitness function used in EvoSE, while being closely related to the classic branch distance, is computed for each unsatisfied condition along the path and not only for the branch that is closest to the target code element. For this reason, the genetic algorithm favors individuals that contain a large number of jointly-satisfiable branch conditions, independently of their position in the path. In light of Goldberg's *building block hypothesis*, we suggest that sequences of jointly-satisfiable branch conditions constitute the fundamental building blocks of the optimal, fully satisfiable, execution path.

2.3 Crossover Operator

Cut and splice is a typical choice as a one-point crossover operator for individuals of variable length like the paths in a graph. Given two individuals of the form `A|B` (list A followed by list B) and `C|D`, the new offsprings will have the form `A|D` and `C|B` respectively.

To guarantee that the newly generated offsprings encode existing paths in the CFG, EvoSE combines *cut and splice* with a repair strategy that uses the CFG to connect part `A` of the first individual with the longest possible postfix of part `D` of the second individual. In the example from Figure 1, the crossover operator could try to join the prefix [2,3] of Ind1 with the postfix [3,6,3,4] of Ind2. Such crossover produces the path [2,3,3,6,3,4] which is not a valid path and needs to be repaired.

The repair strategy would use the CFG to reconnect the two sub-paths and would produce the individual Ind3 that encodes the path [2,3,6,3,6,3,4]. If connecting the last branch in the prefix of Ind1 to the first branch in the postfix of Ind2 is not possible, the repair strategy would consider the next branch in the postfix of Ind2. Eventually the process will produce a valid individual as Ind1 and Ind2 share at least the last branch, that is, the target code element.

2.4 Mutation Operator

The EvoSE mutation operator selects randomly a branch in the individual and replaces it with the paired branch in the CFG. This produces an individual that encodes an execution path in which one branch condition evaluates differently respect to the original individual. Consider Ind3 from the previews paragraph, the mutation operator might decide for example to replace branch 4 with the paired branch 6 (the execution of branch 4 or 6 being the two possible outcomes of the evaluation of the condition at line 2).

As for the case of the crossover operator in Section 2.3, the individual obtained after the mutation might not encode a valid CFG path. This is the case in our example where the mutated path would end on branch 6 and not on the target branch 4. The repair strategy described earlier is finally used to reconnect branch 6 to branch 4 producing the valid individual Ind4 that encodes the path [2,3,6,3,6,3,6,3,4].

2.5 Local Search

GA can be very effective in finding solutions that approximate the global optimum but might miss some local optimization opportunity due to their stochastic nature. To overcome this limitation, *memetic algorithms* combine GA with deterministic local optimization [10].

For a given notion of neighborhood, a local search is performed by systematically evaluating all the neighbors of a candidate solution, retaining the individual with the best fitness. The process is repeated until no further improvement is possible. In our problem representation we could identify as neighbor of a given individual, any individual produced by flipping one of the conditions along the path, that is, any individual that can be obtained by a single application of the mutation operator defined in Section 2.4. This definition however produces a very large neighborhood that is impractical for a deterministic local search.

In EvoSE we included a local search strategy that visits all the neighbors that can be obtained by replacing one of the infeasible branches along the individual's execution path. We called this simple strategy *regret minimization* and we applied it as a fourth genetic operator operator after selection, crossover and mutation. Our preliminary evaluation showed that regret minimization is effective in improving the optimization process and does not affect negatively the EvoSE performance.

Consider again the individual Ind1 from Figure 1b. The only unsatisfied condition in the encoded path is the last one: `i>100`. Flipping this condition produces the individual Ind2 that has a better fitness then Ind1 as the only unsatisfied condition (i.e. `i+1>100`) leads to a smaller *branch distance.*

3. PRELIMINARY EVALUATION

We implemented EvoSE with the help of the open source evolutionary algorithms framework DEAP [1] and the symbolic execution engine CREST [2]. We implemented the evolutionary operators described in Section 2 using the DEAP infrastructure . We modified CREST to symbolically execute (possibly) infeasible paths using *execution hijacking*, integrate the MaxSMT solver Yices [3], and compute the fitness function from Equation 1.

```
extern char *curr_ptr;
GetKeyword(char *kw) {
  char word[KWDSLEN+1];
  char ch=getchar(curr_ptr);
  int i=0;
  while((isalnum(ch)||ch=='_') && i<KWDSLEN) {
    word[i++]=ch;
    ch = getchar(curr_ptr);
  }
  word[i]=0;
  if(strcmp(kw, word) == 0) {
    printf("TARGET!");
  }
}
```

Figure 2: A two-pass text parser.

[1] https://deap.readthedocs.org

[2] https://code.google.com/p/crest

[3] http://yices.csl.sri.com

We evaluated EvoSE on the function GetKeyword that implements a two-pass text parser. A simplified version of the parser code is in Figure 2. The function execution reaches the target branch at line 12 when the keyword pointed by the variable kw is found on the input buffer pointed by curr_ptr. The loop at lines 6—9 filters out the input characters that are not alphanumeric.

From our experience, it is very hard for classic SE to generate an input buffer that reaches line 12. This is because the decision at line 11 depends from which branch was traversed earlier at line 6. Only characters that are alphanumeric in fact, can match the alphanumeric keyword kw, this relation however cannot be observed by an analysis that explores paths independently. In EvoSE, paths that correspond to inputs with many alphanumeric characters will produce smaller fitness values due to the small branch distance at line 11 and will therefore be favored in the evolutionary process.

We compared EvoSE against the different SE search strategies implemented by CREST. We used keywords kw of variable lengths and limited the execution time to 60 minutes for each of the experiments. We found that EvoSE could discover keywords of length up to 50 characters while CREST could only identify keywords up to 4 characters long. The analysis of the results revealed that EvoSE needed to generate paths over 1000 branches long to reach the target line, such depth analysis of the symbolic execution space is normally considered out of reach. These first empirical results indicate that evolutionary algorithms can be effective in directing the exploration of SE paths towards target program elements.

4. CONCLUSION

EvoSE is a novel, goal-oriented test generation technique that combines SBST and SE. The core of the technique is a metaheuristic search strategy to efficiently explore program executions paths. Preliminary experimental results indicate that EvoSE can be more effective then classic SE strategies in producing test inputs that exercise specific code elements.

Future research directions include the extension of the genetic operators to the interprocedural case, the investigation of alternative population seeding strategies and a thorough evaluation of the EvoSE effectiveness, in particular for problems arising in the context of security testing.

5. REFERENCES

[1] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. of Systems and Software*, 86(8):1978–2001, 2013.

[2] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[3] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 53–62, Nov 2011.

[4] M. Baluda, P. Braione, G. Denaro, and M. Pezzè. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Journal*, 19(4):725–751, 2011.

[5] P. Dinges and G. Agha. Solving complex path conditions through heuristic search on induced polytopes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 425–436, New York, NY, USA, 2014. ACM.

[6] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[7] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276 –291, feb. 2013.

[8] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *IEEE International Symposium on Software Reliability Engineering*, 2013.

[9] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 266–280. Springer Berlin Heidelberg, 2002.

[10] H. Ishibuchi, T. Yoshida, and T. Murata. Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling. *Evolutionary Computation, IEEE Transactions on*, 7(2):204–223, April 2003.

[11] P. Larrañaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[12] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] P. D. Marinescu and C. Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, New York, NY, USA, 2013. ACM.

[14] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. *SIGPLAN Not.*, 50(4):473–486, Mar. 2015.

[15] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, ASE '98, Washington, DC, USA, 1998. IEEE Computer Society.

[16] P. Tsankov, W. Jin, A. Orso, and S. Sinha. Execution hijacking: Improving dynamic analysis by flying off course. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 200–209, Washington, DC, USA, 2011. IEEE Computer Society.

[17] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, pages 359–368, June-July 2009.

Testing Data Transformations in MapReduce Programs

Jesús Morán
University of Oviedo
Computer Science Department
Campus de Viesques, Gijón, Spain
(+34) 985 18 2153
moranjesus@lsi.uniovi.es

Claudio de la Riva
University of Oviedo
Computer Science Department
Campus de Viesques, Gijón, Spain
(+34) 985 18 2664
claudio@uniovi.es

Javier Tuya
University of Oviedo
Computer Science Department
Campus de Viesques, Gijón, Spain
(+34) 985 18 2049
tuya@uniovi.es

ABSTRACT

MapReduce is a parallel data processing paradigm oriented to process large volumes of information in data-intensive applications, such as Big Data environments. A characteristic of these applications is that they can have different data sources and data formats. For these reasons, the inputs could contain some poor quality data that could produce a failure if the program functionality does not handle properly the variety of input data. The output of these programs is obtained from a number of input transformations that represent the program logic. This paper proposes the testing technique called MRFlow that is based on data flow test criteria and oriented to transformations analysis between the input and the output in order to detect defects in MapReduce programs. MRFlow is applied over some MapReduce programs and detects several defects.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *Validation*

General Terms

Reliability, Verification.

Keywords

Software Testing, Data Flow Testing, MapReduce programs.

1. INTRODUCTION

The *MapReduce* paradigm [11] is based on the "divide and conquer" principle, which is the breaking down (*Map*) of a large problem into several sub-problems (*Reduce*). *MapReduce* is used in *Big Data* and *Cloud Computing* to process large data. The unit of program information is a <*key, value*> pair, where the *value* has data relative to the sub-problem identified by the *key*. The program output is the result of a series of transformations about the input information stored in the <*key, value*> pairs.

The quality in *MapReduce* programs is important due to their use in critical sectors, like health (ADN alignment [27]) or security (image processing in ballistics [17]). Software testing is one of the industrial practices most used to ensure quality. In recent years testing technique research has advanced [6], but few efforts have been focused on massive data processing like *MapReduce* [8]. These paradigms have new challenges in the field of testing [23][21][29], and some authors [15][26] estimate respectively that 3% and 1.38%-33.11% of *MapReduce* programs do not finish. Another *MapReduce* issue is that in some scenarios the developers create several subprograms with a few transformations instead of creating one program [26]. In these scenarios, the subprograms take more resources and underperform in comparison with a whole program.

On the other hand, a study about the *MapReduce* field has discovered that 84.5% of faults are due to data processing [19]. In order to detect these defects, this paper proposes a testing technique that analyzes the program transformations which could produce the failures. The testing technique named *MRFlow* (MapReduce data Flow) is based on *data flow* test criteria [25]. The program functionality is represented by means of program transformations, and then the test cases are derived from these transformations in order to test the functionality. Firstly, a program graph is elaborated with information about the program transformations, then the paths under test are extracted representing the transformations, and finally each path under test is tested with different data (empty, not empty, valid, non-valid, with emission of result and without emission of result). The main contributions of this paper are (1) a testing technique specifically tailored to test *MapReduce* programs in order to detect defects, and (2) the application over two popular case studies.

The rest of the paper is organized as follows: the *MapReduce* paradigm, *data flow* test criteria and the related work are summarized in Section 2. Next, Section 3 describes the *MRFlow* testing technique, the elaboration of the graph in Subsection 3.1 and the derivation of test cases in Subsection 3.2. In Section 4 *MRFlow* is applied to two programs and reveals some defects. Finally, Section 5 contains the conclusions.

2. BACKGROUND

The *MRFlow* testing technique is based on *data flow* criteria that analyze the evolution of variables in *MapReduce* programs. In Subsection 2.1 the *MapReduce* paradigm is summarized, *data flow* test criteria basis is in Subsection 2.2, and the related work is described in Subsection 2.3.

2.1 MapReduce

The *MapReduce* paradigm solves a problem by splitting it into sub-problems that can run in parallel. Fundamentally, *MapReduce* has two functions: *Map* that splits the problem into sub-problems, and *Reduce* which solves each sub-problem. Both functions handle <*key, value*> pairs, where *key* is the identifier of each sub-

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804326

problem and the *value* corresponds to some data relative to that sub-problem. The *Map* function receives the data input and emits a *<key, value>* pair, then the *Reduce* function receives *<key, list(values)>* pairs that contain all the information about each sub-problem, and finally solves it with a *<key, value>* pairs.

Consider as an example a program that counts the number of occurrences of each word in a text. This problem is divided into as many sub-problems as there are different words, then each sub-problem only counts the occurrences of one word and the *key* is that word. The goal of the program is to count, so the *value* should contain information relative to the counting of the word, then the *value* contains a number of occurrences. For example, if the input texts are "hi Hadoop" and "hi", the *Map* function emits *<hi, 1>*, *<Hadoop, 1>* and *<hi, 1>*. Then there are two sub-problems, so the *Reduce* function receives *<Hadoop, 1>* and *<hi, [1,1]>* and emits *<Hadoop, 1>*, *<hi, 2>* which is the number of occurrences of each word in the texts.

The *MapReduce* programs are often used in *Big Data* programs [28], which process large data (Volume), with a necessary performance (Velocity) and with different types of data, data from different sources, and data without apparently a data model such as for example emails or videos (Variety). To handle this data a parallel and fault tolerant infrastructure is necessary, for this reason typically the *MapReduce* programs run over frameworks, excelling *Hadoop* [1] due to its impact on corporations [2].

2.2 Data Flow Test Criteria

The goal of *data flow* test criteria is to derive tests through the analysis of program variables. Several testing techniques are based on *data flow*, for example to test web applications through the analysis of state variables [5]. *Data flow* is a structure testing technique [4] created from the program *P*. A control flow graph *G(P)* is created from the program, where the edges represent each statement, and the vertices indicate the following possible statements. In addition to the graph, the definition and uses of every variable are determined [25]. In a node $n \in N$, when a *value* is assigned to the variable $v \in V$, the variable *v* is defined and the representation is *DEF(v,n)*. If the variable *v* is in a predicate of a condition (i.e., if (v)), then the representation is *P-USE(v,n)*, and in other uses of *v* the representation is *C-USE(v,n)*. For example, in the statement a = b+1, *a* is defined and *b* is used.

2.3 Related Work

Several testing approaches exist over the *MapReduce* programs, but most of them are focused on testing the performance [16][14][9] and few are oriented to testing the functionality, that is the goal of this paper. A classification of testing in *Big Data* is proposed by Gudipati et al. [13]. On this point Camargo et al. [7] and Morán et al. [22] elaborate a classification of defects, and Csallner et al. [10] test one defect automatically based on a symbolic execution framework. Another defect can be detected in compilation time by Dörre et al. [12]. In order to create test inputs, Mattos [20] develops a bacteriological algorithm supported by a function created by the tester, and Li et al. [18] design a test framework which validates the large database procedures. Our paper is different from other studies in the sense that it obtains the test cases from the program transformations systematically.

3. MRFLOW TESTING TECHNIQUE

The *MapReduce* program logic is represented by the transformations of *keys* and *values* into the program output. In these transformations, the *keys* and *values* can be transformed into one variable, this variable can be transformed into another, and so on until the final output.

Usual *data flow* test criteria like "*all-du-paths*" analyzes the definitions and uses of each variable, but does not consider the transformations between variables in enough degree of detail. In this sense, the testing technique proposed (*MRFlow*) analyzes the transformations from *keys* and *values*. This paper focuses on the *Reduce* function because it has a large part of the program functionality, but it can also be applied over the *Map* function because both handle *key* and *values*. Subsection 3.1 describes the elaboration of the graph, and the derivation of the test is detailed in Subsection 3.2.

3.1 Elaboration of MRFlow Graph

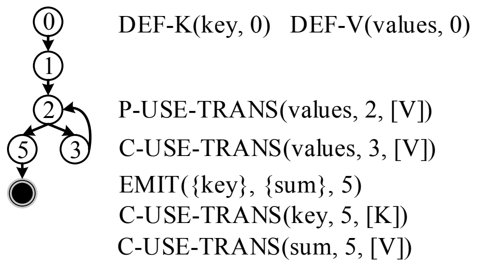
In the *MRFlow* graph, the statements of the program are in the nodes and each edge represents the next potential statement. In this graph, as described below, each node also contains information about the uses of variables coming from transformations, definition of *key*/*values*, and the output.

USE nodes: It contains only the use of a variable *var* coming from a *key*/*value* transformation. A transformation occurs when a variable is formed by information coming from *key*, part of *key*, all/part of *values*, a unique *value* or combinations of the above. A sequence of these elements of *keys* and *values* is labeled in the node and represents a transformation between the input *key*/*values* variable and another variable.

Given a variable *var*, a statement *n* and a transformation *seq*, *P-USE-TRANS(var, n, seq)* is defined when variable *var* is used in the conditional statement *n* and comes from a transformation *seq*; and *C-USE-TRANS(var, n, seq)* when *var* is used in a non-conditional statement. The *seq* label contains the transformation of *var* in a sequence of *key*/*values* with conjunction $\wedge$ and disjunction $\vee$ connectors. The conjunction connector indicates that a transformation exists with both elements of the sequence, and the disjunction connector indicates that several transformations exist, one for each part of the sequence. For example, *P-USE-TRANS(var, 6, (key* $\wedge$ *value)* $\vee$ *key)* means that the variable *var* is used in the conditional statement 6 with two possible transformations, one is formed by the *key* and *value*, and the other only by the *key*. Because the transformation can be formed by parts of *key*/*values*, the *seq* sequence uses the following expressions:

- *Key* transformations:
  - [K]: Transformation over the whole *key*. For example: var = key, or var = key.length().
  - Ki: Transformation over the part *i* of *key*. Sometimes the *key* is composed of several elements. For example if the program should obtain the counting of every word in every year, the *key* is the compound of word and year. A transformation that involves the *key* part "word" (Kword) could be: var = getWord(key).

```
0  Reduce (Key key, List values){
1    sum = 0;
2    while (values.hasNext()){
3       sum += values.next();
4    }
5    emit(key, sum);
6  }
```



0
1
2
5
3
DEF-K(key, 0) DEF-V(values, 0)
P-USE-TRANS(values, 2, [V])
C-USE-TRANS(values, 3, [V])
EMIT({key}, {sum}, 5)
C-USE-TRANS(key, 5, [K])
C-USE-TRANS(sum, 5, [V])

Figure 1. MRFlow graph of WordCount program.

- *Values* transformations:
  - [V]: Transformation over several *values*. For example: var = values[0] + values[1].
  - V: Transformation over one *value*. For example: var = values.next().
- *Values* transformations with *categories*: The *Reduce* function could receive several *values* of a different nature and handle them in a different way. Such different *values* are considered a *category* and could come from different *Map* functions, a different data source or contain very different information. For example, a SPAM detector that receives several types of messages as a *values* (sms and email) has two categories: V:sms and V:email. The character of the sms and email, and the processing in the program is very different, so there are two categories.
  - [V:cat]: Transformation over several *values* of *cat category*. For example, the statement var = values[0] + values[1] could be a [V] transformation, but if values[0] and values[1] are from *category* sms, then the transformation is [V:sms].
  - V:cat: Transformation over one *value* of *cat category*. For example: if(isSms(values[0])) var = values[0].

DEF nodes: It contains the assignation of new content in the input *key* or in the list of *values*. Given a variable *var* and a statement *n*, *DEF-K(var, n)* is defined when new content is assigned to the variable *var* in the statement *n*, and *var* is the input *key* variable. *DEF-V(var, n)* is defined when *var* is the input *list(values)* variable.

Emit Nodes: The *Reduce* output is emitted by a special statement in <*key, value*> pairs. Given the variables $\{k_1,k_2,...k_m\}$, the variables $\{v_1,v_2,...,v_p\}$ and a statement *n*, *EMIT(*$\{k_1,k_2,...k_m\}$*,* $\{v_1,v_2,...,v_p\}$*, n)* is defined when the *n* emits a <*key, value*> pair, the *key* is created by the variables $\{k_1,k_2,...k_m\}$, and the *value* by $\{v_1,v_2,...,v_p\}$.

As an example consider the *Reduce* function of *Wordcount* program [3] that counts the occurrences of each word. Figure 1 illustrates the *MRFlow* graph. The *Reduce* function receives a word as a key, and a list of numbers of occurrence as values, for instance <*hello, [1,1,1]*> means that the word "hello" has 3 occurrences in the text. In this program, the variables *key*, *values* and *sum* come from a transformation of *key*/*values* input variables. If the statement 3 is reached the *values* variable is transformed into *sum* by the addition of all *values* [V], but in other cases *values* is not transformed. The graph contains in node 0 the definition of *key* and *values*. The node 1 is empty because the *sum* variable is not created from *key*/*values* at this point. The node 2 contains a conditional statement of *values* variable. In node 3 there is a transformation of *values* in *sum*, and finally in node 5 the output, which contains *key* and *sum*, is emitted. The program does not combine *key* and *values* in any variable, and each *value* only represents the number of occurrences, so the program has neither categories nor connectors in the sequence of transformation (*seq* label).

3.2 Derivation of Test Cases

The goal of *MRFlow* is to derive tests in order to analyze the different *key*/*value* transformations with or without categories. In *MRFlow* graph, the paths under test start in definition of *key*/*value* and finish in each possible last transformation of such variables. Unlike *data flow* test criteria where each path is covered by a test case, in *MRFlow* for each path under test several situations to be covered (test coverage items) are defined and represent the transformations which are the goal of the test cases. Then the test cases are designed to cover the test coverage items in the path under test.

Transformation paths (*tp*): The paths under test, called transformation paths (*tp*), are extracted from transformations between input and output in *MRFlow* graph. One *tp* is created between each *DEF-K*/*DEF-V* node and *C-USE-TRANS*/*P-USE-TRANS* of each last transformation of *key* or *list(values)*. In the case of *DEF-K/DEF-V* to *P-USE-TRANS(var, n, seq)*, instead of creating one *tp*, several *tp* are created following all of the next nodes after the conditional statement *n*, as in other *data flow* test criteria [25]. For example, the transformations and *tp* of *WordCount* [3] program are represented in Figure 2. The program has 5 *tp* obtained from the transformation between *values* and *sum* (*tp*$_1$), the non-existence of *values* transformations (*tp*$_2$, *tp*$_3$ and *tp*$_4$) and the non-existence of *key* transformations (*tp*$_5$). The *values* variable is defined in node 0 and the last transformations are *sum* and *values* depending on whether statement 3 is reached or not. The sequence of transformation (*seq* label) between *values* and *sum* is [V] because it involves all values. In the case of *key* there is no transformation, so *key* is the last transformation. Finally, the transformation paths are obtained between *DEF-K/DEF-V* and *C-USE-TRANS/P-USE-TRANS* of last transformations. In the case of *P-USE-TRANS* like *P-USE-TRANS(values, 2, [V])*, one *tp* is created following the next nodes after node 2, that is node 3 (*tp*$_2$) and node 5 (*tp*$_3$).

Test coverage items: Each *tp* represents the transformations and the uses of transformation variables. Depending on the type of transformation (*key*, part of *key*, *values*, *value* or combination)

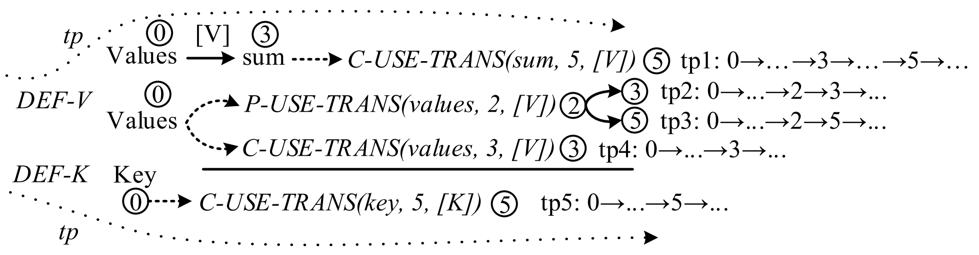

tp
⓪ [V] ③
Values → sum → C-USE-TRANS(sum, 5, [V]) ⑤ tp1: 0→…→3→…→5→…
DEF-V
⓪
Values
P-USE-TRANS(values, 2, [V]) ② ③ tp2: 0→...→2→3→...
⑤ tp3: 0→...→2→5→...
C-USE-TRANS(values, 3, [V]) ③ tp4: 0→...→3→...
DEF-K Key
⓪ C-USE-TRANS(key, 5, [K]) ⑤ tp5: 0→...→5→...
tp

Figure 2. Example of transformation paths (tp) in WordCount program.

Table 1. Summary of program features and test results

	WordCount (Reduce)	IPContry (Reduce)
Number of transformations	3 (*Key*:1, *Values*:2)	3 (*Key*:1, *Values*:2)
DEF-K/DEF-V nodes	2 (*Key*:1, *Values*:1)	2 (*Key*:1, *Values*:1)
C-USE-TRANS nodes	3 (*Key*:1, *Values*:2)	5 (*Key*:2, *Values*:3)
P-USE-TRANS nodes	1 (*Key*:0, *Values*:1)	1 (*Key*:0, *Values*:1)
EMIT nodes	1	2
Transformation paths (*tp*)	5 (*Key*:1, *Values*:4)	6 (*Key*:2, *Values*:4)
Test coverage items	30 (*Key*:6, *Values*: 24)	30 (*Key*:6, *Values*:24)
Number of test cases	2	2
Test coverage items covered	16 (*Key*:4, *Values*:12)	16 (*Key*:4, *Values*:12)
Test coverage items not covered	14 (*Key*:2, *Values*:12)	14 (*Key*:2, *Values*:12)

several situations have to be tested. These situations (test coverage items) are usual in these types of programs and for each *tp* are defined next:

- Existence of information: *tp* created with empty data or non-empty data. Depending on the type of transformation (*seq* label in *MRFlow* graph) can occur:
  - If *tp* contains [V]: for each *category cat*, the transformation is created with *cat* data, or without *cat* data.
  - If *tp* contains [K]: the transformation is created with data in all *key*, or with empty data for each part of *key*.
- Validation: *tp* created with valid data or non-valid data. Depending on the type of transformation (*seq* label in *MRFlow* graph) can occur:
  - If *tp* contains [V]: for each *category cat*, the transformation is created with valid *cat* data, or non-valid *cat* data.
  - If *tp* contains [K]: the transformation is created with valid data in all *key*, or with non-valid data for each part of *key*.
- Output: *tp* reaches *EMIT* node or not.

Consider the *Reduce* function in the *WordCount* example (Figure 1). The test cases are designed in order to cover the test coverage items in each *tp*. For example, the test coverage items in all *tp*: "transformation with non-empty data", "with valid data" and "with output emission", can be covered by a test case with *Reduce* input *<hi, [1,1]>* which means that the word "hi" is repeated twice. In order to cover the other test coverage items (transformation with non-valid *key*, with empty values, and so on), new test cases have to be created, but it is possible that some test coverage items cannot be covered, as for example "Transformation without output emission" in all *tp* of *WordCount* because the *EMIT* node is always reached.

4. CASE STUDIES

In order to explore the applicability of the testing technique, *MRFlow* is applied over two popular programs: *WordCount* [3] which counts the occurrences of each word in a text, and *IPCountry* [24] which counts the number of IPs (Internet Protocol addresses) in each country. The goal of both programs is to count elements represented by the key. Further, in both programs the value is a list of numbers and the functionality consists of adding the elements of the lists. In *WordCount* the *key* is each word and the *value* represents the occurrence of the word, and in *IPCountry* the *key* is each country and the *value* represents the existence of IPs associated with the country.

For each program an *MRFlow* graph is created, from which the *tp* are extracted, then the test coverage items are derived, and finally the test case is created. The information of each step is summarized in Table 1, and in brackets is the information relative to the *key* transformations and *values* transformations. The first part focuses on the *MRFlow* graph, the second part summarizes the test coverage items, and in the third part the test case results are described. In the *MRFlow* graph of both programs, the *<key, list(value)>* input variables has one definition and the program contains 3 transformations: transformation of *values* into another variable, no *values* transformation and no *key* transformation. Then the *C-USE-TRANS*/*P-USE-TRANS* are created from these variables: 1 *P-USE-TRANS* in each program, 3 *C-USE-TRANS* in *WordCount* and 5 in *IPCountry*. In the graph, finally, the *EMIT* nodes are created from each emission statement.

From the above graph, the transformation paths (*tp*) are obtained, and then for each *tp* the test coverage items are derived. The *Wordcount* has 5 *tp* and *IPCountry* has 6 *tp*, but in both cases there are 30 test coverage items.

It is not possible to cover 14 of the test coverage items due to some program constraints such as it is impossible to create *values* with empty content, the node *EMIT* is always reached, and so on. The rest of the test coverage items, 16, are covered with two test cases: *<hi, [1,1]>* and *<hello,, [1,1]>* (hello with a comma) for *WordCount*, and *<Spain, [1,1,1]>* and *<###, [1,1,1]>* for *IPCountry*.

The test cases detect two defects because of the non-validation of *key*. If *WordCount* program receives "hello, hello, hello", the expected output is hello:3, but the real output is hello:1, hello,:2 because the *Reduce* function receives an invalid *key* "hello," that is not a word. In *IPCountry* the program fails when it receives a non-country as *key*, for example *Reduce* receives *<###, [1,1,1]>* in the test case and the expected output is nothing because "###" can be an unexpected log/exceptional data but it is not a country.

The two defects found in the programs are caused by the non-validation of input data together with exceptional/non-valid data. In these two programs, *MRFlow* allows to test the functionality with a few test cases that cover many test coverage items.

5. CONCLUSIONS

The *MapReduce* development and programs contain characteristic defects such as the incorrect validation or incorrect processing of different types of data. These defects produce a failure when the *key* or the *values* contain some data that is not correctly processed in the *MapReduce* programs. In this work, the testing technique *MRFlow* is introduced in order to test the *MapReduce* programs. *MRFlow* is based on data flow test criteria and analyzes the program transformations under several situations to cover. This testing technique is applied over two popular programs and with two test cases covers several situations in the transformations which reveal one defect in each program. The faults are caused by the non-validation of *key*, but *MRFlow* in other programs could detect other defects relative to the transformations of *keys* and *values*.

As future work we plan to apply *MRFlow* in more programs and to automate the technique in areas such as test coverage items, the execution of test cases, the derivation of test cases or the graph on which these test cases are derived.

6. ACKNOWLEDGMENTS

This work was supported in part by project TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Science and Technology, and GRUPIN14-007, funded by the Principality of Asturias (Spain) and ERDF funds.

7. REFERENCES

[1] Hadoop: open-source software for reliable, scalable, distributed computing. http://hadoop.apache.org/ Accessed May, 2015.

[2] Institutions that are using hadoop for educational or production uses. http://wiki.apache.org/hadoop/PoweredBy Accessed May, 2015.

[3] Wordcount 1.0. http://hadoop.apache.org/docs/r2.7.0/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0 Accessed May, 2015.

[4] IEEE draft international standard for software and systems engineering–software testing–part 4: Test techniques, 2014.

[5] Alshahwan, N., and Harman, M. State aware test case regeneration for improving web application test suite coverage and fault detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), ACM, pp. 45–55.

[6] Bertolino, A. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering* (2007), IEEE Computer Society, pp. 85–103.

[7] Camargo, L. C., and Vergilio, S. R. Classicação de defeitos para programas mapreduce: resultados de um estudo empírico. In *AST - 7th Brazilian Workshop on Systematic and Automated Software Testing* (2013).

[8] Camargo, L. C., and Vergilio, S. R. Mapreduce program testing: a systematic mapping study. In *Chilean Computer Science Society (SCCC), 32nd International Conference of the Computation* (2013).

[9] Chen, Y., Ganapathi, A., Griffith, R., and Katz, R. The case for evaluating mapreduce performance using workload suites. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on* (2011), IEEE, pp. 390–399.

[10] Csallner, C., Fegaras, L., and Li, C. New ideas track: testing mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), ACM, pp. 504–507.

[11] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[12] Dörre, J., Apel, S., and Lengauer, C. Static type checking of hadoop mapreduce programs. In *Proceedings of the second international workshop on MapReduce and its applications* (2011), ACM, pp. 17–24.

[13] Gudipati, M., Rao, S., Mohan, N. D., and Gajja, N. K. Big data: Testing approach to overcome quality challenges. *Big Data: Challenges and Opportunities* (2013), 65–72.

[14] Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on* (2010), IEEE, pp. 41–51.

[15] Kavulya, S., Tan, J., Gandhi, R., and Narasimhan, P. An analysis of traces from a production mapreduce cluster. In

Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on (2010), IEEE, pp. 94–103.

[16] Kim, K., Jeon, K., Han, H., Kim, S.-g., Jung, H., and Yeom, H. Y. Mrbench: A benchmark for mapreduce framework. In *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on* (2008), IEEE, pp. 11–18.

[17] Kocakulak, H., and Temizel, T. T. A hadoop solution for ballistic image analysis and recognition. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on* (2011), IEEE, pp. 836–842.

[18] Li, N., Escalona, A., Guo, Y., and Offutt, J. A scalable big data test framework. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on* (2015), IEEE, pp. 1–2.

[19] Li, S., Zhou, H., Lin, H., Xiao, T., Lin, H., Lin, W., and Xie, T. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 963–972.

[20] Mattos, A. J. d. Test data generation for testing mapreduce systems. Master's thesis, Universidade Federal do Paraná, 2011.

[21] Mittal, A. Trustworthiness of big data. *International Journal of Computer Applications 80*, 9 (2013), 35–40.

[22] Morán, J., De La Riva, C., and Tuya, J. Mrtree: Functional testing based on mapreduce's execution behaviour. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on* (2014), IEEE, pp. 379–384.

[23] Nachiyappan, S., and Justus, S. Getting ready for bigdata testing: A practitioner's perception. In *Computing, Communications and Networking Technologies (ICCCNT), 2013 Fourth International Conference on* (2013), IEEE, pp. 1–5.

[24] Owens, J. R., Femiano, B., and Lentz, J. *Hadoop Real World Solutions Cookbook*. Packt Publishing Ltd, 2013.

[25] Rapps, S., and Weyuker, E. J. Selecting software test data using data flow information. *Software Engineering, IEEE Transactions on*, 4 (1985), 367–375.

[26] Ren, K., Kwon, Y., Balazinska, M., and Howe, B. Hadoop's adolescence: an analysis of hadoop usage in scientific workloads. *Proceedings of the VLDB Endowment 6*, 10 (2013), 853–864.

[27] Schatz, M. C. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics 25*, 11 (2009), 1363–1369.

[28] Sharma, M., Hasteer, N., Tuli, A., and Bansal, A. Investigating the inclinations of research and practices in hadoop: A systematic review. In *Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference-* (2014), IEEE, pp. 227–231.

[29] Sneed, H. M., and Erdoes, K. Testing big data (assuring the quality of large databases). In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on* (2015), IEEE, pp. 1–6.

Deterministically Testing Actor-Based Concurrent Software

Piet Cordemans
KU Leuven -
Technology campus Ostend
Zeedijk 101
8400 Ostend, Belgium
piet.cordemans
@kuleuven.be

Eric Steegmans
KU Leuven - Department of
Computer Science
Celestijnenlaan 200A
3000 Leuven, Belgium
eric.steegmans
@cs.kuleuven.be

Jeroen Boydens
KU Leuven -
Technology campus Ostend
Zeedijk 101
8400 Ostend, Belgium
jeroen.boydens
@kuleuven.be

ABSTRACT

Non-deterministic concurrent behavior of software prohibits the idempotent property of tests. XUnit frameworks traditionally do not offer support to deal with these concurrency issues which reduces the significance of unit testing concurrent software. In this paper we propose a tool which supports deterministic testing of concurrent software based on the Actor model. This tool reveals race conditions and seamlessly integrates with xUnit-like frameworks. In our approach, a Coloured Petri Net model is constructed per test as well as the code under test. This model allows isolation of concurrent behavior from the effective actor state. Subsequently, the state space is calculated and traces covering all states are constructed. Corresponding with these traces our tool issues test runs, guaranteeing full state space coverage of each test. Moreover, each failed trace can be backtracked, revealing valuable information concerning the race condition.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets*

General Terms

Reliability

Keywords

Concurrent software, Deterministic Testing, Actor Model

1. INTRODUCTION

In concurrent software two major issues exist. On the one hand data races occur when at least two write operations, access the same memory location concurrently and are not synchronization operations. On the other hand race conditions happen when at least two events have multiple orderings. When the correctness of a program depends on a specific ordering, the race condition becomes an issue.

Testing concurrent software is considered difficult because of two conflicting properties. Whereas concurrent software potentially exhibits non-deterministic behavior, software tests desirably execute in a deterministic fashion. Yang [11] described four challenges to deal with testing concurrent software. (1) Detecting unintentional races and deadlock; (2) forcing a path in the state space to be executed; (3) reproducing test execution; and (4) defining test coverage criteria.

By introducing concurrency, the state space quickly enlarges, a phenomena called state space explosion [3]. Non-deterministic behavior is introduced with a conventional scheduler as execution of a path in the state space is indeterminate at run-time in the test. Therefore, it is hard to guarantee state coverage while testing models of concurrent computation with mutable shared state.

1.1 Actor Model

The Actor model defines an actor as a concurrent entity which reacts to messages [2]. Upon receiving a message, an actor can (1) send a number of messages, (2) create a number of actors, (3) change its local state or (4) alter the behavior upon receiving a subsequent message. Messages received are stored in a mailbox from which the actor selects a message to react upon. Once a message is selected to be processed, the actor completes the corresponding action in a single atomic step. As long as messages are immutable, these are messages which do not change once created, the Actor model prevents data races as mutable data is only accessed in an Actor's local state. However race conditions are not prohibited by this model as the ordering of message handling is non-deterministic.

Lu et al. [9] reported that around one third of the non-deadlock concurrency bugs are due to a violation of the intended order by the programmer. Therefore, detecting race conditions requires meticulous testing of the state space, because these issues might exist in a single path of this state space.

1.2 Contributions

We expand on the ideas of applying state space exploration and the Actor model in the context of testing concurrent software. Our goal is to provide a deterministic testing

A-TEST'15, August 30-31, 2015, Bergamo, Italy
ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804327

technique for actor-based software which alleviates the effects of the state space explosion problem. More specifically, with this paper we tackle the challenges as posed by Yang:

- Allow automated unit tests to detect unintentional race conditions.
- Construct the state space of unit tests and forcing the execution of a test to follow a specific path in its state space.
- Provide information on paths leading to a failing test, in order to replay the paths of interest.
- Guarantee state coverage in the state space of the test.

Furthermore, we implement a lightweight tool implementing the model which seamlessly integrates with the specific run-time environment. More specifically, it integrates with an x-Unit and Actor model framework.

The paper is organized as follows: in section 2 we describe the model which allows to construct the traces in unit tests for software based on the Actor model. Then, in section 3 we describe a tool implementing the model.

2. ACTOR STATE SPACE EXPLORATION

In order to deterministically test actors, the state space of tests must be fully explored. However, to deal with the state space explosion problem, the concurrent behavior should be isolated from the state of the actors. This results in a state space which does not represent the local state managed by the actors. Rather it only contains the state of actor mailboxes and the different actor life cycle states. Once this state space has been constructed, paths of specific message ordering are composed.

2.1 Coloured Petri Net of the Actor Model

In our approach, we model the test and actors under test with a Coloured Petri Net (CPN) [7] model. This CPN model isolates concurrent behavior of actors and partitions the state space of the actor system. From this model, the state space can be constructed, as well as the minimum set of paths in order to visit each state at least once.

A CPN combines Petri Net (PN) modeling with features of high level programming languages. Most importantly, CPNs introduce the concept of a color set and token color which respectively describes place and token types. This type system allows to construct models which are more concise than regular PNs, while maintaining the possibility to decompose any CPN to a regular PN. PN models and by extent CPN models are well suited to model parallel computation, as their execution semantics are inherently non-deterministic.

Figure 1 represents a simplified CPN model of a single actor. *Idle*, *mailbox* and *processing* are states, while *receive* and *return* are transitions. Both *idle* and *mailbox* contain a token, respectively the *actor state* token and the *mailbox* token. The *mailbox* token is a list of messages. The arrows with annotations between transitions and states describe the behavior when firing the transition. For instance, when firing *receive*, the *actor state* token of *idle* is consumed, a message is consumed from the *mailbox* state, while the *mailbox* token is returned to the *mailbox* state and a tuple token of *message* and *actor state* is produced in the *processing* state.

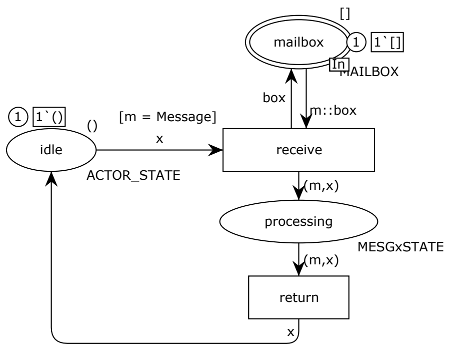


Figure 1: Simplified CPN model of the Actor model

2.1.1 Isolating Concurrent Behavior

Figure 2 is the generic CPN model of a single actor. Two tokens are always present in this model, one to depict actor state, while the other is the representation of its mailbox. The typical message reception procedure is as follows: upon reception of a message, a token is added to the mailbox token. This activates the respective receive transition, on condition that the actor state token is in the idle state. Subsequently, the token traverses to the processing state and is incremented to depict a new local actor state. While the token resides in the processing state, no other messages will be processed. After the processing state, the token for actor state returns to idle which enables the receive transitions to process a new message. After processing, the four resulting effects can be defined as follows:

1. Change of local state: local decisions result in a corresponding action. This includes either no continuation effect or one of the other resulting effects. Nevertheless, the local state of the actor does not affect the state space of concurrent behavior.
2. Change behavior upon reception of a subsequent message: by counting the number of received messages the subsequent concurrent behavior can be selected. However, the resulting effect does not affect the concurrent state space.
3. Send X messages: tokens are produced in the respective mailboxes of the recipients, as shown in the *Msg* branch of Figure 2.
4. Create N actors: tokens are produced in the activation places of the child actors. A token from the activation and the *notAlive* place are needed to activate the transition to the *idle* place. The activation place is an input socket in the hierarchical CPN model.

Once the message has been processed, its state on the one hand is modeled as either:

1. return the state token to the *idle* state,
2. return the state token to the *notAlive* state.

On the other hand the continuation behavior has one of three possible actions:

1. there is no effect on the test entity or any other actor,

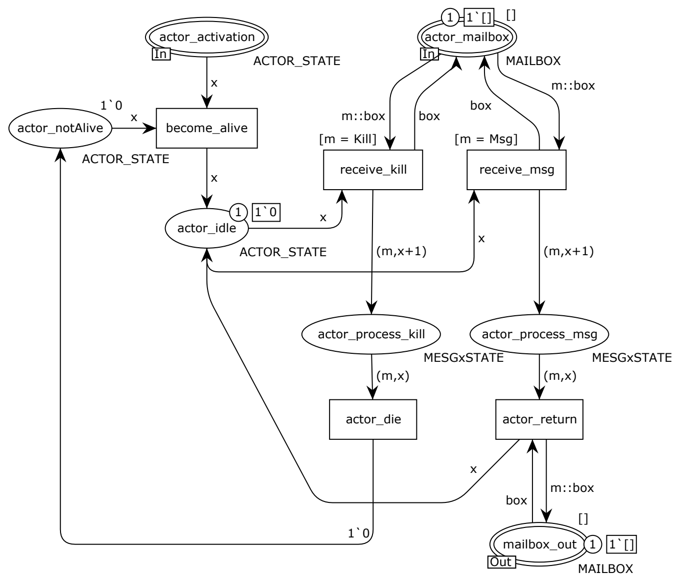


Figure 2: CPN model of the Actor model

2. generate X tokens in Y mailbox places. With X and Y integers equal or larger than 1,

3. generate N tokens in M activation places. With N and M integers equal or larger than 1.

2.2 Hierarchical Test Model

The top level CPN contains all actors involved, the test actor, and a set of initial messages. The generic CPN model for actors provides two places accessible from the top level CPN model. These are the mailbox and activation place. Tokens are produced in these places from the continuation transitions of other actors.

Finally, the top level CPN model contains a representation of the xUnit test definition. This entity initiates the test and captures the results. Furthermore, the test behaves as an actor, because it has an implicit mailbox to allow message-based communication. Therefore, the CPN of the test definition is derived from the generic CPN of the Actor model. The generic CPN actor model can be reduced, because test actors do not need life cycle management. Instead, a test has a place which contains the set of messages to drive the test. Additionally, this place is represented in the top level CPN.

2.2.1 *Constructing the Test State Space*

The state space can be deduced from the test CPN model. Each state represents a particular set of tokens, state and message tokens, at the corresponding places. Arcs between these places are transitions which upon activation reach the designated state. This is an implementation of the basic algorithm for state space construction as described by Jensen and Kristensen [7]. This algorithm generates the state space of concurrent behavior with regard to the test.

However, this set of states might consist of unreachable states. Namely, some states represent the path of the continuation of a local decision branch. Due to the test setup, only a single path is chosen in the set of possible continuations. On account of isolating concurrency from local actor state, it is impossible to indicate which states cannot been chosen in the model for concurrent behavior. However, by considering each state of the set of continuation states as valid, the effective continuation behavior will be identified at run-time.

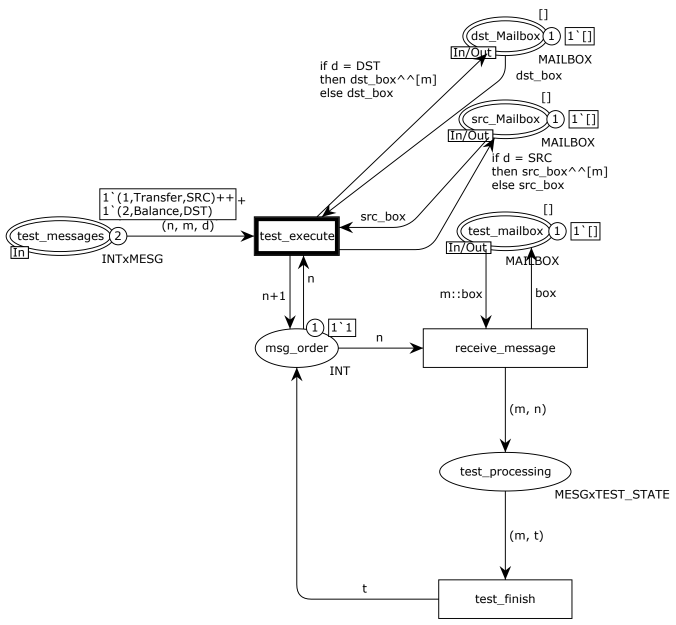


Figure 3: CPN model of the test actor

2.3 Deterministic Traces

The purpose of constructing the state space of concurrent behavior is to determine message ordering in the test. In this state space, traces are designated to provide coverage of the state space of the test. Traces are paths in the state space graph which are chosen deterministically, guaranteeing coverage and reproducibility. As the number of traces is proportional to the run-time performance of the tool, the number of traces per test needs to be minimized.

When considering test coverage, two different viewpoints can be adopted. First is the coverage of the state space of the test, i.e. state space coverage with the set of messages and local actor state as defined by the test. This deals with the problem of non-deterministic test execution. Furthermore is the coverage of the test state space as part of the larger system. Namely which partition of the state space of the larger system is covered by the test.

2.3.1 *State Space Coverage of a Single Test*

In order to guarantee determinism in a test with actors, state coverage of the concurrency state space is sufficient. Namely, non-determinism is introduced when multiple messages are bound to arrive at a single actor. Therefore, states with different message tokens at a mailbox place determine the effect of this non-determinism. Consequently, the occurring binding element which led to this state is insignificant for the purpose of identifying race conditions. Furthermore, outgoing arcs are either the sequential continuation effect of an actor, or an unrelated event to the current token in the actor state.

With respect to determinism guarantees, the minimum set of traces to cover all states is proportional to the maximum number of messages in concurrent execution across all states. For instance, a test containing only actors forwarding a single message, will not contain any non-determinism. In effect, only a single trace will be generated for this test. On the other hand, a test with n concurrent messages, effectively leads to $n!$ traces, as $n!$ represents the combinatorial set of message orderings. In general, in order to cover all states a minimum set of $n!$ traces will be needed with n being the maximum number in a concurrency race. In order to construct these traces, a depth-first search algorithm for a directed acyclic graph is implemented.

2.3.2 *Partitioning the State Space*

Tests partition the state space of the system under test. Namely, a unit test consists of a limited set of actors and messages. Moreover, unit tests typically focus on a specific functionality of the system, thus most unit tests only explore a single logical path. Multiple tests are combined in a test suite to cover a larger set of states in the state space. This rationale does not change for concurrent software, however the strategy of state space exploration relies on the narrow focus of unit tests to be scalable. Namely, the combinatorial set of messages in concurrency, n, is defining for the worst case of the number of traces in the state space. With a limited set of messages and actors, the resulting set of traces, is limited, especially when considering the size of the state space of the system.

3. IMPLEMENTATION

In order to test the CPN model of the Actor model, a tool called ActorRunner has been developed. ActorRunner is implemented in Scala with the Akka actors library [6]. The x-Unit test runner of choice is ScalaTest and JUnit [1].

3.1 ActorRunner

The purpose of ActorRunner is to accept a set of traces and adapt the execution of unit tests to match these traces. This tool integrates with a conventional x-Unit framework and seamlessly intercepts and resends messages, in order to control message ordering. The general structure of this tool is illustrated in Figure 4.

For each test, ActorRunner starts with a set of traces which have been derived from the unit test and actors under test. For each trace, ActorRunner issues the x-Unit framework to run the test anew. Furthermore, ActorRunner instructs the marshal component with a list of states to conduct the test in a specific message ordering. Following these states, the test explores a specific path in the state space of the test. Finally, the test executes its assertions and determines whether the test passes or fails. Consequently, if there are traces which have not been executed yet, ActorRunner issues another test run with a different trace. Eventually, all the individual test results are aggregated. If a single trace of the test fails, the test is considered to fail altogether. Trace information is added to the test report, to facilitate debugging on the race condition. The coverage criteria, as defined in Section 2.3.1 ensure that the test runs deterministically, regardless of processor load, or properties of the non-deterministic scheduler.

The marshal component is an actor introduced to intercept all messages and resend them in compliance with the order of trace input. In order to intercept all messages, all

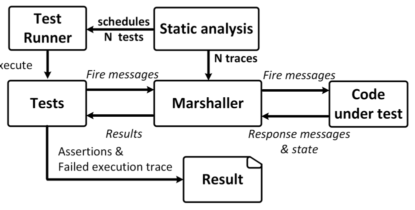


Figure 4: ActorRunner takes a set of traces as input, schedules N tests and reorders messages as defined in the trace.

actors are created under supervision of the marshal actor. Furthermore, once the test sends its first message the test actor reference is registered. At creation, instead of returning the real actor reference, a proxy is returned which redirects all messages to the marshal.

In the marshal all messages are gathered, as well as the current state in the trace is indicated. As the marshal is an actor, it acts upon the arrival of messages. Once the set of conditions have been obtained to advance to the next state, the marshal component advances. In effect, these conditions are defined in the following state of the trace. On the one hand these condition can be reached as messages arrive as defined in the following state of the trace. On the other hand one of the actors under test processes a message and the marshal actor internally continues to the next state.

3.2 Limitations

This approach is limited by the implementation of the Actor model. Namely, only race conditions can be detected, while data races should be prohibited by the Actor model itself. However, should a programmer violate against this condition, by sharing mutable data, sending mutable messages or no longer ensuring the Actor behavior as atomic, the deterministic state space exploration approach will not be able to detect these concurrency issues or correctly identify race conditions.

Furthermore, regarding liveness issues, such as deadlock or livelock, this approach will invoke the conditions leading to this behavior. However, depending on the properties of the testing framework, it will likely lead to a time-out, without any valuable debugging information. In effect, this approach is ineffective in detecting these issues.

4. RELATED WORK

State space exploration has been introduced by Edelstein et al. [5]. They proposed to explore the state space by rerunning existing tests, while manipulating thread interleaving. This technique allows to explore and replay different paths in the state space of the test. However, their approach suffers from the state space explosion problem, as each atomic operation can be interleaved. Moreover, each of the test runs is slowed down by the run-time performance cost of the multitude of context switches. Therefore, the tool provides a heuristic solution to detect concurrency problems which allows configuration of the number of context switches to explore.

Chess [10] is a tool which conducts state space exploration for .NET. However, this implementations did not deal with the state space explosion problem and relied on heuristics to indicate concurrency problems. Namely the number of explored thread interleaving is limited to constrain the number of paths explored.

State space exploration has been adopted in Basset [8] for Actor systems. Lauterburg et al. [8] continued on the idea of state space exploration. Instead of manipulating thread interleaving on the level of bytecode, they apply it to a higher level model, more precisely, the Actor model. Lauterburg et al. decided to build a model checker for actor programs based on Java Pathfinder [4]. However, instead of focusing on tests, their tool explores the state space of the whole actor program. By doing so, the state space explosion problem deteriorates, and to mitigate this effect, the tool is based on a heuristic to linearize the set of states. Furthermore, ActorRunner does not need to run an adapted JVM which Basset needs for Java Pathfinder.

5. FUTURE WORK

Firstly, decisions based on local state might change the concurrent behavior. Consequently, a similar test with different modalities might cover a different state space. Yet, as part of the concurrent state space analysis, these tests will partition a part of the state space of the test and even share some states. This is a possible optimization which might reduce the number of states and traces. Moreover when considering the test space over a multiple test span, it might be possible to indicate concurrency states which are not covered by the test suite, due to local state. This information might be included in a coverage report, so that a tester is aware that the test suite might be lacking some tests.

Secondly, the deterministic state space exploration approach requires an extensive case study, within a larger existing code base. This will allow to prove the feasibility and scalability of this approach. This will also require that some steps, such as CPN generation from code become automated, as described by the scheme in Section 2.1.

6. CONCLUSION

Due to indeterminism, the result of a test run on concurrent software is not reliable. In order to deal with this problem, this paper described an approach to conduct deterministic state space exploration for the Actor model. This approach allows to deterministically run a conventional test suite for concurrent software, as defined by the criteria of Yang. Coloured Petri Net models of tests isolate the concurrent behavior from the local state of the actors. This partitions the state space of the system, in which a limited set of traces allow to cover all states of tests. A marshal actor is introduced which reorders messaging in tests according to the generated traces. By aggregating the results of all traces, tests become deterministic. We implemented this approach in a tool called ActorRunner. This tool provides a proof of concept to introduce a seamless message scheduling system which in the context of unit testing is scalable, contains valuable debugging information and is effective to detect race conditions in a deterministic fashion.

7. REFERENCES

[1] Junit.org: Resources for test driven development, http://www.junit.org/.

[2] G. A. Agha. Actors: a model of concurrent computation in distributed systems. 1985.

[3] R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351. Springer, 1997.

[4] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder-second generation of a java model checker. In *In Proceedings of the Workshop on Advances in Verification*. Citeseer, 2000.

[5] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[6] P. Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 1–6. ACM, 2012.

[7] K. Jensen and L. Kristensen. *Coloured Petri Nets: modelling and validation of concurrent systems.* Springer, 2009.

[8] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of java-based actor programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 468–479. IEEE Computer Society, 2009.

[9] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes - a comprehensive study on real world concurrency bug characteristics. In *Architectural Support for Programming Languages*, 2008.

[10] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*, 2008.

[11] C.-S. D. Yang. *Program-based, structural testing of shared memory parallel programs.* PhD thesis, University of Delaware, 1999.

Concurrent Software Testing in Practice: A Catalog of Tools

Silvana M. Melo, Simone R. S. Souza, Rodolfo A. Silva, and Paulo S. L. Souza
Institute of Mathematics and Computer Sciences, University of São Paulo
Avenida Trabalhador São-carlense, 400 - 13566-590
São Carlos, São Paulo, Brazil
{morita, srocio, adamshuk, pssouza}@icmc.usp.br

ABSTRACT

The testing of concurrent programs is very complex due to the non-determinism present in those programs. They must be subjected to a systematic testing process that assists in the identification of defects and guarantees quality. Although testing tools have been proposed to support the concurrent program testing, to the best of our knowledge, no study that concentrates all testing tools to be used as a catalog for testers is available in the literature. This paper proposes a new classification for a set of testing tools for concurrent programs, regarding attributes, such as testing technique supported, programming language, and paradigm of development. The purpose is to provide a useful categorization guide that helps testing practitioners and researchers in the selection of testing tools for concurrent programs. A systematic mapping was conducted so that studies on testing tools for concurrent programs could be identified. As a main result, we provide a catalog with 116 testing tools appropriately selected and classified, among which the following techniques were identified: functional testing, structural testing, mutation testing, model based testing, data race and deadlock detection, deterministic testing and symbolic execution. The programming languages with higher support were Java and C/C++. Although a large number of tools have been categorized, most of them are academic and only few are available on a commercial scale. The classification proposed here can contribute to the state-of-the-art of testing tools for concurrent programs and also provides information for the exchange of knowledge between academy and industry.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.1.3 [**Programming Techniques**]: Concurrent Programs

General Terms

Systematic review, Software Testing, Concurrent programs

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804328

Keywords

Systematic mapping, Concurrent programs, Testing tools

1. INTRODUCTION

The activities of Verification, Validation, and Testing ensure quality of the software. Software testing is the process of executing a program for finding errors. Mistakes can occur in the software development process, therefore, the testing activity should be conducted throughout the software development cycle. Different testing phases, namely unit testing, integration testing, functional testing, system testing and acceptance testing should be performed. This study focuses on unit testing tools, in which each system module is tested separately so that logical and implementation faults can be found [71].

Testing techniques, such as structural, functional, and fault-based testing proposed to sequential programs have been adapted for use in concurrent programs. Other techniques have been developed specially for concurrent programs and consider features, as non-determinism, synchronization and communication of concurrent/parallel processes. They also look on common mistakes found in the concurrent software, such as race conditions, deadlocks, livelocks, and atomicity violation.

The use of concurrent software has increased, mainly because of the availability of multicore processors and computer clusters. Modern business applications use concurrency to improve the overall system performance, consequently, a variety of testing techniques (and their associated tools) have been proposed to test concurrent programs. However, no classification methodology of testing tools that helps the testing practitioner in the analysis and selection of a tool adequate to their needs has been designed. This paper proposes a new classification for a set of testing tools for concurrent programs regarding attributes, such as testing technique, programming language and paradigm of development. A useful categorization is provided to guide the tester during the selection of testing tools for concurrent programs.

The paper is organized as follows: Section 2 presents the concepts and challenges related to concurrent software testing; Section 3 provides a catalog with 116 testing tools for concurrent programs with some of their descriptions; finally, Section 4 addresses the conclusions and future work.

2. CONCURRENT SOFTWARE TESTING AND CHALLENGES

Concurrent programming enables a smart use of features

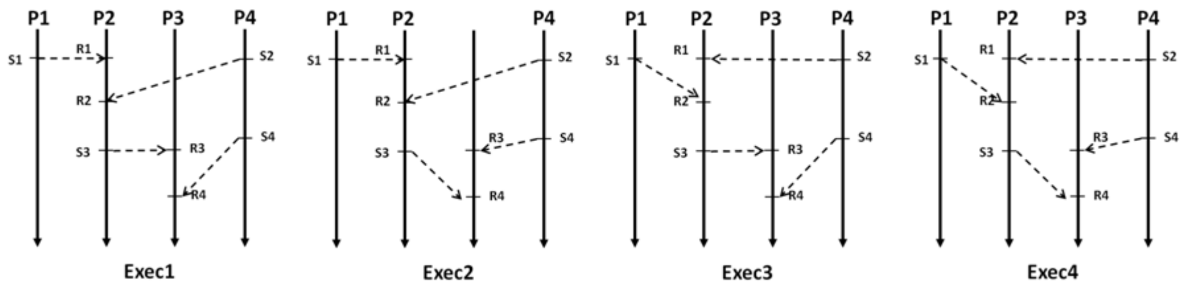


Figure 1: Example of non-determinism in concurrent programs.

for the increase in efficiency (in terms of time of execution), avoiding idleness of resources (as it occurs in the sequential process) and lowering computational costs [32]. However, some challenges may raise in the testing of such programs. The non-determinism enables different executions of a program with a single input and production of different and correct outputs. This non-deterministic behavior is due to communication and synchronization of concurrent (or parallel) processes (or threads). Figure 1 shows an example of non-determinism, in a program composed of four parallel processes. In *Exec1* a race condition occurs between *s1* and *s2*, related to *r1* and *r2*, and *s3* and *s4* related to *r3* and *r4*. Each execution represents a likely synchronization sequence in the concurrent program. The testing activity identifies all possible synchronization sequences and analyzes the outputs. The deterministic execution technique can be used to force the execution of a sequence for a given input in the presence of non-determinism [57].

Other features related to communication and synchronization between processes (or threads) impose challenges on concurrent program testing, such as development of techniques for static analysis, detection of errors related to synchronization, communication, data flow, deadlocks, livelocks, data race, and atomicity violation, adaptation of testing techniques for sequential programming to concurrent programs, definition of a data flow criterion that considers message passing and shared variables, automatic test data generation, efficient exploration of *interleaving* events, reduction of costs in testing activities, deterministic reproduction for a given synchronization sequence, and representation of a concurrent program that captures relevant information to the test.

Studies in the domain of software testing for concurrent programs have proposed solutions for such problems and some testing tools have been developed to support the utilization of the techniques. The need for the execution and testing of different synchronization sequences and the deterministic execution of the program are solutions to this issue. However, they impose high costs on the testing activity. Regarding of this, we consider the building of tools to automatize this activity very promising.

Li et al. [41] propose a taxonomical overview of software testing tools for both sequential and concurrent programs. The classification is based on testing activities and testing stages. The considered activities were test planning/designing, test generation, test execution, test adequacy, test feedback/fault localization, assess readiness and test process management. In relation to testing stages, the following stages are covered: static checking, unit testing, integration testing, system testing/ maintenance testing. In relation to concurrent testing, the authors cite just one model checking tool. Differently, in this paper we present several testing tools for concurrent programs, mainly for the unit testing stage.

Muhammad and Labiche [97] conducted and described a systematic review on state-based testing tools. They proposed a classification of the tools found. The authors highlight that just a few commercial tools were found in the review. The authors argue that this happened due the use of only academic databases for selection of studies. In our study we face with the same problem, but nevertheless, we believe that the academic databases are the most reliable bases for systematic mapping.

3. A CATALOG OF TESTING TOOLS FOR CONCURRENT PROGRAMS

We conducted a systematic mapping (following the process defined by Petersen et al. [80]) to identify tools proposed for testing concurrent programs. The focus of this paper is not the systematic mapping and, therefore, details about the mapping are not shown due to space restrictions The conducted mapping was more extensive, including other research questions (out of scope of this paper). Thus, only the necessary information to understand how the catalog was generated is shown here. A search string was defined with the words "testing", "concurrent software" and their synonyms. The search was performed in 5 research databases and 6316 papers were returned, of which 334 were selected. We identified 116 different testing tools for concurrent programs. Figure 2 shows the number of testing tools developed from 1992 to 2014.

We can observe a continuous increase in the number of papers in this research area. The bubble chart in Figure 3 illustrates the current state-of-the-art of the concurrent software testing domain in relation to the total number of tools available for each testing technique proposed and programming language supported.

Although a large number of supporting tools for concurrent program testing has been proposed, their maturity level should be analyzed. Most tools represent *concepts proof* of academic proposals, which may be a threat to the validity of this study that considered only academic data bases to conduct the search of primary studies. Finding commercial tools is hard because the vendors offer only user's manuals

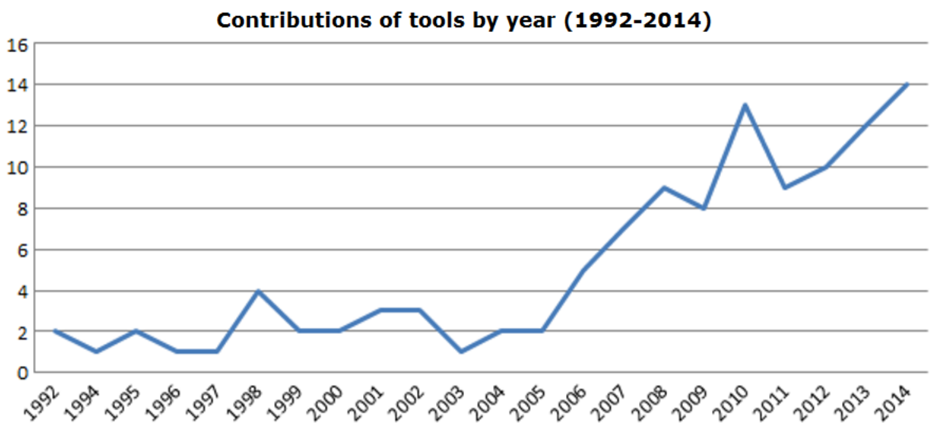


Figure 2: Proposition of concurrent testing tools over the years (1992-2014).

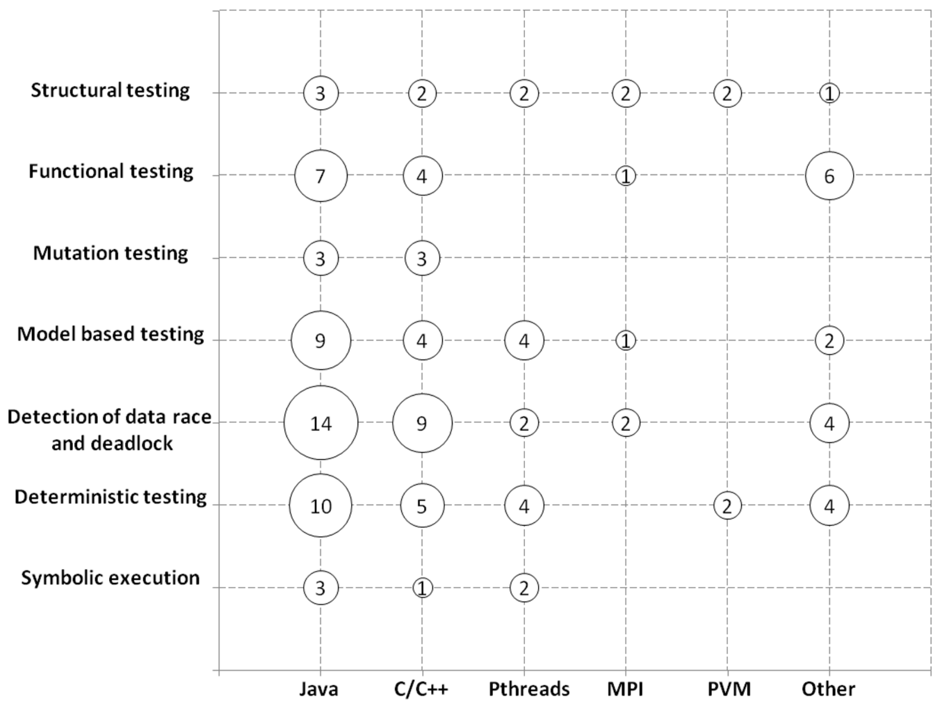


Figure 3: Testing tools by testing approach and implementation language.

and case studies with no technique information in scientific paper for proprietary reasons. The transference of technology from the academy to the industry still remains a challenge in the concurrent software testing domain. Therefore, a closer interaction between the interests of academy and industry is required so that a feedback loop can be created between them.

We have defined a set of relevant attributes to classify the concurrent testing tool selected from the systematic mapping. The definition was based on features of the concurrent programs and information considered relevant for the tester to select the desired testing tool. The following attributes were defined: testing technique, paradigm programming, and language supported. Based on such attributes, we have developed a catalog of tools for testing concurrent programs, shown in Table 1. Subsections 3.1 and 3.2 address some most important tools divided into two groups: one containing tools that apply testing techniques (functional, structural, and mutation testing) and another with tools that test specific characteristics of concurrent programs (model checking, deadlock and data race detection, deterministic testing, and symbolic execution).

3.1 Structural Testing Tools

For the structural testing technique, **ValiPar** [105] supports the application of control flow and data flow criteria for concurrent programs in different programming languages and using different paradigms of development. For programs that use the message-passing paradigm, **ValiPVM** [103] supports the testing of programs in PVM (*Parallel Virtual*

Table 1: A testing tools catalog for concurrent programs

Technique	Paradigm	Language	Tools
Structural testing	Shared memory	Pthread	ValiPthread [88], DellaPasta [118]
	Message passing	MPI	ValiMPI [35]
		C	Monitoring tool [40], Maple [121]
		Pascal	Steps [51], Pet [33]
		PVM	ValiPVM [103]
	Both	Ada	CATS [120]
		Java	ValiJava [104], New JLint [5], JML toolset [4]
		C	Valipar [105]
Functional testing	Shared memory	Java	Oshajava [116], Tiddle [86], Ndetermin [10], Race-Fuzzer [93], Rstest [107]
		C	TMUnit [34], Storm [83], Relaxer[11]
	Message passing	MPI	ISP-GEM [38]
		Ada	TSG [13]
	Both	UML	TCaseUML [2]
		PLINQ	SLUG [108]
		Ada	TCgen [47]
		C/C++	ATEMES [49]
Mutation testing	Shared memory	Java	Javalanche [91], MutMut [30], ConMan [8]
		C, C++	Comutation [31], CCmutator [54]
	Message passing	MPI	ValiMut [100]
Model Based testing	Shared memory	Java	Vyrdmc [25], Cute [94], Fusion [113], Bandera [21], TJT [1], TIE [65], SearchBestie [55]
		C, C#, Java	Chess [70]
		C, C++	CDSchecker [74], Inspect [119]
		C, Pthread	Concurrit [9], C2Petri [48], RegressionMaple [110]
		.Net	Gambit [20]
		C#, Java, D	DemonL [115]
	Message passing	C	Magic [14]
		C, MPI	MPI-SPIN [99]
	Both	C, C++	VIP [23]
		LISP	Spin [36]
		Java, LTL	EDA [106]
Data race and deadlock detection	Shared memory	Java	Droidracer [66], ConEE [76], Carisma [123], Jcute [95], Concrash [64], Contest [53], Epaj/Eprfj [90], Have [17], Javapathfinder [112], Omen [87], Penelope [102], RccJava [27], Enforcer [6], Calfuzzer [92], ConcJunit [85], Kivati [18]
		C, C++	ConMem [3], Ctrigger [79], Light64 [72], Pike [28], SPin [12], Racez [98], MultiRace [81], ThreadSanitizer [96], Gadara [114]
		C, Pthread	MDAT [56]
		.Net	Colfinder [117], AutoRT/CorrRT [43]
		UPC	UPC-Check [22]
		Fortran	Eraser [68]
	Message passing	C, MPI	Marmot [50], MPIRace-Check [78]
Deterministic Testing	Shared memory	C, C++	Dthreads [59], InstantCheck [73], DeSTM [84]
		Pthreads	Kendo [77], FPDet [124], Synctester [122], DetLock [69]
		Java,C, C++	RichTest [58]
		Java	Conan [60], IMunit [42], Dejavu [19], SAM [16], Cooperari [67], Java PathExplorer [37], TransDPOR [109]
		C	Direct [15]
		Titanium	Titanium [46]
		C++, Pthreads	RFDet [62]
		STM,C,C++	DeTrans [101]
		Ruby	DPR/TARDIS [63]
	Message passing	PVM	Viper [75]
		C, PVM	DEIPA [61]
		Ada	SpyLayer [7], AIDA [24]
Symbolic execution	Shared memory	C	Concrest [26]
		Java	SPF [82], Z3 [44], LCT [45]
		C/C++/Java	BEST [29]
		C/Pthread	MultiOtter [111], CDT-Eclipce [39]

Machine) and **ValiMPI** [35] for programs in MPI (*Message Passing Interface*). For programs that use the shared memory paradigm, **ValiPthread** [89] tests programs using Posix standard for *threads* (PThreads) and **ValiJava** [104] supports the testing of Java concurrent programs. Other tools, such as **STEPS** [52] and **Dellapasta** [118] use a graphical representation of the program to derive test cases and apply coverage testing criteria to evaluate the testing activity. **MonitoringTool** [40] the coverage of concurrent programs according to the testing criterion *k-tuples of concurrent commands*, proposed by the same authors. This criterion requires implementation of all sequences of *k* length concurrent commands. This tool can be applied to concurrent C programs and the coverage analysis is achieved by monitoring of the testing execution. Mechanisms to force the execution of concurrent commands are implemented on tool.

3.2 Functional Testing Tools

For functional testing technique, **OSHAJAVA** [116] uses dynamic analysis to test the specification of concurrent programs written in Java annotations. The instrumentation of the bytecode is used to set each "write" operation with the state of the communication updated and the "read" operation to check if a method violated or not its specification. The semantic formalism is used to indicate when a dynamic operation has violated the specification of an inter-thread communication, so that the safety properties of multithreaded programs can be checked. Other tools, such as **SLUG** [108] and **Ndetermin** [10] also use a program specification to derive test cases and evaluate the testing results.

3.3 Mutation Testing Tools

For mutation testing, **MutMut** [30] proposes an approach for an efficient execution of mutants in multithreaded programs. It uses a technique for the selection of mutants to be executed. When the original program is executed, the technique selects points in the code for mutation considering relevant aspects of the concurrent programs. The approach also enables the tester to select a *thread* to be executed, forcing the mutation introduced to be executed. **ConMan** [8] implements a set of mutation operators for concurrent programs in Java (J2SE 5.0). The mutation operators are classified into operators that modify critical regions, keywords, and calls for concurrent methods and operators that replace concurrent objects. **CCmutator** [54] implements those operators as well as new specific mutation operators for concurrent programs in *PThreads*. It utilizes the High Order Mutation technique, in which two or more mutations are inserted in the program for the creation of strong mutants and improvements in the quality of the testing case set. **Comutation** [31] uses selective mutation based on the mutation operators for concurrent Java programs. Selective mutation selects a subset of mutation operators in which test cases that have a high mutation score for this subset also feature for the other operators. The objective is reduce the mutation testing cost.

3.4 Model Checking Testing Tools

The model checking technique has been widely used in concurrent software testing and enables the analysis of system properties by a formal model. It can also be used to explore the state space of a system. Techniques for state space reduction are used to limit the testing search space. **Inspect** [119] uses model checking for concurrent programs in C language. The exploration of relevant interleavings is facilitated by the use of an executable model of the instrumented version of the program and enables the tool to communicate with the scheduler. **CHESS** [70] implements a model checker to analyze the correctness of concurrent programs in relation to the expected properties (e.g. *interleavings*) derived from a test scenario. Testing scenarios are defined by the tester and explore all possible synchronizations among threads. **Magic** [14] analyzes events and states of the operating system. The temporal logic language LTL (*Linear Temporal Logic*) is used to instantiate finite state machines. Also considering a concurrent system formalized in LTL, it is proposed **SPIN** [36] which implements a model checker to analyze the correctness of concurrent systems in relation to the properties formally defined. This tool is instantiated for the MPI pattern, **MPISpin** [99] and later used as the basis for verification of concurrent code in Java, **Bandera** [21].

3.5 Deadlock and Data Race Detection Tools

Carisma [123] implements a data race detector based on statistic sampling. A program, in a single site of the code, can perform multiple accesses to the memory, therefore, the tool uses an analysis of the trace of execution to estimate and distribute sampling between such locations and collects a fraction of all memory accesses. The information assists the tool in detecting data races. In an attempt to prevent data races, programmers generally write a code that will result in a deadlock when executed with some inputs, due to the misuse of synchronization primitives. Some tools, such as **Gadara** [114], **Marmot** [50], and **UPC-Check** [22] address the problem of deadlock detection. They analyze the code and insert delays into it to force the execution of a given synchronization sequence and then detect the presence of deadlocks, or monitor the execution through a scheduler of processes. **Javapathfinder (JPF)** [112] was developed by NASA Research Center. It uses model checking to detect deadlock and data race in Java programs (bytecode). The user can also define the property classes to be analyzed. JPF monitors the execution, extracts events (synchronization and communication) that occur and analyzes them through an observer process. The observer performs a verification based on the information of the monitoring and information of an analysis of error pattern. JPF is especially useful for the verification of concurrent Java programs due its systematic exploration of scheduling sequences of threads, which is a difficult task in traditional testing tools. **MPIRace Check** [78] performs data race detection for programs in MPI by checking the communication messages between the processes.

3.6 Deterministic Testing Tools

Tools are developed for provide threads control and deterministic execution/re-execution in a non-deterministic environment. They usually store information about a preliminary execution (traces) to enable its re-execution, performing the same synchronization sequence. **Dejavu** [19] records thread schedules and the reproduction of a schedule in a controlled execution. **Dthreads** [59] ensures deterministic execution, even in the presence of data race, forcing the program to produce the same output for each input sequence. **SPY-Layer** [7] records and re-runs concurrent or distributed Java programs, verifying and validating synchronization sequences. The re-execution is used for error detection.

3.7 Symbolic Execution Tools

Symbolic execution is a powerful technique for the exploration of systematic paths of a program with symbolic values as inputs. **MultiOtter** [111] uses a symbolic executor to trace values following the control flow of the program and conceptually changes the execution if it finds a conditional dependence of a symbolic value. **LCT** [45] uses a combination of dynamic and symbolic executions, known as *Concolic testing*, in which the program under testing is executed in a hybrid way with real test data and symbolic values for the exploration of different behaviors of the program.

4. CONCLUSIONS

This paper presents a catalog that has addressed the state-of-the-art of concurrent software testing area. The study covered the period from 1992 to 2014 and 116 testing tools were identified and classified into different testing techniques and programming languages. We strongly believe the catalog of tools and the other results provided in this study will be useful for future research and also to help practitioners of the area in the selection of testing techniques and tools.

The results also show concurrent software testing is still a domain for new studies and a research trend. In recent years, researchers have concentrated their efforts mainly on the C/C ++ and Java languages and on techniques for concurrent context, such as: formal verification techniques, model checking, static and dynamic analysis and deterministic execution. Many tools implement a testing approach that combines different testing techniques for increases in the quality of testing.

In future studies, we aim at the development of an online iterative catalog with information on all tools identified by each technique, paradigm, language and others important attributes. Additional research will be focus on analyses of the benefits of the catalog to different stakeholders (testing practitioners, enterprises and researchers) and how such techniques and tools can be employed to improve higher software quality.

5. ACKNOWLEDGMENT

This research is sponsored by FAPESP under process no. 2013/05046-9 and 2013/01818-7.

6. REFERENCES

[1] D. Adalid, A. Salmerón, M. D. M. Gallardo, and P. Merino. Using spin for automated debugging of infinite executions of java programs. *J. Syst. Softw.*, 90:61–75, Apr. 2014.

[2] C. ai Sun. A transformation-based approach to generating scenario-oriented test cases from uml activity diagrams for concurrent applications. In *COMPSAC*, pages 160–167. IEEE Computer Society, 2008.

[3] B. Aichernig, A. Griesmayer, E. Johnsen, R. Schlatte, and A. Stam. Conformance testing of distributed concurrent systems with executable designs. In F. de Boer, M. Bonsangue, and E. Madelaine, editors, *Formal Methods for Components and Objects*, volume 5751 of *Lecture Notes in Computer Science*, pages 61–81. Springer Berlin Heidelberg, 2009.

[4] W. Araujo, L. Briand, and Y. Labiche. On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software. *Software Engineering, IEEE Transactions on*, 40(10):971–992, Oct 2014.

[5] C. Artho. Finding faults in multi-threaded programs. Master's thesis, Swiss Federal Institute of Technology ETH Zŭrich, Zŭrich, 2001.

[6] C. Artho, A. Biere, and S. Honiden. Enforcer - efficient failure injection. In *Proceedings of the 14th International Conference on Formal Methods*, FM'06, pages 412–427, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] A. Bechini, J. Cutajar, and C. Prete. A tool for testing of parallel and distributed programs in message-passing environments. In *Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean*, volume 2, pages 1308–1312 vol.2, May 1998.

[8] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation operators for concurrent java (J2SE 5.0). *Workshop on Mutation Analysis*, page 11, 2006.

[9] J. Burnim, T. Elmas, G. Necula, and K. Sen. Concurrit: Testing concurrent programs with programmable state-space exploration. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 16–16, Berkeley, CA, USA, 2012. USENIX Association.

[10] J. Burnim, T. Elmas, G. Necula, and K. Sen. Ndetermin: Inferring nondeterministic sequential specifications for parallelism correctness. *SIGPLAN Not.*, 47(8):329–330, Feb. 2012.

[11] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 122–132, New York, NY, USA, 2011. ACM.

[12] Y. Cai, W. Chan, and Y. Yu. Taming deadlocks in multithreaded programs. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 276–279, July 2013.

[13] R. Carver and R. Durham. Integrating formal methods and testing for concurrent programs. In *Proceedings of the Tenth Annual Conference on Computer Assurance, 1995. COMPASS '95. Systems Integrity, Software Safety and Process Security.*, pages 25–33, Jun 1995.

[14] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, V17(4):461–483, December 2005.

[15] K. Chatterjee, L. de Alfaro, V. Raman, and C. Sánchez. Analyzing the impact of change in multi-threaded programs. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, FASE'10, pages 293–307, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Q. Chen, L. Wang, and Z. Yang. Sam: Self-adaptive dynamic analysis for multithreaded programs. In *Proceedings of the 7th International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'11, pages 115–129, Berlin, Heidelberg, 2012. Springer-Verlag.

[17] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. Have: Detecting atomicity violations via integrated dynamic and static analysis. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2009.

[18] L. Chew and D. Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 307–320, New York, NY, USA, 2010. ACM.

[19] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 210–220, New York, NY, USA, 2002. ACM.

[20] K. E. Coons, S. Burckhardt, and M. Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 15–24, New York, NY, USA, 2010. ACM.

[21] J. Corbett, M. Dwyer, and J. Hatcliff. Bandera: a source-level interface for model checking java programs. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 762–765, 2000.

[22] J. Coyle, I. Roy, M. Kraeva, and G. Luecke. Upc-check: a scalable tool for detecting run-time errors in unified parallel c. *Computer Science - Research and Development*, 28(2-3):203–209, 2013.

[23] J. Dingel and H. Liang. Automating comprehensive safety analysis of concurrent programs using verisoft and txl. In *In Proceedings of the International Symposium on Foundations of Software Engineering ACM SIGSOFT 2004/FSE12*, 2004.

[24] F. E. Eassa, L. J. Osterweil, and M. Z. Abdel Mageed. Aida: A dynamic analyzer for ada programs. *J. Syst. Softw.*, 31(3):239–255, Dec. 1995.

[25] T. Elmas and S. Tasiran. Vyrdmc: Driving runtime refinement checking with model checkers. *Electr. Notes Theor. Comput. Sci.*, 144(4):41–56, 2006.

[26] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *ESEC/FSE 2013*, pages 37–47, New York, NY, USA, 2013. ACM.

[27] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI '00*, pages 219–232, New York, NY, USA, 2000.

[28] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *EuroSys '11*, pages 215–228, New York, NY, USA, 2011.

[29] M. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan. Best: A symbolic testing tool for predicting multi-threaded program failures. In *ASE 2011*, pages 596–599, 2011.

[30] M. Gligoric, V. Jagannath, and D. Marinov. Mutmut: Efficient exploration for mutation testing of multithreaded code. In *ICST '10*, pages 55–64, Washington, DC, USA, 2010.

[31] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In *ISSTA 2013*, pages 224–234, New York, NY, USA, 2013. ACM.

[32] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison Wesley, 2^o edition, January 2003.

[33] E. L. Gunter and D. Peled. Path exploration tool. In *TACAS '99*, pages 405–419, London, UK, UK, 1999.

[34] D. Harmanci, P. Felber, V. Gramoli, and C. Fetzer. C.: TMunit: Testing software transactional memories. In *TRANSACT 2009*, 2009.

[35] A. C. Hausen, S. R. Vergilio, S. Souza, P. Souza, and A. Simao. Valimpi: Uma ferramenta para teste de programas paralelos. In *XX SBES*, pages 1–6, Florianopolis, SC, 2006.

[36] K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space-craft controller using spin. *IEEE Trans. Softw. Eng.*, 27(8):749–765, Aug. 2001.

[37] K. Havelund and G. Rosu. Monitoring java programs with java pathexplorer. Technical report, 2001.

[38] A. Humphrey, C. Derrick, G. Gopalakrishnan, and B. Tibbitts. Gem: Graphical explorer of mpi programs. In *ICPPW 2010*, pages 161–168, Sept 2010.

[39] A. Ibing. Path-sensitive race detection with partial order reduced symbolic execution. In *Software Engineering and Formal Methods*, volume 8938 of *Lecture Notes in Computer Science*, pages 311–322. 2015.

[40] E. Itoh, Z. Furukawa, and K. Ushijima. A prototype of a concurrent behavior monitoring tool for testing of concurrent programs. In *APSEC 1996*, pages 345–354, Dec 1996.

[41] E. M. J. Jenny Li and D. M. Weiss. *Software Testing: Tools*, pages 1178–1187. In: Encyclopedia of Software Engineering Two-Volume Set (Print). Auerbach Publications, 2010.

[42] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In T. Gyimothy and A. Zeller, editors, *SIGSOFT FSE*, pages 223–233, 2011.

[43] A. Jannesari and F. Wolf. Automatic generation of unit tests for correlated variables in parallel programs. *International Journal of Parallel Programming*, pages 1–19, 2015.

[44] K. Kahkonen and K. Heljanko. Testing multithreaded programs with contextual unfoldings and dynamic symbolic execution. In *ACSD 2014*, pages 142–151, June 2014.

[45] K. Kähkönen, O. Saarikivi, and K. Heljanko. Lct: A parallel distributed testing tool for multithreaded java programs. *Electronic Notes in Theoretical Computer Science*, 296:253 – 259, 2013.

[46] A. Kamil and K. Yelick. Enforcing textual alignment of collectives using dynamic checks. In *LCPC'09*, pages 368–382, Berlin, Heidelberg, 2010.

[47] T. Katayama, Z. Furukawa, and K. Ushijima. Design and implementation of test-case generation for concurrent programs. In *Software Engineering Conference, 1998. Proceedings. 1998 Asia Pacific*, pages 262–269, Dec 1998.

[48] K. M. Kavi, A. Moshtaghi, and D.-J. Chen. Modeling multithreaded applications using petri nets. *Int. J. Parallel Program.*, 30(5):353–371, Oct. 2002.

[49] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W. C. Chu, N.-L. Hsueh, and C.-T. Yang. Automatic testing environment for multi-core embedded software—atemes. *Journal of Systems and Software*, 85(1):43 – 60, 2012.

[50] B. Krammer, M. S. Müller, and M. M. Resch. Mpi application development using the analysis tool marmot. In *ICCS 2004*, volume 3038 of *Lecture Notes in Computer Science*, pages 464–471, 2004.

[51] H. Krawczyk and B. Wiszniewski. Systematic testing of parallel programs. Technical report, Massachusetts Institute of Technology, 1999.

[52] H. Krawczyk, B. Wiszniewski, P. Kuzora, M. Neyman, and J. Proficz. Integrated static and dynamic analysis of pvm programs with steps. *Computers and Artificial Intelligence*, 17(5), 1998.

[53] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD '07*, pages 54–64, New York, NY, USA, 2007.

[54] M. Kusano and C. Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In *ASE*, pages 722–725, 2013.

[55] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A platform for search-based testing of concurrent software. In *8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 48–58, 2010.

[56] E. Larson and R. Palting. Mdat: A multithreading debugging and testing tool. In *SIGCSE '13*, pages 403–408, New York, NY, USA, 2013.

[57] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Software Eng.*, 32(6):382–403, 2006.

[58] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32(6):382–403, June 2006.

[59] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *SOSP '11*, pages 327–336, New York, NY, USA, 2011.

[60] B. Long, D. Hoffman, and P. Strooper. Tool support for testing concurrent java components. *Software Engineering, IEEE Transactions on*, 29(6):555–566, June 2003.

[61] J. Lourenço, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In *EUROMICRO 97*, page 291, Budapest, Hungary, 1997.

[62] K. Lu, X. Zhou, T. Bergan, and X. Wang. Efficient deterministic multithreading without global barriers. In *PPoPP '14*, pages 287–300, New York, NY, USA, 2014.

[63] L. Lu, W. Ji, and M. L. Scott. Dynamic enforcement of determinism in a parallel scripting language. In *PLDI '14*, pages 519–529, New York, NY, USA, 2014.

[64] Q. Luo, S. Zhang, J. Zhao, and M. Hu. A lightweight and portable approach to making concurrent failures reproducible. In *FASE'10*, pages 323–337, Berlin, Heidelberg, 2010.

[65] G. Maheswara, J. S. Bradbury, and C. Collins. Tie: An interactive visualization of thread interleavings. In *SOFTVIS '10*, pages 215–216, New York, NY, USA, 2010.

[66] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. *SIGPLAN Not.*, 49(6):316–325, June 2014.

[67] E. R. B. Marques, F. Martins, and M. Simões. Cooperari: A tool for cooperative testing of multithreaded java programs. In *PPPJ '14*, pages 200–206, New York, NY, USA, 2014. ACM.

[68] J. Mellor-Crummey. Compile-time support for efficient data race detection in shared-memory parallel programs. In *PADD '93*, pages 129–139, New York, NY, USA, 1993.

[69] H. Mushtaq, Z. Al-Ars, and K. Bertels. Efficent and highly portable deterministic multithreading (detlock). *Computing*, 96(12):1131–1147, 2014.

[70] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI'08*, pages 267–280, Berkeley, CA, USA, 2008.

[71] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, 2 edition, 2004.

[72] A. Nistor, D. Marinov, and J. Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, pages 541–552, 2009.

[73] A. Nistor, D. Marinov, and J. Torrellas. Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing. In *MICRO-43 2010*, pages 251–262, Dec 2010.

[74] B. Norris and B. Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. *SIGPLAN Not.*, 48(10):131–150, Oct. 2013.

[75] M. Oberhuber, S. Rathmayer, and A. Bode. Tuning parallel programs with computational steering and controlled execution. In *HICSS 1998*, volume 7, pages 157–166 vol.7, Jan 1998.

[76] A. Offenwanger and Y. Lucet. Conee: An exhaustive testing tool to support learning concurrent programming synchronization challenges. In *WCCCE '14*, pages 11:1–11:6, New York, NY, USA, 2014.

[77] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108, Mar. 2009.

[78] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park. Mpirace-check: Detection of message races in mpi programs. In *GPC'07*, pages 322–333, 2007.

[79] S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS XIV*, pages 25–36, New York, NY, USA, 2009.

[80] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic mapping studies in software engineering. In *EASE'08*, pages 68–77, 2008.

[81] E. Pozniansky and A. Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, Mar. 2007.

[82] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: Symbolic execution of java bytecode. In *ASE '10*, pages 179–180, New York, NY, USA, 2010.

[83] Z. Rakamaric. Storm: static unit checking of concurrent programs. In *32nd International Conference on Software Engineering, 2010 ACM/IEEE*, volume 2, pages 519–520, May 2010.

[84] K. Ravichandran, A. Gavrilovska, and S. Pande. Destm: Harnessing determinism in stms for application development. In *PACT '14*, pages 213–224, New York, NY, USA, 2014.

[85] M. Ricken and R. Cartwright. Concjunit: unit testing for concurrent programs. In B. Stephenson and C. W. Probst, editors, *PPPJ*, pages 129–132. ACM, 2009.

[86] C. Sadowski and J. Yi. Tiddle: A trace description language for generating concurrent benchmarks to test dynamic analyses. In *WODA 2009*, 2009.

[87] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. *SIGPLAN Not.*, 49(10):473–489, Oct. 2014.

[88] F. S. Sarmanho. Teste de programas concorrentes com memória compartilhada. Master's thesis, ICMC/USP, São Carlos, SP, 2009.

[89] F. S. Sarmanho, P. S. L. Souza, S. R. S. Souza, and A. S. S. ao. Structural testing for semaphore-based multithread programs. In *ICCS (1)*, pages 337–346, 2008.

[90] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05*, pages 83–94, New York, NY, USA, 2005. ACM.

[91] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *ESEC/FSE '09*, pages 297–298, New York, NY, USA, 2009. ACM.

[92] K. Sen. Effective random testing of concurrent programs. In *ASE '07*, pages 323–332, New York, NY, USA, 2007.

[93] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 08, pages 11–21, New York, NY, USA, 2008. ACM.

[94] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *In CAV*, pages 419–423. Springer, 2006.

[95] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In E. Bin, A. Ziv, and S. Ur, editors, *Haifa Verification Conference*, pages 166–182, 2006.

[96] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *WBIA '09*, pages 62–71, New York, NY, USA, 2009. ACM.

[97] M. Shafique and Y. Labiche. A systematic review of state-based test tools. *International Journal on Software Tools for Technology Transfer*, 17(1):59–76, 2015.

[98] T. Sheng, N. Vachharajani, S. Eranian, and R. Hundt. Racez: A lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 401–410. ACM, 2011.

[99] S. F. Siegel. Verifying parallel programs with mpi-spin. In F. Cappello, T. Hérault, and J. Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007.

[100] R. A. Silva, S. d. R. S. de Souza, and P. S. L. de Souza. Mutation operators for concurrent programs in mpi. In *Test Workshop (LATW), 2012 13th Latin American*, pages 1–6, April 2012.

[101] V. Smiljkovic, S. Stipic, C. Fetzer, O. Unsal, A. Cristal, and M. Valero. Detrans: Deterministic and parallel execution of transactions. In *26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2014*, SBAC-PAD 2014, pages 152–159, Oct 2014.

[102] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM.

[103] P. S. L. Souza, E. Sawabe, A. S. Simao, S. R. Vergilio, and S. R. S. Souza. ValiPVM - a graphical tool for structural testing of PVM programs. In A. Lastovetsky, T. Kechadi, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science (LNCS)*, pages 257–264. Springer Berlin, Heidelberg, 2008.

[104] P. S. L. Souza, S. R. S. Souza, M. G. Rocha, R. R. Prado, and R. N. Batista. Data flow testing in concurrent programs with message-passing and shared-memory paradigms. In *ICCS*, pages 149–158, 2013.

[105] S. R. S. Souza, S. R. Vergilio, P. S. L. Souza, A. S. Simão, T. B. Goncalves, A. M. Lima, and A. C. Hausen. Valipar: A testing tool for message-passing parallel programs. In *SEKE*, pages 386–391, 2005.

[106] J. Staunton and J. A. Clark. Applications of model reuse when using estimation of distribution algorithms to test concurrent software. In *SSBSE'11*, pages 97–111, Berlin, Heidelberg, 2011.

[107] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Second Workshop on Runtime Verification (RV)*, volume 70(4), July 2002.

[108] R. Tan, P. Nagpal, and S. Miller. Automated black box testing tool for a parallel programming library. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 307–316, April 2009.

[109] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234, 2012.

[110] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 15–28, New York, NY, USA, 2014. ACM.

[111] J. Turpie, E. Reisner, J. S. Foster, and M. Hicks. Multiotter: Multiprocess symbolic execution. Technical report, 2011.

[112] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[113] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 221–230, New York, NY, USA, 2011. ACM.

[114] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 281–294, Berkeley, CA, USA, 2008. USENIX Association.

[115] S. West, S. Nanz, and B. Meyer. Demonic testing of concurrent programs. In T. Aoki and K. T. 0001, editors, *ICFEM*, volume 7635 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2012.

[116] B. P. Wood, A. Sampson, L. Ceze, and D. Grossman. Composable specifications for structured shared-memory communication. In *OOPSLA*, pages 140–159. ACM, 2010.

[117] Z. Wu, K. Lu, X. Wang, and X. Zhou. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming*, 43(2):260–285, 2015.

[118] C. S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. *ACM SIGSOFT Software Engineering Notes*, 23:153–162, March 1998.

[119] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, 2008.

[120] M. Young, R. N. Taylor, D. L. Levine, K. A. Nies, and D. Brodbeck. A concurrency analysis tool suite for ada programs: Rationale, design, and preliminary experience. *ACM Trans. Softw. Eng. Methodol.*, 4(1):65–106, Jan. 1995.

[121] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. *ACM SIGPLAN Notices*, 47(10):485–502, Oct. 2012.

[122] X. Yuan, Z. Wang, C. Wu, P.-C. Yew, W. Wang, J. Li, and D. Xu. Synchronization identification through on-the-fly test. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 4–15, Berlin, Heidelberg, 2013. Springer-Verlag.

[123] K. Zhai, B. Xu, W. K. Chan, and T. H. Tse. Carisma: A context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In *ISSTA 2012*, pages 221–231, New York, NY, USA, 2012.

[124] X. Zhou, K. Lu, X. Wang, and X. Li. Exploiting parallelism in deterministic shared memory multiprocessing. *Journal of Parallel and Distributed Computing*, 72(5):716–727, May 2012.

Bayesian Concepts in Software Testing: An Initial Review

Daniel Rodriguez
Dept. of Comp. Science
University of Alcalá
Alcalá de Henares, 28871,
Madrid, Spain
daniel.rodriguezg@uah.es

Javier Dolado
Univ. Basque Country
UPV/EHU
Donostia-San Sebastián,
20080, Spain
javier.dolado@ehu.eus

Javier Tuya
Dept. of Comp. Science
University of Oviedo
Campus of Gijón, 33204,
Gijón, Spain
tuya@uniovi.es

ABSTRACT

This work summarizes the main topics that have been researched in the area of software testing under the umbrella of "Bayesian approaches" since 2010. There is a growing trend on the use of the so-called Bayesian statistics and Bayesian concepts in general and software testing in particular. Following a Systematic Literature Review protocol using the main digital libraries and repositories, we selected around 40 references applying Bayesian approaches in the field of software testing since 2010. Those references summarise the current state of the art and foster better focused research.

So far, the main observed use of the Bayesian concepts in the software testing field is through the application of Bayesian networks for software reliability and defect prediction (the latter is mainly based on static software metrics and Bayesian classifiers). Other areas of application are software estimation and test data generation. There are areas not fully explored beyond the basic Bayesian approaches, such as influence diagrams and dynamic networks.

Categories and Subject Descriptors

A.1 [**Introductory and Survey**]; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Theory

Keywords

Bayesian statistics, probabilistic graphical models, Bayesian networks, software testing

1. BAYESIAN CONCEPTS FOR SOFTWARE TESTING: AN INITIAL REVIEW

There have been several articles in the past advocating the use of Bayesian methods in software testing. The position paper by Namin and Sridharan [27] stated that Bayesian reasoning methods have the capability of improving the field of software testing by providing solutions based on probabilistic methods. However, Namin and Sridharan discuss three obstacles faced when applying Bayesian networks: (i) the generalization of the conclusions, (ii) the sensitivity to the prior probabilities and (iii) the difficulties for software engineers to grasp the statistical concepts underlying the Bayesian approach. The first two obstacles will be further discussed in Section 4 (Discussion) after reviewing the literature. With respect to the last obstacle, the hurdles of understanding the Bayesian concepts, the Bayesian approach takes a different viewpoint in the concept of probability from the frequentist approach [37], which could make it hard to understand:

- *Frequentists*: the definition of probability is related to the frequency of an event. The parameters of interest are fixed but the data are a repeatable random sample, hence there is a frequency. No prior information is used. In a strict frequentist view, it does not make sense to talk about the true value of the parameter θ under study. The true value of θ is fixed, by definition.
- *Bayesians*: the definition of probability is related to the level of knowledge about an event. The value of knowledge about an event is based on prior information and the available data. The parameters of interest are unknown and the data are fixed. From a Bayesian viewpoint we can talk about the probability that the true value of the parameter θ lies in an interval.

The fact that Bayesians use prior information about θ makes the statistical reasoning different. Given a set of data observations represented by D, we can compute $P(D|\theta)$ (θ is fixed) in the frequentist approach, but we can compute $P(\theta|D)$ in the Bayesian approach. $P(\theta|D)$ is computed using "Bayes' theorem":

$$\Pr(\theta|D) = \frac{\Pr(D|\theta)\Pr(\theta)}{\Pr(D)}. \quad (1)$$

Equation (1) contains the elements of the Bayesian inference process: $Pr(\theta|D)$ is the posterior probability, $\Pr(D|\theta)$ is the likelihood, $Pr(\theta)$ is the prior probability and $\Pr(D) = \Pr(D|\theta)\Pr(\theta)+\Pr(D|\neg\theta)\Pr(\neg\theta)$ is a normalizer factor. Thus, equation 1 allows us in this case to compute the probability of θ given D.

Bayes' theorem enables the computation of the posterior probabilities for a variable. A Bayesian Network (BN) is

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804329

a probabilistic graphical model with variables linked by directed arcs. A BN provides a way of modelling a domain problem using graph and probability theory, where the network representation of the problem can be used to generate information about some variable, provided that the information of its parents is available. The joint probability distribution for variables $X_1, X_2, \ldots, X_n$ can be calculated as follows:

$$P(X_1, X_2, \ldots, x_n) = \prod_{i=1..n} P\left(X_i | Parents(X_i)\right) \qquad (2)$$

where $Parents(X_i)$ denote the specific values of the nodes in the graph that are linked towards the node X_i in the BN. BNs are the main use of the Bayes approach. BNs are also known as Bayesian Belief Networks or Belief Networks, which are Probabilistic Graphical Models. There are other formalisms related to the basic Bayesian approach such as Dynamic Belief Networks, Influence Diagrams and Hidden Markov Models.

2. SYSTEMATIC LITERATURE REVIEW OF BAYESIAN NETWORKS IN SOFTWARE TESTING SINCE 2010

Next, we review relevant literature related to software testing (including quality) and Bayesian concepts according to the Systematic Literature Review (SLR) protocol and guidelines suggested by Kitchenham et al. [19] and the EBSE website[1] [12].

2.1 Background and Research Aim

Recent works reviewing Bayesian networks (BN) include a position paper by Namin [27] and a review by Misirli and Basar [26] which covers several issues of decision making including defect prediction. Misirli and Basar list seven articles in "software testing" using the Bayesian approach and further 54 works under the topic of "software quality". The authors included under "software quality" the concepts of fault, failures, defect prediction and software reliability. However, there is an increasing number of papers applying Bayesian concepts in general and in testing in particular. Therefore, in this initial review our research aim is to *identify, classify and analyse the available literature since 2010 related to different aspects of software testing and quality that apply Bayesian concepts.*

2.2 Data Sources

We have used the following digital libraries and repositories as data sources:

1. ISI Web of Science (`http://apps.webofknowledge.com/`)
2. Scopus (`http://www.scopus.com/`)
3. Elsevier Science Direct (`http://www.sciencedirect.com/`)
4. IEEE Xplore (`http://ieeexplore.ieee.org/`)
5. SpringerLink (`http://www.springerlink.com/`)
6. ACM Digital Library (`http://www.acm.org/`)
7. Wiley Interscience (`http://onlinelibrary.wiley.com/`)
8. Google Scholar (`http://scholar.google.com/`)
9. The Collection of Computer Science Bibliographies (`http://liinwww.ira.uka.de/bibliography/`) Later discarted due to a large and irrelevant number of papers returned.

These generic digital libraries and repositories, together, reference all relevant publications in the software engineering field. Some of them are digital libraries, but some others are *meta-repositories*, including most relevant journals, conference and workshop proceedings.

2.3 Search Strategy

The search strategy was conducted with the following keywords in the query: "Bayesian" & "networks" & "software testing". We considered that those keywords were enough to cover all articles of interest. We tested other combinations of potential keywords but the results did not conform to the research aim. For example, we did not find relevant literature using the terms *probabilistic graphical models* but not containing *Bayes* or *networks* at the same time. Similarly, all papers about testing are almost certain to contain the substring "software testing". Finally, we followed some references from selected papers confirming that all relevant literature was found.

Table 1 shows the digital libraries used, the search domain within the repository (when possible) and the number of papers found. We merged all the references found and decide whether the paper was eligible for this survey mainly based on the title and abstract with the exception of Google scholar due to the large number of references found. Google Scholar was mainly used for checking that no important and relevant paper was missing. If a paper had an apparently relevant title or abstract, its full content was also checked to decided whether the study was to be included in the final selection.

2.4 Selection Criteria

The selection criteria consists of inclusion and exclusion criteria for the papers found. We include papers published since 2010, related to software testing, written in English and accessible on the Web. We decided to cover in this review the literature after the position paper of Namin and Sridharan [27]. There is a growing number of articles including the topics of software testing and Bayesian methods. Also, this criterion limits the number of papers to the most recent research and limits the length of this conference paper. A more comprehensive SLR is part of our current work.

The exclusion criteria include papers published before 2010. We also exclude papers related to the application of Bayesian concepts as part of some kind of optimisation in relation to other methods (e.g. forming part of neural networks). Also, some documents and articles that we have excluded, mentioned BNs without dealing with the technique. There were some articles that developed different tests based on BNs, but not strictly related to software testing, e.g., they developed "software safety tests". We leave out of review the use of neural networks that use some kind of Bayesian optimisation and other articles that are not fully focused on Bayesian

[1]Evidence-Based Software Engineering
`http://www.dur.ac.uk/ebse/`

Table 1: Repositories and papers found and selected

Digital Library	*Domain*	*Returned*	*Relevant*
ISI Web of Science	Computer Science, Engineering	10	6
Scopus	Computer Science	45	16
Elsevier ScienceDirect	Computer Science	40	6
SpringerLink	-	133	10
IEEExplore	-	24	3
ACM DL	-	12	-
Wiley Interscience	-	62	-
Google Scholar	-	2,390	-
Total			41

procedures. For example, and interesting work by Wiper et al. [40] use Bayesian concepts for providing the prior distribution probabilities to an artificial neural network. Strictly speaking, this work uses Bayesian concepts but does not use Bayesian networks.

In the next Section we analyse the selected literature grouping the different subareas and discussing possible research paths.

3. RESEARCH TOPICS ADDRESSED

We have organised the research topics into categories classified by common subdisciplines in software testing: effort testing prediction, software reliability and fault prediction, quality models, test data generation, GUI testing and a less known but interesting subarea named philosophy of technology.

3.1 Software Testing Effort Prediction and Productivity Estimates

This topic is concerned with the estimation of the test costs in terms of person-hours or person-days. Few works have recently applied Bayesian models for testing effort estimation, exceptions include the works by Torkar et al. [36], Schulz et al. [33] and Dalmazo et al. [8] which describe a defect correction effort model. A generic survey about BNs in effort estimation, including the test phase, was carried out by Radlinski [30].

3.2 Fault and Defect Prediction. Software Reliability

The topic of reliability is another area where Bayesian approaches have been explored by multiple researchers, specially for real-time systems. "Software Reliability" is the probability that software will work without failing in a specified environment for a given amount of time. Software reliability testing tries to discover as many defects as possible as early as possible.

Defect prediction from static measures from private or open repositories such as Tera-Promise[2] (formerly known as the Promise repository) have been reported on multiple studies. Bayesian classifiers have been widely used. Recent examples include the work by Dejaeger et al. [9] who compared 15 BN classifiers for the task of identifying software faulty modules. Weyuker et al. [39] also performed a comparison of tools for fault prediction that included Bayesian additive regression trees. Another comparison of classifiers including BNs and Naïve Bayes is described in Dhankhar et al. [10]. The Naïve Bayes classifier, the simplest Bayesian approach, is extensively used before and after 2010. For example, we can cite the papers by Catal et al. [5], Ma et al. [25] and by Hewett [15] for comparing the approaches to software defect prediction.

Okutan and Yildiz [28] used Bayesian networks to explore the relationship between sets of metrics and defect proneness using datasets from the Promise repository. Other works that build Bayesian networks with predictive reliability are those of Cheng-Gang et al. [6], Kumar and Yadab [20], Abreu et al. [1], Jongsawat and Premchaiswadi [16], Li and Wang [22], Rekab et al. [31], Lv et al. [24], Blackburn and Huddell [4], Qiuying et al. [29], Jun-min et al. [17], Khan et al. [18], Li and Leung [21], Cotroneo et al. [7], Ba and Wu [3] and Zheng et al. [43]. An application of software reliability with BNs in the domain of fire control radar can be found in the work by Li et al. [23].

3.3 Quality Models

A quality model describes in a structured way the concept of quality in a software system. In this category, we found the work by Wagner [38] who considers software quality based on constructing a BN from an activity-based quality model. Schumann et al. [34] also describe a Bayesian Software Health Management system in which the reliability of a system, including software and hardware, is monitored with BNs.

3.4 Test Data Generation, Test Case Selection and Test Plan Generation

Test data generation and test case priorization are important areas within software testing. A recent work by Sagarna et al. [32] explore this path as part of search based software test data generation. The improvement of random testing has been tackled by Zhou et al. [44, 45]. Sridharan and Namin reported [35] on the priorization of mutation operators. Several experiments were carried out by Do et al. [11] concerning the priorization of test cases. The authors used BNs as one of the methods for ordering the test suites. Fang and Sun [13] proposed a strategy to optimize the re-execution of test cases (regression testing) based on BNs. Finally, Han [14] built a BN by converting a Fault Tree structure of events in order to perform forward and backward reasoning.

3.5 Graphical User Interface (GUI) Testing

Another area of recent application of BNs is GUI testing. Yang et al. [41, 42] built a BN that uses the prior knowledge of testers and the BN updates the values depending on the

[2]`http://openscience.us/repo/`

results of the test cases.

3.6 Philosophy of Technology

As an *outlier* paper, we were positively surprised by the recent work by Angius [2] where the Bayes concepts and the software testing field have been used as the substrate for defining the software engineering area as a "scientifically attested technology". This paves the way for more studies relating the disciplines of software testing and the philosophy of technologies.

4. DISCUSSION

The main use of the Bayesian concepts in software testing lie on the "software reliability" area, with 60% of the publications falling in this category. Other topics of applications are "test data generation" and "test effort estimation", with 11% and 10% of the references, respectively. Topics where BNs are not so extensively used were "quality models", "GUI testing" and "philosophy of technology".

Although there is an increasing number of works applying BN approaches, there are issues that hinder their application as previously mentioned such as their steep learning curve and problems related to the statistical analyses. With respect to these two problems (also previously mentioned in the introduction) and discussed by Namin and Sridharan [27], we may highlight the following issues, after reviewing the literature:

- *Generalization of the conclusions*: every work builds its BN starting from scratch and the BN is adapted to its specific problem. A "meta study" or meta-analysis of the results obtained by different researchers would uncover potential similarities in the results and in the graphical structure of the BN.
- *Sensitivity to priors*: an essential characteristic of BNs is the need to provide prior probabilities to variables. One way to avoid discrepancies is to set standard priors in the field, which could be agreed upon in case of parameters such as productivity, etc. However, it is not always possible to agree on priors nor the non-informative priors are adequate to the BN model. Other alternatives could include the use of hyperprior distributions. The fact that BNs allow us to update the variable probabilities can moderate the results obtained with different priors, provided a robust BN.

Probabilistic graphical models can help in testing activities (and decision making in general) as supervised (prediction) and unsupervised (clustering) techniques from the data mining point of view as well as optimisation approaches. In prediction, we can consider classifiers such as Naïve Bayes and more complex structures such as TAN (Tree Augmented Naïve Bayes) to generic networks such as Bayesian Networks or Markov Models and their extensions (e.g. Dynamic BNs, Influence Diagrams). These latter Bayesian approches have not yet been fully exploited (in comparison with the former simpler Bayesian classifiers).

In the case of graphical models for optimisation, Evolutionary Distribution Algorithms have been applied successfully in software testing (although mainly prior to 2010). These approaches have also been applied to data generation which is considered to be a preprocessing step in data mining and, in our opinion, they can be further explored.

5. CONCLUSIONS

In this work, we reviewed the recent literature on probabilistic graphical models in software testing. We found around 40 references dealing with the topics of interest since 2010. The spread of topics found within the software testing area applying BNs is fairly limited. We classified the references into six categories. The fact that the main category is related to "software reliability" may distort the potential applications of BNs to other areas in software testing. Interestingly, there was a reference that positioned the concept of "software testing" in the center of study of software engineering as a science.

As our current work, we are extending this systematic literature survey.

6. ACKNOWLEDGMENTS

The authors are grateful to the reviewers and chairs for their helpful comments. Partial support has been received by Project Iceberg FP7-People-2012-IAPP-324356 (D. Rodriguez) and Project TIN2013-46928-C3.

7. REFERENCES

[1] R. Abreu, A. Gonzalez-Sanchez, and A. J. Van Gemund. A diagnostic reasoning approach to defect prediction. In *Modern Approaches in Applied Intelligence*, pages 416–425. Springer, 2011.

[2] N. Angius. The problem of justification of empirical hypotheses in software testing. *Philosophy & Technology*, 27(3):423–439, 2014.

[3] J. Ba and S. Wu. Propred: A probabilistic model for the prediction of residual defects. In *Mechatronics and Embedded Systems and Applications (MESA), 2012 IEEE/ASME International Conference on*, pages 247–251. IEEE, 2012.

[4] M. Blackburn and B. Huddell. Hybrid bayesian network models for predicting software reliability. In *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, 2012.

[5] C. Catal, U. Sevim, and B. Diri. Practical development of an eclipse-based software fault prediction tool using naive bayes algorithm. *Expert Systems with Applications*, 38(3):2347 – 2353, 2011.

[6] B. Cheng-Gang, J. Chang-Hai, and C. Kai-Yuan. A reliability improvement predictive approach to software testing with bayesian method. In *Control Conference (CCC), 2010 29th Chinese*, pages 6031–6036, July 2010.

[7] D. Cotroneo, R. Natella, and R. Pietrantuono. Predicting aging-related bugs using software complexity metrics. *Performance Evaluation*, 70(3):163 – 178, 2013. Special Issue on Software Aging and Rejuvenation.

[8] B. L. Dalmazo, A. L. R. de Sousa, W. L. Cordeiro, J. Wickboldt, R. C. Lunardi, R. L. dos Santos, L. P. Gaspary, L. Z. Granville, C. Bartolini, M. Hickey, et al. It project variables in the balance: a bayesian approach to prediction of support costs. In *Software Engineering (SBES), 2011 25th Brazilian Symposium on*, pages 224–232. IEEE, 2011.

[9] K. Dejaeger, T. Verbraken, and B. b. Baesens. Toward comprehensible software fault prediction models using

bayesian network classifiers. *IEEE Transactions on Software Engineering*, 39(2):237–257, 2013.

[10] S. Dhankhar, H. Rastogi, and M. Kakkar. Software fault prediction performance in software engineering. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 228–232, March 2015.

[11] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *IEEE Transactions on Software Engineering*, 36(5):593–617, 2010.

[12] EBSE. Template for a systematic literature review protocol. `http://www.dur.ac.uk/ebse/resources/templates/SLRTemplate.pdf`, 2010.

[13] Z. Fang and H. Sun. A software regression testing strategy based on bayesian network. In *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pages 1–4. IEEE, 2010.

[14] L. Han. Evaluation of software testing process based on bayesian networks. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 7, pages V7–361. IEEE, 2010.

[15] R. Hewett. Mining software defect data to support software testing management. *Applied Intelligence*, 34(2):245–257, 2011.

[16] N. Jongsawat and W. Premchaiswadi. Developing a bayesian network model based on a state and transition model for software defect detection. pages 295–300, 2012.

[17] Y. Jun-min, C. Ying-xiang, C. Jing-ru, and F. Jia-jie. Optimization model of software fault detection. In *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*, pages 129–136. Springer, 2013.

[18] G. Khan, S. Sengupta, and K. Das. A probabilistic model for analysis and fault detection in the software system: An empirical approach. *Lecture Notes in Electrical Engineering*, 298 LNEE:253–265, 2014.

[19] B. A. Kitchenham, T. Dyba, and M. Jørgensen. Evidence-based software engineering. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 273–281, Washington, DC, USA, 2004. IEEE Computer Society.

[20] C. Kumar and D. Yadav. Software defects estimation using metrics of early phases of software development life cycle. *International Journal of System Assurance Engineering and Management*, pages 1–9, 2014.

[21] L. b. Li and H. Leung. Bayesian prediction of fault-proneness of agile-developed object-oriented system. *Lecture Notes in Business Information Processing*, 190:209–225, 2014.

[22] Q. Li and J. Wang. Determination of software reliability demonstration testing effort based on importance sampling and prior information. *Advances in Intelligent and Soft Computing*, 126 AISC:247–255, 2012.

[23] Z. Li, Q. Zhao, C. Li, and X. Yang. Design and reliability evaluation of simulation system for fire control radar network. In *International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE)*, pages 1314–1317, June 2012.

[24] J. Lv, B.-B. Yin, and K.-Y. Cai. Estimating confidence interval of software reliability with adaptive testing strategy. *Journal of Systems and Software*, 97(0):192 – 206, 2014.

[25] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, 54(3):248 – 256, 2012.

[26] A. T. Misirli and A. B. Bener. Bayesian networks for evidence-based decision-making in software engineering. *IEEE Transactions on Software Engineering*, 40(6):1–1, 2014.

[27] A. S. Namin and M. Sridharan. Position paper: Bayesian reasoning for software testing. *Proc. of the FSE/SDP workshop on Future of software engineering research (FoSER '10)*, 2010.

[28] A. Okutan and O. T. Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

[29] L. Qiuying, L. Haifeng, and W. Guodong. Sensitivity analysis on the influence factors of software reliability based on diagnosis reasoning. In *Intelligence Computation and Evolutionary Computation*, pages 557–566. Springer, 2013.

[30] L. Radlinski. A survey of bayesian net models for software development effort prediction. *International Journal of Software Engineering and Computing*, 2(2):95–109, 2010.

[31] K. Rekab, H. Thompson, and W. Wu. A multistage sequential test allocation for software reliability estimation. *IEEE Transactions on Reliability*, 62(2):424–433, 2013.

[32] R. Sagarna, A. Mendiburu, I. Inza, and J. A. Lozano. Assisting in search heuristics selection through multidimensional supervised classification: A case study on software testing. *Information Sciences*, 258(0):122 – 139, 2014.

[33] T. Schulz, L. Radlinski, T. Gorges, and W. Rosenstiel. Defect cost flow model. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, page 1, New York, New York, USA, Sept. 2010. ACM Press.

[34] J. Schumann, T. Mbaya, O. Mengshoel, K. Pipatsrisawat, A. Srivastava, A. Choi, and A. Darwiche. Software health management with Bayesian networks. *Innovations in Systems and Software Engineering*, 9(4):271–292, June 2013.

[35] M. Sridharan and A. S. Namin. Prioritizing mutation operators based on importance sampling. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 378–387. IEEE, 2010.

[36] R. Torkar, N. M. Awan, A. K. Alvi, and W. Afzal. Predicting software test effort in iterative development using a dynamic bayesian network. In *21st IEEE International Symposium on Software Reliability Engineering*. IEEE, 2010.

[37] J. VanderPlas. Frequentism and bayesianism: A python-driven primer. *arXiv preprint arXiv:1411.5018*, 2014.

[38] S. Wagner. A Bayesian network approach to assess

and predict software quality using activity-based quality models. *Information and Software Technology*, 52(11):1230–1241, Nov. 2010.

[39] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010.

[40] M. Wiper, A. Palacios, and J. Marın. Bayesian software reliability prediction using software metrics information. *Quality Technology & Quantitative Management*, 9(1):35–44, 2012.

[41] Z. Yang, Z. Yu, and C. Bai. The approach of graphical user interface testing guided by bayesian model. *Lecture Notes in Electrical Engineering*, 277 LNEE:385–393, 2014.

[42] Z.-F. Yang, Z.-X. Yu, B.-B. Yin, and C.-G. Bai. Gui reliability assessment based on bayesian network and structural profile. *International Journal of Signal Processing, Image Processing and Pattern Recognition*, 8(1):225–240, 2015.

[43] C. Zheng, F. Peng, J. Wu, and Z. Wu. Software life cycle-based defects prediction and diagnosis technique research. In *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, volume 8, pages V8–192. IEEE, 2010.

[44] B. Zhou, H. Okamura, and T. Dohi. Markov chain monte carlo random testing. In *Advances in Computer Science and Information Technology*, pages 447–456. Springer, 2010.

[45] B. Zhou, H. Okamura, and T. Dohi. Enhancing performance of random testing through markov chain monte carlo methods. *IEEE Transactions on Computers*, 62(1):186–192, 2013.

Author Index