Automated Testing of Context-Aware Applications

Ralf Tönjes[1], Eike Steffen Reetz[1,2], Marten Fischer[1], Daniel Kuemper[1]

[1]Faculty of Engineering and Computer Science, University of Applied Sciences Osnabrück
Osnabrück, Germany
Email: {r.toenjes, e.reetz, m.fischer, d.kuemper} @hs-osnabrueck.de
[2]Centre for Communication Systems Research, University of Surrey
Guildford, United Kingdom

***Abstract*—The development of context-aware applications is a difficult and error-prone task. The dynamics of the environmental context combined with the complexity of the applications poses a vast number of possibilities for mistakes during the creation of new applications. Therefore it is important to test applications before they are deployed in a life system. For this reason, this paper proposes a testing tool, which will allow for automatic generation of various test cases from application description documents. Semantic annotations are used to create specific test data for context-aware applications. A test case reduction methodology based on test case diversity investigations ensures scalability of the proposed automated testing approach.**

I. Introduction

Context-aware applications promise to adapt to the situation of an entity (person, place, physical or computational object). This requires reliable techniques for information extraction gathered from physical or virtual sensors and consecutive adaptation of the application to the current context. Context-aware applications interact with the real world and have to cope with harsh environments where the provided information may be incomplete or erroneous. This demands for sophisticated evaluation and testing of context-aware applications before deployment. This is particularly true for recommender and actor systems in a productive environment (e.g. traffic control system not working properly). However, testing of context-aware applications differs from classical application testing:

- Specific real world situations may not be available for testing and need to be emulated.
- The time constrains of real world processes hinder fast testing.
- Sensor devices are often resource constrained (e.g. battery) thus forbidding extensive testing.
- Context-aware applications controlling actors need critical testing without/before attaching the actors.

Hence, the strong interaction with the physical world needs to be addressed by the test environment. Furthermore, the scalability and time efficiency of the approach needs to be estimated realistically. Previous testbed approaches handle external resources in different ways (e.g. WISEBED [1], Kansei [2] cf. Gluhak et al. [3] for a comprehensive comparison of testbed features):

- physical resources (e.g. sensors) in a testbed
- virtual resources (e.g. emulated sensors or data bases) in a testbed
- a combination of physical and virtual resources
- gateways to generically connect physical resources

The main drawback of the first approach including physical resources is the limited scalability and ability to test specific situations (e.g. room temperature $< -20\,^{\circ}\mathrm{C}$). Improved scalability as well as larger control of test situations can be achieved with virtual resources. Most approaches build a virtual machine that represents the external resource. The target platform can then be directly deployed in this virtual machine. Tests can therefore be run from the perspective of the virtual resources as well as from the connected context-aware applications. Although, concepts combining virtual and physical resources in the testbed have a great coverage of hardware and software-related failures, they show limitations with regard to scalability. To overcome the heterogeneity of context-provisioning resources Diaz et. al [4] suggest a systematic implementation of gateways to lower the testing costs by connecting the application, i.e. System Under Test (SUT), and testing tool. As a result, complexity of interaction between the SUT and the external resources is hidden from the test framework but still needs to be modelled within the gateway. Model-based testing is a well-investigated research area whilst the allowance of the integration of resource constrained devices is not solved. Especially concerning emulation and simulation due to the avoidance of device specific costs [5].

To overcome the complexity and drawbacks of these approaches this paper proposes an automated process of test case generation based on application description documents. The main novelties of the proposed solutions are as follows:

- A new context-aware application model designed for automated test case creation and execution is presented. It reflects the differences of the context-aware application behaviour to other applications.
- The test approach enables an early and simplified testing of context-aware applications by encapsulation and separation of application and external (physical or virtual) resources based on interface emulations during SUT execution in a sandbox environment. As test specification language and for test control the Testing and Test Control Notation Version 3 (TTCN-3) [6] is employed. TTCN-3 is a European Telecommunications Standards Institute (ETSI) standard [7] that supports black-box testing of distributed systems.
- A complete automation of test case generation is provided that i) analyses machine interpretable application

descriptions to ii) generate an Application Behaviour Model (ABM) based on extended finite state machines. The ABM is exploited to iii) extract TTCN-3 test cases employing the script engine Velocity [8]. The automation ensures that context-aware applications can be tested systematically without the need for human intervention during test design and execution, thereby ensuring scalability and efficiency.
- The problem of test case explosion is addressed by computing the similarity between Test Cases (TCs). New optimisation techniques based on a Greedy algorithms are utilised to optimise the test case group selection.
- Data types are restricted based on guard dependency analysis for each test case individually. The data values are generated prior to test execution. This allows an efficient way of integrating data values into the test case diversity optimisation.

The remainder of this paper is structured as follows: Section II provides an overview of the test concept for context-aware applications. Section III describes the automated test case derivation from an application description. Section IV deals with reduction of test cases. In Section V the test cases are executed and the results are discussed. Finally Section VI concludes this paper.

II. Test Concept

A. Categorisation of Context-Aware Applications

To ease the generation of new context-aware applications the suggested approach supports service oriented architectures (SOA). SOA supports reuse of service components and composition during run-time. The service logic, i.e. workflow, can be described with Business Process Model and Notation (BPMN). Here the service employs RESTful interfaces based Get, Post, Put, and Delete request methods, whose invocation is defined in a Web Application Description Language (WADL) document [9]. The implemented service can be deployed in a run-time environment for web services.

As defined in [10], two types of context-aware services can be specified to ensure direct consumption and composition of external resources without dealing with heterogeneous interfaces:

- An Atomic Service (AS) accessing 0 - n external resources via their own individual interfaces and radio technologies. It provides access to these resources via its standardised service interface.
- A Composite Service (CS) enables a business process based composition of various AS and CS. It can provide an interface for services that do not directly connect to external resources. It employs only AS and other CS to acquire sensor information and to control actuator nodes.

Please note that the suggested approach is not limited to SOA and the following test automation can be realised with all kinds of documents which include information to derive a state machine model of the application (e.g. provides information about, Input, Output, Precondition and Effect (IOPE)). For matter of demonstration purposes, the developed testing tool is based on BPMN and WADL documents.

B. Overall Test Architecture

The architecture supports testing by specific components for test design and execution in a test environment i.e. Sandbox, that provides the context instead of the real world:

- The Test Design Engine (TDE) is responsible for storing the application behaviour model of the SUT, enabling the interaction with the test designer, analysing the application behaviour model, and for deriving executable TCs.
- The Test Execution Engine (TEE) executes the tests and ensures that the Run-Time Sandbox environment is set up according to the current test case. It provides emulated AS based on the specified tests.
- The Run-Time Sandbox environment hosts the SUT and manipulates the communication to ensure that the CS can be tested without the need for interaction with the connected ASs and their links to sensors and actuators.

C. Test Generation Process

The test case derivation and execution is divided in five steps as described in the following sections:

- The application description provides information about the application in a machine interpretable format (Section III-A).
- The application description is processed to derive an ABM (Section III-C).
- Test Cases are derived from the ABM (Section III-D).
- Test case reduction selects distinct test cases to keep testing time in limits while preserving test coverage (Section IV).
- The selected test cases are executed in a Sandbox. (Section V)

III. Test Case Derivation

A. Application Description Documents

For automation purposes documents describing the application and its behaviour are utilised. The available methods of a simple AS are described using the WADL format. The input parameters and return values are further described in referenced XML Schema Definition (XSD) documents. Here, technical limits of the datatypes used as the parameters and return values are stated. Such technical limitations include value ranges, lists of mandatory and optional paramters etc.. Knowledge about the context of a parameter can help to limit them further. Therefore a link between a parameter and an instance in an upper ontology (e.g. the Suggested Upper Merged Ontology (SUMO) ontology) is created by using the *type* attribute within the *parameter* node in the WADL document. A prominent example are temperatures. A parameter expecting a float value has an absolute minimum of 2^{-149} (in the programming language JAVA), but the knowledge that this parameter represents a temperature allows confining the minimum at -273.15 degree Celsius. The knowledge can be used to either generate meaningful test data for each parameter or to evaluate the plausibility of the output by an AS. Finally, BPMN is used to compose multiple AS for a complete application.

B. Application Behaviour Model

For the ABM, state machine concepts of states and transitions are re-used. In addition, the inclusion of concepts of TTCN-3 (e.g., Ports, Components, MessageTemplates) enables an easy model transformation. The basic model objects are shown in Figure 1 and can be described as follows:

States represent different logical conditions of the SUT and limit the number of correct functionality.

Events characterise the starting of an activity, which might result in actions or a state change. Events can be either from the type timer or input message. One or more events are attached to a transition.

Actions describe the reaction of the system to an event. An Action can be either a response message (output) or can result in a request sent to an external resource. Actions are attached to a transition.

Transitions can be divided into two types. Active transitions (those containing an action) describe how the SUT reacts to a certain event and at a specific state. Passive transitions (those containing only events) describe how a user or the test environment might interact with the SUT. Both types connect different states within the model.

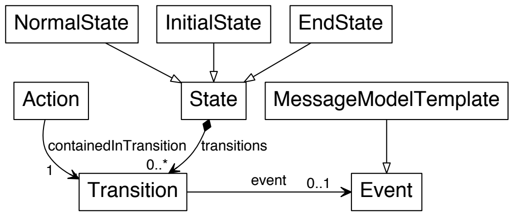


Fig. 1. Major elements of ABM.

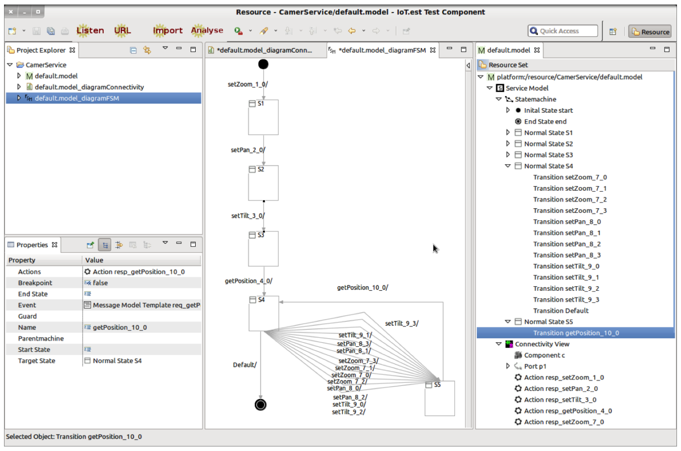
C. Application Behaviour Model Generation

The starting point for the generation of the ABM is the applications BPMN description document. The business logic is parsed and an ABM instance is build iteratively. *Invocations* and *Receptions* in BPMN are mapped to transitions and states are inserted in between. Loops are realised by transitions pointing at a previous state and conditions lead to multiple transitions leaving from a single state. All involved AS, required to execute the application, are included as *service endpoints* in the BPMN. They reference to the corresponding WADL document, describing the AS. Each WADL itself references the XSD descriptions and optionally an ontology describing the parameters and return values. This information is utilised to define the test data as *Action* or *Event*, depending on the SUT acting as the sending or receiving party, respectively. Consecutively, *Actions* and *Events* are attached to the corresponding transition.

Figure 2 shows an example ABM displayed in the editor of the developed testing tool. The test developer can view and modify the created model but no intervention is required since the model is created in a fully automated way.

D. Test Case Extraction

The ABM is used to derive test cases. In the first stage each possible path in the state machine, from a start point to an end point is considered an individual test case. A configuration of the test case generation allows transition coverage or transition and state-coverage based on the W method [11].



CamerService
- default.model
- default.model_diagramConnectivity
- default.model_diagramFSM

Properties

Property	Value
Actions	Action resp_getPosition_10_0
Breakpoint	false
End State	
Event	Message Model Template req_getP
Guard	
Name	getPosition_10_0
Parentmachine	
Start State	
Target State	Normal State S4

Selected Object: Transition getPosition_10_0

Resource Set
- platform:/resource/CamerService/default.model
  - Service Model
    - Statemachine
      - Inital State start
      - End State end
      - Normal State S1
      - Normal State S2
      - Normal State S3
      - Normal State S4
        - Transition setZoom_7_0
        - Transition setZoom_7_1
        - Transition setZoom_7_2
        - Transition setZoom_7_3
        - Transition setPan_8_0
        - Transition setPan_8_1
        - Transition setPan_8_2
        - Transition setPan_8_3
        - Transition setTilt_9_0
        - Transition setTilt_9_1
        - Transition setTilt_9_2
        - Transition setTilt_9_3
        - Transition Default
      - Normal State S5
        - Transition getPosition_10_0
    - Connectivity View
      - Component c
      - Port p1
      - Action resp_setZoom_1_0
      - Action resp_setPan_2_0
      - Action resp_setTilt_3_0
      - Action resp_getPosition_4_0
      - Action resp_setZoom_7_0

Fig. 2. ABM evaluation in developed testing tool.

The Model transformation from ABM to standardised TTCN-3 ensures explicit representation and reproducibility of test cases. During the model transformation each element is inspected and the required TTCN-3 elements are created. The actual writing of the TTCN-3 code is realised with a template engine. This enables the separation of syntactical details of the TTCN-3 language from the analysing logic thus reducing the complexity and enhancing the manageability. The followed approach uses the Java-based template engine Velocity [8]. In the following the transformation step is outlined with some detail. The model object *InitialState* is used to create the general test case structure and assures that the test case stops after a defined time by adding a timer. Afterwards, the TTCN-3 element *function* is created and added to the test case. TTCN-3 functions are utilised to separate different steps of the test execution. These reusable functions are used to represent the different states of the SUT.

The next element Transition consists of an Event that can describe that an input is received by the SUT and an Action that describes the output reaction of the SUT to this input message. Table I sketches the transformation from the model object event to a send operation and the storage of the sent values for later usage. Since the ABM is created from the service point of view the translator inverts certain expressions for the purpose of testing. In this case the event of a transition becomes a send call.

TABLE I. TTCN-3 Translation of ABM Object Event.

Action:	TTCN-3 Output:

Create send call and local variable	`template HttpRequest req_setPan_1_0 := {`
	`    postRequest := {`
	`url := "http://10.1.1.42:80/CameraService/`
	`    iot/Camera/pan/10.11.127.6/19.27", ...`
	`    } }`
	`v_PositionResponse_pan := 19.27;`
	`f_request(p1, req_setPan_1_0);`
	`v_req_setPan_1_0 := req_setPan_1_0;`

Subsequently, the action part of the transition is utilised to derive TTCN-3 code. Initially a new function for the next state is created. Afterwards the defined response of the SUT is translated into TTCN-3. Then, the TTCN-3 element *alt* is used to form the possibilities of the SUT behaviour. At first, the failure case for delayed or unexpected service responses is modelled.

After that the followed approach assumes deterministic service behaviour with only one possible valid reaction. This expected behaviour is included in the *alt* element of TTCN-3 including the jump to the next TTCN-3 function (state) created before. Table II shows the discussed transformation process of the model object action.

TABLE II. TTCN-3 TRANSLATION OF ABM OBJECT ACTION.

Action:	TTCN-3 Output:
Create Target Call	`S1_1_2();`

Create expected response message	`var template GETResponse resp_setPan_1_0 := {`
	`statusCode := (200 .. 299),`
	`content := {rawContent := omit, plainTextContent :=?},`
	`headers := ? }`

Form alt for Message	`alt {`
	`[] testcaseMaxExecutionTimer.timeout { tcMaxExecutionTimeout_1();}`
	`[] any port.receive { unexcepctedStateReached_1(); }`
	`}`

Create reply element in alt	`alt {`
	`[ischosen(req_setPan_1_0.postRequest)] p1.getreply(POSTreq: {req_setPan_1_0 .postRequest} value resp_setPan_1_0) -> value v_resp_setPan_1_0 { S1_1_2 ();}`
	`...}`

While the link to the next function has been created during the action transformation, in the last step the function itself is created at the time the next element (*NormalState*) of the Test Case (TC) is inspected. The test case is completed when the model object *EndState* is reached. This completes the TTCN-3 code creation by setting the verdict to pass. If all functions, corresponding requests and response messages have been transmitted during the TC execution this final statement indicates that the SUT has the expected behaviour for this TC.

IV. TEST CASE REDUCTION

The followed model-based testing approach enables an automated Test Case (TC) creation and execution based on machine interpretable application descriptions. However, the derivation process creates lots of TCs, which can not be executed within a reasonable time. Therefore, it is eminent to identify which TCs should be selected for execution to ensure the best target test coverage within given time and resources.

The present work follows a similarity investigation approach, which tries to identify the most diverse TCs for test execution based on a pairwise similarity between all TCs. For the followed TC reduction approach it is necessary to identify the similarity between TCs. The result of the similarity computation is a similarity score matrix which contains the pairwise similarity score between all TCs. Afterwards, the target number of TCs is selected based on the similarity score matrix. The objective is to find the group of TCs which have the lowest average similarity between the TCs of the selected group. Note, that this is a NP-hard problem [12] and the optimum can therefore not be found in polynomial time.

The similarity score is computed based on the Levenshtein algorithm. Based on the basic operations of *add, del, modify* the sequence of symbols (e.g, states, transitions) of the *FirstTestCase* is compared to the sequence of symbols of the *SecondTestCase*. It computes the number of operation steps required to alter the *FirstTestCase* for reaching the second one. The computed distance is then converted to a similarity score and normalised to the maximum number of symbols. Different to the realisation of Hematie et. al [13], counting only matches, this algorithm is able to measure the distance between two TCs as intended by the Levenshtein algorithm. Further information about the realisation can be found at [14].

An example state machine with 5 states, 10 transitions (and one transition for initiation), 2 input messages and 2 output messages is implemented and Path Finder creates 132 TCs with full state and transition coverage. The experiment has been repeated 10,000 times to ensure converging results. Fig. 3 shows the Empirical Cumulative Density Function (ECDF) of the similarity score based on the similarity score matrix, computed with the Levenshtein algorithm. If the sample quantiles are connected with a regression, the slope of the resulting function is high, i.e., between 0.1 and 0.5 (axis of abscissae). Therefore, the test case optimisation algorithm needs to handle similarity score values with small differences.

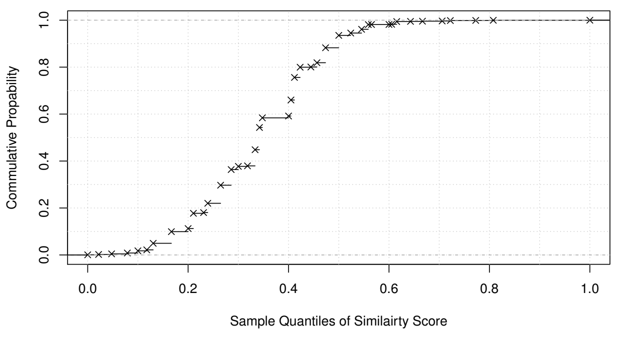


Fig. 3. Empirical cumulative density function of the Levenshtein similarity score algorithm

As the baseline, the random selection of a group of TCs has been implemented. It starts with the input of the *Similarity Scores* matrix, all *TCs*, the target number of TCs and the number of trials (*numTrials*). The output of the algorithm is the list of selected TCs for the execution. For each trial, the target number of TCs is selected out of all TCs. Afterwards, the summarized similarity score between all selected TCs is computed based on the pairwise similarity scores. For each iteration, the computed summarized similarity score is compared to the lowest previous summarized similarity score. After the defined number of trials the group of TCs with the lowest summarized similarity score is selected for the test execution. This approach converges slowly and finds the best group of TCs if the number of trials is infinite. Hence another algorithms should be favoured, that finds better groups of TCs in shorter time.

Within this paper the selection of a group of TCs is based on an extension of the Greedy algorithm. The Greedy algorithm is a heuristic problem solving approach where in each step an optimal solution is selected. It can be utilised to approximate a global optimum based on local optima [15]. For TCs reduction, the goal is to remove the worst TCs from all available TCs until the target amount of TCs is reached. As proposed by Hematie in [16], the comparison can be realised by applying a pairwise comparison between TCs. While the target number of TCs is not reached, the two most equal TCs are identified and the shorter TC is removed (assuming that longer TCs can find more failures). A drawback of the Greedy algorithm is that the decision which TCs should be removed relies on a pairwise comparison of TCs [16]. This type of Greedy algorithm does not support the goal of optimising a group of TCs. To overcome this, a new Greedy algorithm called Group Greedy is proposed. At first, the algorithm selects one TC as the first element of the target TC group. Then, for each unselected TCs the total similarity score between the selected TCs and the unselected TCs is computed. The two unselected TCs with the lowest total similarity to the selected group are identified and the longer of these two TCs is added to the target TC group. The selection process is repeated until the desired target TC group size is reached.

V. Test Case Execution

After compilation of the TTCN-3 test cases the whole test flow can be executed by a web service interface or manually using the TTworkbench [17]. It enables a visualised logging of test execution in a log report, which can be used to evaluate the detailed test results.

The goal to reduce the average similarity for a dedicated number of selected TCs is based on the assumption, that a lower average similarity results in more diverse TCs and therefore can find more failures. To verify this assumption, a generic service and the correspondent ABM is created. Afterwards, failures are inserted into the ABM and the TCs are executed. Due to failure insertion, the created service and the ABM differ and those failures can be detected during execution. Figure 4 shows the failure detection rate with different target number of TCs and compares the performance of the random selection (*RS*) algorithm with 100 trials with Group Greedy (*GGr*) selection. The results show, that *GGr* can increase the failure detection rate (doubled from 27% to 55% with 60 TCs) while requiring less computation time (not shown in the figure) compared to random search selection.

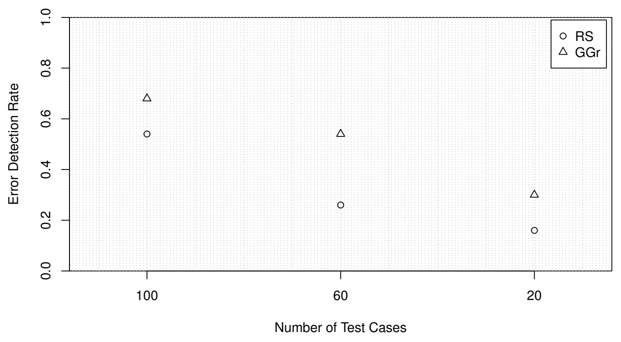


Fig. 4. Failure detection rate for test case group selection with random and group Greedy selection

VI. Conclusion

The development of reliable context-aware applications requires sophisticated testing. This paper suggests to foster testing by a semi-automated test case generation and test case execution. It addresses the particular requirements of context-aware applications that rely on external resources to provide context information. It separates the context provisioning of the real world from the application, i.e. system under test. This allows controlled and faster replay of specific situations to test the functional behaviour of context-aware applications. It was shown that test cases can automatically be extracted from an application behaviour model. Good test coverage, while keeping test time low, was achieved by selecting test cases using Group Greedy algorithm and Levenshtein similarity.

References

[1] I. Chatzigiannakis, S. Fischer, C. Koninis, G. Mylonas, and D. Pfisterer, "Wisebed: an open large-scale wireless sensor network testbed," *Sensor Applications, Experimentation, and Logistics*, pp. 68–87, 2010.

[2] A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal, "Kansei: A high-fidelity sensing testbed," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 35–47, 2006.

[3] A. Gluhak, S. Krco, M. Nati, D. Pfisterer, N. Mitton, and T. Razafindralambo, "A survey on facilities for experimental internet of things research," *Communications Magazine, IEEE*, vol. 49, no. 11, pp. 58–67, 2011.

[4] J. Díaz, A. Yague, and J. Garbajosa, "A systematic process for implementing gateways for test tools," in *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*. IEEE, 2009, pp. 58–66.

[5] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, pp. 297–312, 2012.

[6] "ETSI: ES 201 873-1 V4.6.1, Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language."

[7] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, "An introduction to the testing and test control notation (ttcn-3)," *Computer Networks*, vol. 42, no. 3, pp. 375–403, 2003.

[8] Apache Software Foundation, "The apache velocity project," Website, available online at http://velocity.apache.org/ retrieved: 2014-08-30.

[9] M. J. Hadley, "Web application description language (wadl)," Sun Microsystems, Inc., Mountain View, CA, USA, Tech. Rep., 2006.

[10] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne *et al.*, "Owl-s: Semantic markup for web services," *W3C member submission*, vol. 22, pp. 2007–04, 2004.

[11] A. Gargantini, "4 conformance testing," in *Model-Based Testing of Reactive Systems*. Springer, 2005, pp. 87–111.

[12] A. P. Mathur, *Foundations of Software Testing*. Addison-Wesley Professional, 2008.

[13] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 6, Feb. 2013.

[14] E. S. Reetz, D. Kuemper, K. Moessner, and R. Tönjes, "Investigation of opportunities for test case selection optimisation based on similarity computation and search-based minimisation algorithms," in *VALID 2014, The 6th International Conference on Advances in System Testing and Validation Lifecycle*, 2014.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 2009.

[16] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," *Department of Computer Science, Kings College London, Tech. Rep. TR-09-03*, Apr. 2009.

[17] Testing Technologies, "TTworkbench," Website, available online at http://www.testingtech.com/products/ttworkbench.php retrieved: May, 2015.