Application of Inference Rules to a Software Requirements Ontology to Generate Software Test Cases

Vladimir Tarasov[1], He Tan[1], Muhammad Ismail[1], Anders Adlemo[1], and Mats Johansson[2]

[1] School of Engineering, Jönköping University, Box 1026, 551 11 Jönköping, Sweden
{vladimir.tarasov,he.tan,muhammad.ismail,anders.adlemo}@ju.se
[2] Saab AB, Slottsgatan 40, 551 11, Jönköping, Sweden
mats.e.johansson@saabgroup.com

Abstract. Testing of a software system is resource-consuming activity. One of the promising ways to improve the efficiency of the software testing process is to use ontologies for testing. This paper presents an approach to test case generation based on the use of an ontology and inference rules. The ontology represents requirements from a software requirements specification, and additional knowledge about components of the software system under development. The inference rules describe strategies for deriving test cases from the ontology. The inference rules are constructed based on the examination of the existing test documentation and acquisition of knowledge from experienced software testers. The inference rules are implemented in Prolog and applied to the ontology that is translated from OWL functional-style syntax to Prolog syntax. The first experiments with the implementation showed that it was possible to generate test cases with the same level of detail as the existing, manually produced, test cases.

Keywords: Inference Rules, Ontology; OWL, Prolog, Requirement Specification, Test Case Generation

1 Introduction

In modern society software products and systems permeates every aspect of our lives, such as our homes, cars, the public infrastructure and even our bodies. As a consequence, quality concerns are becoming much more vital and critical as we get more dependent on these products and systems. The yearly cost of software errors as a consequence of poor quality procedures in the software industry was estimated to roughly $312 billion, according to a report in 2013 by the Cambridge University [8], and the cost still continues to increase. As detection of software errors goes hand-in-hand with testing, the same increase in cost is true for all kind of software testing activities [8, 3].

One way of curbing this ongoing trend is to automate as many as possible of the software test activities. As far as test case execution goes, this is already

a mature field where commercial products help software testers in their daily work. The automatic generation of test cases, however, is an entirely different matter.

The use of ontologies for testing has not been discussed as much as the use of ontologies in other stages of the software development process. In [6] the authors discussed possible ways of utilizing ontologies for the test case gen-eration, and the feasibility of reuse of domain knowledge encoded in ontologies for testing. In practice, however, few results have been presented in the area. Most of them have had a focus on testing web-based software and especially web services (e.g. [16, 14].

This paper proposes an approach to combine an OWL ontology with inference rules in order to construct an ontology-based application. The purpose of the application is to generate software test cases based on a software requirements specification[1]. The ontology, which represents both the software requirements and the software, hardware and communication components belonging to an embedded system, is translated from OWL functional-style syntax into Prolog syntax. The inference rules, that represent the expertise of an expert software tester, are coded in Prolog and make use of the ontology entities to generate test cases. The Prolog inference engine controls the process of selecting and invoking inference rules.

The rest of this paper is structured as follows. Section 2 details the proposed approach, including the ontology representing software requirements and software components, inference rules capturing strategies for test case generation, and the OWL to Prolog translation. Section 3 presents an evaluation of the approach. Related work is described in Section 4. Our conclusions on the result are given in Section 5.

2 Approach to Test Case Generation

When testers create test cases, they do it based on software requirements specifications and their own expertise, expertise that comes from previous work on testing software systems. To automate this process, both parts should be represented in machine-processable form. The requirements are usually described in semi-structured text documents or stored in requirements management systems. This part is captured in a requirements specification ontology, which conceptualizes the structure of software requirements and their relations to different components of a software system (Section 2.1). The second part, the testers expertise is less structured and is acquired by interviewing experienced testers and studying existing test cases with their corresponding requirements. Such knowledge is represented with inference rules that utilize the ontology for checking conditions and querying data (Section 2.3). To make it possible to use the ontology entities together with the inference rules, it is necessary to translate the ontology to a format supported by the rules (Section 2.2).

[1] The study presented in this paper is part of the project Ontology-based Software Test Case Generation (OSTAG).

2.1 Representation of Requirements with Requirements Specification Ontology

The ontology in this paper includes three pieces of knowledge: 1) a meta model of the software requirements, 2) the domain knowledge of the application, e.g. general knowledge of the hardware and software, the electronic communication standards etc., and 3) each system requirements specifications. The components that make up the domain ontology come from an embedded system within an avionic system provide by Saab Avionics. In this work the ontology is used to support test case generation. It can also be used to support other tasks in the software development process, such as requirement analysis, and requirement verification and validation.

The current version of the ontology contains 42 classes, 34 object properties, 13 datatype properties, and 147 individuals in total. Fig. 1 presents the meta model of the software requirements. As indicated in the figure, each requirement is concerned with certain functionalities of the software component. For example, a requirement may be concerned with data transfer. Each requirement consists of at least 1) requirement parameters, which are inputs of a requirement, 2) requirement conditions, and 3) results, which are usually outputs of a requirement, and exception messages. Some requirements require the system to take actions. Furthermore, there exists traceability between different requirements, e.g. traceability between an interface requirement and a system requirement.

Fig. 2 shows the ontology fragment for one particular functional requirement, SRSRS4YY-431. If the communication type is out of its valid range, the initialization service shall deactivate the UART (Universal Asynchronous Receiver/Transmitter), and return the result comTypeCfgError. In Figure 2, the rectangles represent the concepts of the ontology; the round rectangles represent the individuals; and the dashed rectangles provide the data values of datatype property for individuals.

2.2 OWL-to-Prolog Translation

To prepare the ontology for the use by the inference rules, it is necessary to translate it into the syntax that is supported by the rules. As soon as Prolog is

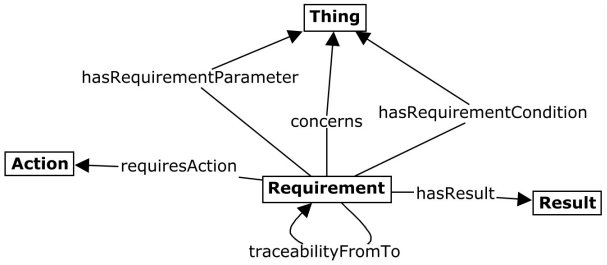

Fig. 1. The meta model of a requirement in the ontology

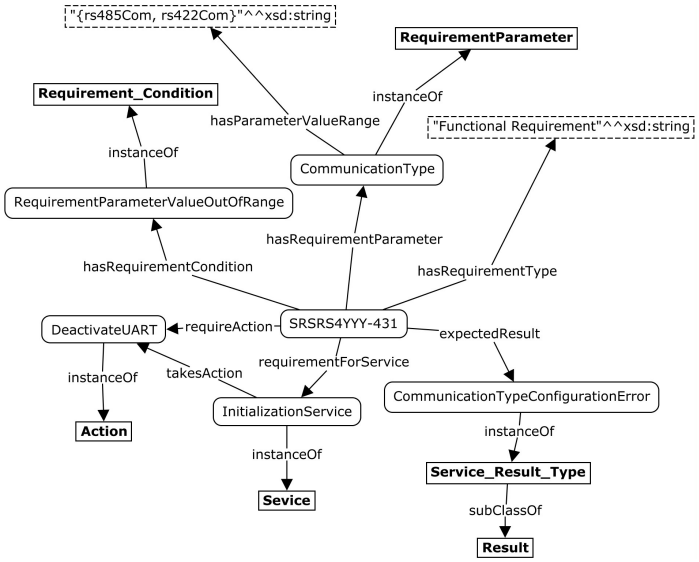

Fig. 2. Ontology fragment for the SRSRS4YY-431 requirement specification

chosen for coding inference rules (see Section 2.3), the ontology constructs have to be translated into the Prolog syntax. There exist a number of serialisation formats that can be used to save an OWL ontology to a file: RDF/XML, Turtle, OWL/XML, Manchester OWL syntax or functional-style syntax. The functional-style syntax is the closest one to the Prolog syntax. An ontology document in the functional-style syntax is a sequence of OWL constructs enclosed in the Ontology statement as well as a number of prefix definitions [11]. As a logical consequence, we have chosen functional-style syntax as the starting point for the ontology translation.

A Prolog program consists of clauses. The term clause denotes a fact or a rule in a knowledge base. A clause is ended with full stop (.) and different terms in a clause are separated with commas (,). The basic terms that are used in Prolog programs are atoms, numbers, variable and structures [2]:

- An atom is a string of characters that starts with a lower-case letter,
- A variable is a string of characters that starts with an upper-case letter,
- Integers and real numbers (floating point numbers) are also allowed in Prolog,
- Structures or complex data objects are objects with several components. Functor is used to combine several components into a single one, e.g. date(14, June, 2006).

When an ontology is written in the functional-style syntax, every single line is a separate statement that represents one construct. Each line is processed separately to translate it into the corresponding Prolog statement. A Python

Table 1. Example of translation of some OWL statements

OWL functional-style syntax	Prolog syntax
ClassAssertion(OSTAG:Error_Handling_ Requirement :SRSRS4YY-431)	classAssertion(error_handling_requirement, srsrs4yy_431).
DataPropertyAssertion(:hasParameterValueList :NumberOfStopBits "[stopBits1, stop-Bits2]"^^xsd:string)	dataPropertyAssertion(hasParameterValueList, NumberOfStopBits, [stopBits1, stopBits2]).
DataPropertyDomain(:hasParameterValueList :Requirement_Parameter)	dataPropertyDomain(hasParameterValueList, requirement_Parameter).
ObjectPropertyAssertion(OSTAG:requirementForService :SRSRS4YY-431 :InitializationService)	objectPropertyAssertion(requirementForService, srsrs4yy_431, initializationService).
SubClassOf(OSTAG:Error_Handling_ Requirement OSTAG:Requirement)	subClassOf(error_handling_ requirement, requirement).
AnnotationAssertion(rdfs:label OSTAG:FIFO "FIFO")	annotationAssertion(rdfslabel, fifo, 'FIFO').

script has been written for the OWL-to-Prolog translation, which performs these steps for every OWL statement:

- Read an OWL statement and remove OWL prefixes[2],
- Tokenize the statement and convert each token into lowerCamelCase notation because Prolog atoms start with a lower case latter.
- Convert the list of tokens into a Prolog clause in the form of a fact.

The following OWL statements are translated at the moment: ClassAssertion, subClassOf, ObjectPropertyAssertion, DataPropertyAssertion, objectPropertyRange, objectPropertyDomain, annotationAssertion. Table 1 shows several examples of translation from OWL to Prolog.

2.3 Deriving Test Cases from the Ontology with Inference Rules

To derive test cases from the ontology, it is necessary to represent testers expertise on how they use requirements to create test cases. This kind of knowledge is less structured and more difficult to capture. Few general guidelines can be found in literature, such as boundary value testing. However, most expertise is specific to particular types of software systems and/or particular domains. To capture this expertise or knowledge it is necessary to interview experienced testers and study existing test cases and their corresponding requirements. Such knowledge embodies inherent strategies for test case creation, knowledge that can be expressed in the form of heuristics represented as if-then rules.

In this study we examined 16 requirements and 20 corresponding test cases. Each requirement describes some functionality of a service (function) from a

[2] There is only one ontology used at the moment but if there are imported ontologies in the future, prefixes can be translated as well.

driver for a hardware unit, in this case represented by an embedded avionic system component. Thus, all requirements are grouped according to services. We analysed requirements covering six services. During the analysis an original test case, previously created manually by a software tester, was compared with the corresponding requirement to fully understand how different parts of the original test case had been constructed. Then, any inconsistencies or remaining doubts were cleared during discussions with the industry software testers participating in the study.

The outcome from these activities was a set of inference rules formulated in plain English. Each original test case consists of four parts: prerequisite conditions, test inputs, test procedure, and expected test results. Consequently, inference rules were formulated for each of the test case parts. An example of a inference rule for the test procedure part of the requirements SRSRS4YY-431 is shown below:

```
IF the requirement is for a service and a UART controller is to be
    deactivated
THEN add the call to the requirement's service, calls to a transmission
    service and reception service as well as a recovery call to the
    first service.
```

The condition (if-part) of a heuristic rule is formulated in terms of the individual representing the requirement and the related ontology entities representing connected hardware parts, input/output parameters for the service and the like. The action (then-part) part of the rule contains instructions on how a test case part is to be generated.

After formulating the inference rules, they need to be implemented in a programming language. There are two basics requirements that have to be met by such a language: 1) it should have means to represent the rule in a natural way and 2) it should have means to access the entities in the ontology. We chose Prolog [2] as the language for the implementation as Prolog complies with both of the basic requirements. The acquired inference rules can be implemented with the help of Prolog rules (a Prolog rule is analogous to a statement in other programming languages). After the OWL-to-Prolog translation (described in the previous sub-section), the ontology becomes an inherent part of the Prolog program, and, as a consequence, the ontology entities can be directly accessed by the Prolog code. Finally, the inference engine that is built-in into Prolog is used to execute the coded rules to generate test cases.

An example of the inference rule written in Prolog that implements the previous heuristic rule is given below:

```
1 tc_procedure(Requirement, Procedure) :-
    % get service individual for call #1
2   objectPropertyAssertion(requirementForService, Requirement, Service),
    % check condition for calls #2-4
3   objectPropertyAssertion(requiresAction, Requirement, DeactivateUART),
4   objectPropertyAssertion(actsOn, DeactivateUART, UartController),
```

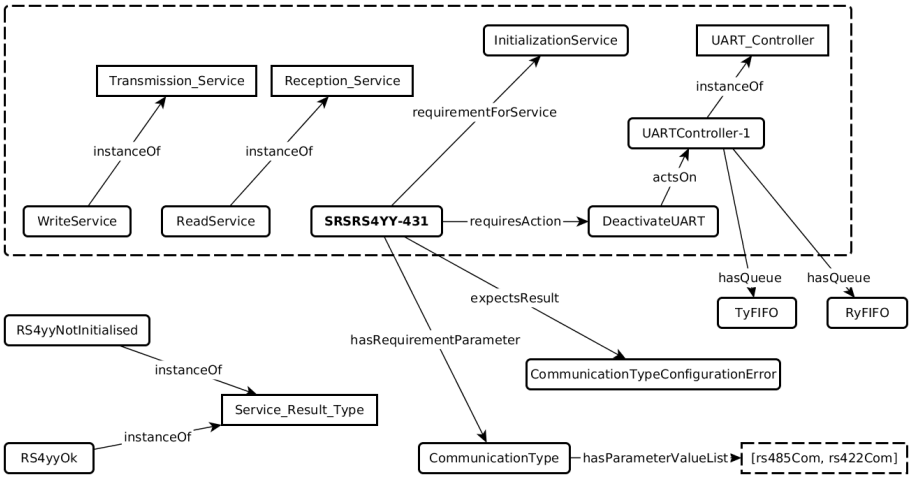

Fig. 3. Ontology paths used by the inference rules to generate a test case for the requirement SRSRS4YY-431. The dashed line indicates the paths used by the inference rule demonstrated in the example above to generate the test procedure part of the test case. The other paths are used by the remaining rules to generate the other parts of the test case.

```
5   classAssertion(uart_controller, UartController),
    % get individuals of the required services
6   classAssertion(transmission_service, WriteService),
7   classAssertion(reception_service, ReadService),
8   Procedure = [Service, WriteService, ReadService, recovery(Service)].
```

Line 1 in the example is the head of the rule consisting of the name, input argument and output argument. Lines 2-7 encode the condition of the heuristic as well as acts as queries to retrieve the relevant entities from the ontology. Line 8 constructs the procedure part of the test case as a list of terms. The list is constructed from the retrieved ontology entities and special term functors.

Fig. 3 shows the ontology entities used by the inference rule, when it is invoked to generate a test case for the requirement SRSRS4YY-431. The figure shows ontology paths, each one being a number of ontology entities connected by object properties or subsumption relation.

Each test case is generated sequentially, from the prerequisites part through to the results part. The generated parts are collected into one structure by the following rule:

```
test_case(Requirement,
  tc(description(TCid, ReqID, Service), Prerequisites, Inputs, Procedure,
     Results)) :-
  req_id(Requirement, ReqID),
  objectPropertyAssertion(requirementForService, Requirement, Service),
```

Table 2. Number of inference rules used to generate different parts of test cases

Test case part	Main rules	Auxiliary rules
Prerequisite conditions	8	4
Test inputs	8	1
Test procedure	5	
Expected test tesults	9	5

```
tc_prerequisites(Requirement, Prerequisites),
tc_inputs(Requirement, Inputs),
tc_procedure(Requirement, Procedure),
tc_results(Requirement, Results),
new_tcid(TCid).
```

Finally, the test case structure is translated into plain text in English. The final result can be found in the right column in Table 3.

3 Experiment and Evaluation

The example provided by Saab consisted of a hardware module with embedded code. The examined part of the example consisted of 15 requirements, specified in the SRS (Software Requirement Specification document), with corresponding 18 test cases, specified in the STD (Software Test Description document). In most cases one requirement is evaluated by executing one test case but in some occasions one requirement is evaluated by executing two or more test cases.

A total of 40 inference rules were used to generate the 18 test cases. The number of rules for each test case part is detailed in Table 2 (an auxiliary rule is intended to be invoked by a main rule). The corresponding test cases have been reproduced in plain English, using the same format as described in the STD document, by applying the inference rules to the ontology. The result from this exercise can be observed in Table 3 where the text in left column is a slightly modified excerpt from the STD document while the text in the right column is the generated output through applying some of the inference rules to the ontology. The result presented in the table corresponds to one specific requirement, in this case SRSRS4YY-431, a requirement that is evaluated in one test case, in this case STDRS4YY-114. As can be observed, there is an almost one-to-one correspondence between the texts in the two columns. However, the authors would like to point out that in some occasions the generated test case texts indicated a discrepancy with the corresponding test case texts found in the STD document. These discrepancies were presented to and evaluated by personnel from Saab and on occasions the observed discrepancies indicated a detected error in the STD document. Hence, this correctness insurance exercise helped improving the quality of the STD document.

Table 3. Test case from the STD (left column) and the corresponding generated test case by applying inference rules to the ontology (right column)

...	...
Test Inputs 1. According to table below. 2. <uartId> := <uartId> from the rs4yy_init call 3. <uartId> := <uartId> from the rs4yy_init call 4. <comType> := rs4yy_rs422Com	**Test Inputs:** 1. <communicationType> := min_value - 1 <communicationType> := max_value + 1 <communicationType> := 485053 2. <uartID> := <uartID> from the initializationService call 3. <uartID> := <uartID> from the initializationService call 4. <communicationType> := RS422
Test Procedure 1. Call rs4yy_init 2. Call rs4yy_write 3. Call rs4yy_read 4. Recovery: Call rs4yy_init	**Test Procedure:** 1. Call initializationService 2. Call writeService 3. Call readService 4. Recovery: Call initializationService
Expected Test Results 1. <result> == rs4yy_comTypeCfgError 2. <result> == rs4yy_notInitialised 3. <result> == rs4yy_notInitialised, <length> == 0 4. <result> == rs4yy_ok	**Expected Test Results:** 1. <result> == communicationTypeConfigurationError 2. <result> == rs4yyNotInitialised 3. <result> == rs4yyNotInitialised, <length> == 0 4. <result> == rs4yyOk
...	...

4 Related Work

The reason for conducting tests on a software product/system is mainly to be able to put some level of trust on the quality and requirement fulfilment of the prod-uct/system. To run the tests on the product/system, some kind of test case(s) must be designed and the corresponding test code(s) be programmed. In many occasions, if not most, this is a manual activity with everything that this embodies of potential errors in the test code caused by missed or misinterpreted requirements due to a deficient test case description or a non-experienced tester.

In an attempt to counteract on these negative effects, model-driven testing techniques have surged in recent years as an alternative field of applied research in the software testing domain [12, 1]. One specific modelling language that has emerged as the prime modeling-tool in this domain is UML. There have been presented a large number of projects with a focus on automatic generation of test cases based on the usage of UML to describe some parts of the testing activities [9] Other examples of model-driven test case generation projects have been based on Function Block Diagrams [4] or State-based testing [7], just to mention two. The different model-driven test case generation approaches presented by different researcher teams often depend on some kind of requirement specification as input to the process [9]. When it comes to the focus of the test activities, i.e. what is the output from the testing activities that needs to be evaluated, two main areas can be identified, code coverage testing (which could be looked upon as testing the output of a software design process) and require-ment coverage

testing (which could be looked upon as testing the input to a software design process). All of the presented model-driven test case generation approaches referred to earlier have had a focus on some kind of code coverage. However, in some application domains the verification of the coverage of the requirements, which means that all requirements stated in a requirements specification document have been considered and tested in a traceable manner, is equally, and sometimes even more, important than code coverage. This is the case in, for example, the avionics industry of which Saab is a perfect example.

There exist only a few projects that rely on ontologies for software testing activities, for example [13, 5]. As mentioned earlier in this paper, an ontology represents a formal model of the knowledge captured for a specific domain, in this paper being software testing. However, it should be stressed that the creation of an ontology is only the first step in order to automatically create software test cases. It must also be contemplated that the test cases must be generated with some specific test objectives in mind. The OSTAG-project that has been presented in this paper is one of very few examples where both code coverage and requirement coverage can be handled.

Prolog has been used as a reasoner for OWL ontologies in a number of cases. For example, in [15] the authors describe an approach to reasoning over temporal ontologies that translates OWL statements to clauses in Prolog and then uses the built-in inference mechanism. In [10] an OWL ontology and OWLRuleML rules are translated into Prolog clauses, which are then used to infer new facts by the Prolog inference engine. The work presented in this paper has utilised a similar idea, however, we have used OWL functional-style syntax for the OWL to Prolog translation, which makes queries to the ontology as close as possible to OWL syntax.

5 Conclusions

We have proposed an approach to generate software test cases based on the use of an ontology, representing software requirements as well as knowledge about the components of the software system under development, and inference rules, representing strategies for test case creation. The inference rules are coded in Prolog and the built-in inference engine is used for executing the rules. During the execution the rules query the ontology to check conditions and retrieve data needed for the construction of test cases. To make this possible, the ontology is serialised in OWL functional-style syntax and then translated to Prolog syntax. The first experiment showed that, by using 40 inference rules, 18 test cases for 15 requirements were generated as plain text in English. The examination of the result showed an almost one-to-one correspondence between the texts in the generated test cases and the texts provided by one of our industrial partners, Saab.

The translation from the OWL functional-style syntax to the Prolog syntax allowed for seamless integration of the ontology into the Prolog program. On one hand, the syntax of the OWL statements was preserved to a great extent.

On the other hand, the inference rules could directly reference the ontology constructs in their bodies. The Prolog inference mechanism took care of finding an inference rule with a satisfied condition and firing it. As a result, the ontology was effectively used for an applied purpose automation of software testing. However, it should be noted that not all OWL statements are translated at the moment. Most notably complex class constructors are not translated (due to the fact that we did not find the need to use them in the ontology so far). There is also lack of inference rules preserving the semantics of OWL, e.g. rules to find all individuals of a class having several subclasses. Moreover, the conducted experiment is of limited scale. More experiments with an increased number of inference rules are needed to evaluate the proposed approach to demonstrate its full potential.

There exist other languages to implement inference rules, e.g. SWRL or the inference rule language built-in in Jena, which are closer to the syntax and semantics of OWL and follow the open world assumption. Such languages may be better suited for situations when new data need to be integrated into the knowledge base. Despite that, we have chosen Prolog because our case does not require data integration. Additionally, Prolog provides both inference mechanism and traditional programming faciliteis, thus, eliminating the need to use one language for implementing inference rules and another one for developing a software prototype. At the same, time OWL was chosen as the languange to implement the ontology to support test case generation bacause the ontology can also be used to support other tasks in the software development process, such as requirement analysis, and requirement verification and validation.

The future work will go along the lines of increasing the number of inference rules to generate test cases for the so far uncovered requirements. This will allow us to further test the applicability of the proposed approach of combining an OWL ontology and Prolog inference rules in an ontology-based application, such as in the software test case generation domain. A comparision can also be done between the Prolog and OWL reasoning systems.

So far the results from the project have been positive and have demonstrated the feasibility of producing test cases in a semi-automatic fashion. The automation of the test case generation process has demonstrated that the quality (in our case the correctness) of the generated test cases were improved. Minor errors that went undetected by the human test case designers were identified and corrected as mentioned in Section 3. This result puts the finger on the benefits of automating a process in general and the test case generation process in specific. However, this is not the only measurable result that is expected to come from the project. In the near future other types of metric are going to evaluated, such as to quantify the time savings gained from automating the test case generation process through real-life time studies, and to verify the coverage of the requirements to demonstrate that all requirements stated in a requirements specification document have been considered and tested.

Acknowledgments. The research reported in this paper has been financed by grant #20140170 from the Knowledge Foundation (Sweden).

References

1. Anand, S., Burke, E., Chen, T., Clark, J., Cohen, M., Grieskamp, W., Harman, M., Harrold, M., McMinn, P.: An orchestrated survey on automated software test case generation. Journal of Systems and Software 86(8) (August 2013)
2. Bratko, I.: Prolog Programming for Artificail Intelligence. Pearson Education, 4th edn. (2011)
3. CapGemini, HP, Sogeti: World quality report 2015-16 (2015), 80 pages
4. Enoiu, E., Causevic, A., Ostrand, T., Weyuker, E., Sundmark, D., Pettersson, P.: Automated test generation using model-checking: An industrial evaluation. International Journal on Software Tools for Technology Transfer pp. 1–19 (2014)
5. Freitas, A., Vieira, R.: An ontology for guiding performance testing. In: 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). pp. 400–407 (2014)
6. Happel, H.J., Seedorf, S.: Applications of ontologies in software engineering. In: Proc. of Workshop on Sematic Web Enabled Software Engineering (SWESE) on the ISWC. pp. 5–9 (2006)
7. Holt, N., Briand, L., Torkar, R.: Empirical evaluations on the cost-effectiveness of state-based testing: An industrial case study. Information and Software Technology 56, 890–910 (2014)
8. Judge Business School, Cambridge University: Cambridge university study states software bugs cost economy $312 billion per year (2013), `http://www.prweb.com/releases/2013/1/prweb10298185.htm`. Accessed September 22nd, 2016
9. Kaur, A., Vig, V.: Systematic review of automatic test case generation by UML diagrams. International Journal of Engineering Research & Technology (IJERT) 1(6) (August 2012), 17 pages
10. Laera, L., Tamma, V., Bench-Capon, T., Semeraro, G.: SweetProlog: A system to integrate ontologies and rules. In: Proc. of the 3rd RuleML workshop Rules and Rule Markup Languages for the Semantic Web. LNCS, vol. 3323, pp. 188–193. Springer (2004)
11. Motik, B., Patel-Schneider, P., Parsia, B.: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C, 2nd edn. (2012)
12. Mussa, M., Ouchani, S., Al Sammane, W., Hamou-Lhadj, A.: A survey of model-driven testing techniques. In: QSIC '09. 9th International Conference on Quality Software. pp. 167–172 (2009), 24-25 August 2009, Jeju, South Korea
13. Nasser, V., W., D., MacIsaac, D.: Knowledge-based software test generation. In: The 21st International Conference on Software Engineering and Knowledge Engineering. pp. 312–317. Boston, U.S.A. (July 2009)
14. Nguyen, C.D., Perini, A., Tonella, P.: Ontology-based test generation for multi-agent systems. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems. vol. 3, pp. 1315–1320 (2008)
15. Papadakis, N., Stravoskoufos, K., Baratis, E., Petrakis, E., Plexousakis, D.: PROTON: A Prolog reasoner for temporal ontologies in OWL. Expert Systems with Applications 38(12), 14660–14667 (2011)
16. Wang, Y., Bai, X., Li, J., Huang, R.: Ontology-based test case generation for testing web services. In: Autonomous Decentralized Systems, ISADS'07. Eighth International Symposium on. pp. 43–50. IEEE (2007)