Ontology-based Software Test Case Generation (OSTAG)

Vladimir Tarasov[1], He Tan[1], Anders Adlemo[1], Anders Andersson[1], Muhammad Ismail[1], Mats E. Johansson[2] and Daniel Olsson[3]

[1]School of Engineering, Jönköping University, Box 1026, 551 11 Jönköping, Sweden
[2]Saab AB, Avionics Systems, Stensholmsvägen 20, 561 85 Huskvarna, Sweden
[3]AddQ AB, Odinsgatan 11, 411 03 Göteborg, Sweden
{vladimir.tarasov, he.tan, anders.adlemo, anders.andersson, muhammad.ismail}@ju.se, mats.e.johansson@saabgroup.com, daniel.olsson@addq.se

Abstract. Testing is a paramount quality assurance activity in every software development project, especially for embedded, safety critical systems. During the test process, a lot of effort is put into the generation of test cases. The presented OSTAG project aimed at developing methods and techniques to automate the software test case generation for black-box testing. The proposed approach was based on the creation of a software requirements ontology and the application of inference rules on the ontology to derive test cases. The ontology represents knowledge of the requirements, the software system and the corresponding application domain while the inference rules formalize knowledge from documents and experienced testers in the domain of test planning and test case generation. A software prototype of the approach was implemented and one of the industrial project partners evaluated the results. An alternative method for generating test cases, based on genetic algorithms, was also explored.

Keywords. Black-box Testing, Embedded Systems, Genetic Algorithms, Inference Rules, Knowledge Modelling, Model-Based Testing, Ontology Development, Ontology Quality Evaluation, Ontology Verbalisation, OWL, Prolog, Protégé, Software Requirements Specification, Test Case Generation.

1 Introduction

The software market is increasing on a yearly basis and shows no sign of slowing down. According to Gartner, the worldwide IT spending is predicted to grow 2.7% in 2017, to reach a total of USD 3.5 trillion [1]. As software products and systems permeate every aspect of our lives, we become more and more dependent on their correct functioning. Consequently, quality concerns are becoming much more vital and critical as end-users become increasingly dependent on products that include software. As is the case in all product development, the quality of the software must be verified and validated through painstaking test activities, such as test planning and design, ocular reviews of requirements documents and program code, program testing, system testing, acceptance testing, and so on. Despite these efforts, errors sometimes remain undetected in the code.

Tarasov V., Tan H., Adlemo A., Andersson A., Muhammad I., Johansson M. and Olsson D.
Ontology-based Software Test Case Generation (OSTAG).
DOI: 10.5220/0007901301350159
In *European Projects in Knowledge Applications and Intelligent Systems (EPS Lisbon 2016 2015)*, pages 135-159
ISBN: 978-989-758-356-8

In accordance to Capgemini World Quality Report 2017 [2], the budget allocation for quality assurance and testing, as percentage of IT expenditures in the software industry, was 31% in 2016. Another recent figure, coming from a study performed by the Cambridge University [3], estimated the yearly cost of software errors caused by poor quality procedures to roughly USD 312 billion.

One way of slowing down this ongoing cost increase related to software testing activities is to automate as many as possible of these activities. As far as test management and test script execution goes, this is a mature field where commercial products assist software testers in their daily work, like TestingWhiz [4] or HPE Unified Functional Testing [5]. Recent research results indicate that automatically generated tests achieve similar code coverage as manually created tests, but in a fraction of the time (an average improvement of roughly 90%) [6]. The starting point of the OSTAG project, presented in this book chapter, has been to provide a method of semi-automatically preparing software test cases, followed in the near future by a project developing a complete automation process. One problem when generating test cases is to come up with a ”reasonable good” set of ”adequate” software test cases (terms that have not yet been unambiguously defined by the research community), but results presented by Enoiu at al. indicate, that ”the use of an automated test generation tool does not result in better fault detection compared to manual testing” [6]. For the past 10-15 years, the research community has been proposing different techniques to alleviate the burden of manually creating test cases through an automatic generation of test cases (e.g. [7]) that converts software requirements into a formal model (e.g. using statecharts [8]). Another area of automatic test case generation is through evolutionary testing that appears to be successful for automatic test case generation in white-box testing [9, 10]. The use of evolutionary testing techniques has also been evaluated in the OSTAG project in the form of genetic algorithms for the generation of software test cases, but for black-box testing, with the big difference that no source code was required.

A software product is usually valued or appreciated through its functionality, sometimes ”externally” through the interaction with a human, or otherwise ”internally” through the interaction with another software module. Whatever the case and apart from this functionality validation, software also embodies and reflects the implicit knowledge of the application domain in which it is supposed to function. At its core, the software can be viewed as being a knowledge repository where the knowledge is largely related to the application domain [11]. Consequently, it is essential to be able to make use of the knowledge related to relevant aspects surrounding and influencing the software, e.g. specific domain knowledge, past and new requirements, policies, and contexts in which people or end-users use and interact with the software. Thus, using this knowledge to support more intelligent software development processes requires a machine-facilitated understanding of the knowledge. The management of relevant and essential knowledge related to software engineering in general, and software testing in specific, is thus of utmost importance. A mapping study by Ferreira de Souza et al. [12] initiated with an evaluation of 562 publications related to a number of knowledge management initiatives in the software testing domain. Their study did not disclose any previous research making use of ontologies for *software test case generation*. They have, however, presented a reference domain ontology of their own, representing *software testing knowledge* [13].

One way of handling software requirements knowledge used to generate test cases (as described in this book chapter), is by modelling the knowledge relying on semantic technologies. An example of such a semantic technology is a formal model known as an ontology. A commonly cited definition by Studer et al. states that ”an ontology is a formal explicit specification of a shared conceptualisation” [14], a definition that extends the definition by Gruber [15] and Borst [16] that both stated that ”an ontology is an explicit specification of a conceptualisation”. The following explications of the terms in the definition come from Studer et al. A *conceptualisation* refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. *Explicit* means that the type of concepts used, and the constraints on their use are explicitly defined. For example, in medical domains, the concepts are diseases and symptoms, the relations between them are causal and a constraint is that a disease cannot cause itself. *Formal* refers to the fact that the ontology should be machine readable, which excludes natural language. *Shared* reflects the notion that an ontology captures consensual knowledge, that is, it is not private to some individual, but accepted by a group. By explicitly modelling a shared conceptualisation, or domain knowledge, in a machine-readable format, ontologies provide the possibility of representing, organizing and reasoning over the knowledge of a domain of interest, and can serve as a basis for different purposes. The software engineering community has recognized ontologies as a promising way of addressing many current problems in different phases of the software life-cycle process, such as requirements specifications, software design, software implementation and integration, and software maintenance [17].

Until recently, the use of ontologies in software testing has been one of the least explored areas of software engineering [17, 18] and has thus not been discussed as much as their use in other stages of the software life-cycle process. In [17], Happel and Seedorf present possible ways of utilizing ontologies for the generation of test cases, and discuss the feasibility of reusing domain knowledge encoded in ontologies for testing. In practice, however, few tangible results have been presented. Most of the research have had a focus on the testing of web-based software and especially web services (e.g. [19–21]). One mature area where ontologies have been successfully applied is *requirements engineering*. Requirements engineering is concerned with the elicitation, specification and validation of the requirements encountered in software systems [22]. The use of ontologies in requirements engineering date back to the 1990s, e.g. [23–25]. There exists a clear synergy effect between the ontological modelling of domain knowledge and the modelling of requirements performed by requirement engineers [26]. Recently, a renewed interest in utilizing ontologies in requirements engineering has surged due to the appearance of semantic web technologies [17, 26].

In the OSTAG project, two case studies were provided by two of the participating project companies, Saab Avionics and AddQ. The first case study (provided by Saab Avionics) originated from the *avionics domain*. In the avionics industry, many of the systems are required to be highly safety-critical. For these systems, the software development process must comply with several industry standards, like DO-178B [27]. The requirements of the entire system, or units making up the system, must be analysed, specified and validated before initiating the design and implementation phases. In the research project presented in this book chapter, an ontology was developed representing the requirements of a software component pertaining to an embedded system located in a fighter airplane. The requirements used by Saab are written in natural language and joined in documents, such as software requirements specification (SRS) documents and interface requirements specification (IRS) documents, followed by a manual validation performed by domain experts within the avionic industry. The requirements ontology, created in the project, represented the software requirements, the software, the hardware and the communication components belonging to the embedded system. An ontology by itself is a static representation of a knowledge domain, such as software requirements. Once the ontology has been created, it must be used to create the software test cases. One way of doing this is by using inference rules that represent the expertise of an expert software tester. The inference rules were coded in Prolog and made use of the ontology entities to generate the test cases. The Prolog inference engine controlled the process of selecting and invoking the inference rules.

The second case study (provided by AddQ) originated from the *car manufacturing domain* and, more specifically, the representation of an embedded module complying with the Automotive Open System Architecture (AUTOSAR 4.0) standard [28]. In general, the AUTOSAR standard serves as a platform upon which vehicle applications are implemented to minimize the barriers between different functional domains. It introduces a standardized layer between the application software and the electronic control unit hardware, thus making the software largely independent of the chosen microcontroller and original equipment manufacturer (OEM). The result is a simplified development process that enables high flexibility and an easy reuse of the application software. AUTOSAR is used by most of the car manufacturers, like BMW, Volkswagen, Toyota and Volvo. The AUTOSAR application that was modeled in the second case study consisted of a communication manager module (ComM) that acts as a resource manager which encapsulates the control of the underlying communication services.

Until this moment in time (2017), within the OSTAG project, we have implemented the method of ontology-based software test case generation for the first case study (the avionics case), and we have also run some evaluations. The results indicate that it is feasible to create software test cases relying on a requirements ontology, and that the quality of the generated test cases is equally good as the manually produced test cases. Details of the implementation of the first case study, together with some results and the evaluations of these, are presented in section 3, 4 and 5, respectively. Currently, we continue our research with a focus on the AUTOSAR case study. The requirements from the avionics case are on a *functional level* (unit testing). They are rather well defined and documented in text with a clear structure. Hence, it is relatively straightforward to generate an ontological model from these requirements. The AUTOSAR requirements, on the other hand, are on a *high level*. The requirements are documented in long and complex texts. Preliminary results indicate that additional tools are required to be able to produce the requirements ontology. At the end of this book chapter, we comment on the additional challenges related to the AUTOSAR case study and propose possible solutions to respond to these challenges.

The rest of the book chapter is structured as follows. Section 2 introduces the readers to related work in the ontology, inference rules, and genetic algorithms areas. Section 3 discloses the details of the avionics case study and the generated ontology representing the software requirements and the software components. After having a good knowledge of the ontology, section 4 outlines the details of how to produce the software test cases. The section includes a presentation of the translation of the ontology from an OWL representation to an executable Prolog equivalent, presents the application of the inference rules (also represented in Prolog) on the ontologies to create the software test cases, and ends with an outline of genetic algorithms, their use in software testing and some preliminary results originating from the OSTAG project. No implementation of a model is complete without a proper evaluation of it. Thus, some initial evaluation results are presented and discussed in section 5. It should be noted that the evaluations and the results from these are coming from the first case study only, i.e. related to the avionics domain. In section 6, some general conclusions of the OSTAG project are presented and, to sum up, in section 7, several promising future research directions are outlined.

2 Related Work

The reason for conducting tests on software products/systems is mainly to be able to put some level of trust on the quality and requirement fulfilment of said products/systems. To be able to run tests on a product/system, test case(s) must be designed and the corresponding test script(s) developed. When it comes to the focus of software test activities, as far as the test code is concerned, two main areas can be identified; code coverage testing (which could be looked upon as testing the output of a software design process) and requirement coverage testing (which could be looked upon as testing the input to a software design process). All the presented model-driven test case generation approaches referred to earlier have had a focus on code coverage. However, in some application domains the verification of the coverage of the requirements, that is that all requirements stated in a requirements specification document, have to be considered and tested in a traceable manner, and the requirements coverage is sometimes equally or even more important than code coverage. An example of this are the testing activities performed by one of the industrial partners, Saab Avionics, where both code coverage testing and requirement coverage testing are of equal importance. Code coverage testing can be looked upon as the verification of the functional correctness of a software product/system (i.e. is the functionality correctly implemented) while requirement coverage testing can be looked upon as the validation of the functional correctness of a software product/system (i.e. is the correct functionality implemented). Thus, the OSTAG project is one of few research projects where requirement coverage has been contemplated from a test case generation point of view.

In many occasions, the design and implementation of test cases is a purely manual activity. If this task could be automated it would help test designers who are developing the test cases. Model-Based Testing (MBT) is a method to create *functional* test cases [29–31]. With MBT it is possible to generate test cases from models that describe the test object or system under test. Some benefits of MBT are:

- it provides the opportunity to automate the process for test specification,
- it creates an acceleration in the specification of test scripts,
- it makes the time and resource consuming task of test specification less dependent on the amount and expertise of testers,
- with the use of a model, small changes in the documentation are translated into new test scripts in only a few seconds.

One specific modelling language that has emerged as the prime modelling tool is the Unified Modelling Language (UML). Many projects have been presented that have had a focus on the automatic generation of test cases based on the usage of UML [32]. Other model-based test case generation projects have relied on Function Block Diagrams [33] or (Finite) State Machines [34, 35]. Two specific research approaches, with the goal of automatically producing test cases and that both bear resemblances to the OSTAG project, is the TextAnalyzer tool by Sneed [36, 37] and the SOLIMVA methodology by de Santiago et al. [38–40]. MBT is an approach based on creating test cases derived from a behaviour model of the test object, the (test) model. This model describes the expected behaviour of the test object. Test cases are then, where possible, automatically generated from the test object. The challenge with this approach lies in the creation of a formal behaviour model in which the operation of (part of) the application is represented. How this model should be created, and exactly what details that should go into the model, is not self-evident. Furthermore, if the development of the model is not performed by experts, potential errors could be introduced, errors that could propagate to the generated test cases. Instead of the extra step of having to create a test model to generate the test cases, in the OSTAG-project the test cases were created based on the requirements captured in the requirements ontology.

A lot of research efforts have been put into the application of ontologies in requirements engineering (e.g. [41–43]). Much of the research deals with inconsistency and incompleteness problems in requirement specifications. For example, in [41], an ontology is proposed to support the cooperation between requirements engineers. The ontology provides a formal representation of the engineering design knowledge that can be shared among engineers, such that the ambiguity, inconsistency, incompleteness and redundancy can be reduced when engineers concurrently develop requirements for sub-systems of a complex artifact. In other related work, such as described in [24, 42], ontologies are proposed to provide a framework for the requirements modelling to support the requirements engineering process. Such a framework can help mitigating the difficulties encountered in requirements engineering, such as negotiating a common understanding of concepts, dealing with ambiguity, and clarifying desires, needs and constraints. Another direction in the application of ontologies in requirements engineering is to represent requirements in a formal ontology language, to support consistency checking, question answering, or inferring propositions (e.g. [23, 44]). Most of the work within this direction focus on the analysis of consistency, completeness and correctness of requirements through reasoning over requirements ontologies. The work presented in this book chapter is also concerned with the representation of requirements in an ontology, but the ontology is mainly employed to support advanced methods in the subsequent stages of the software development process and, more specifically, the generation of test cases.

The Prolog language has been used as a reasoner for OWL ontologies in many different projects. For example, in [45], the authors describe an approach of reasoning over temporal ontologies that translates OWL statements to clauses in Prolog and then makes use of the built-in inference mechanism. In [46], an OWL ontology and OWLRuleML rules were translated into Prolog clauses, which were then used to infer new facts through the use of the Prolog inference engine. The work presented in this book chapter has also used Prolog as a reasoner. The difference is that the ontology was translated from OWL functional-style syntax to Prolog syntax, thus providing a more natural way of writing rules querying the ontology.

Another way of producing test cases that was evaluated in the OSTAG project, apart from using ontologies and inference rules, was *evolutionary testing*. The field of *evolutionary computing* goes back to the 1960's, when *evolutionary programming* [47], *evolution strategies* [48] and *genetic algorithms* [49] surged, later to be joined by *genetic programming* and other similar techniques. The idea behind these efforts was to come up with algorithms inspired by the biological evolution and apply these on computational problem areas. The special area of software testing and the application of evolutionary techniques and genetic algorithms to automatize testing started in the early 90's. Since then, several articles have been presented, e.g. [50–55]. It should be noted, however, that all of the results presented in these articles have been based on the fact that they had access to the source code. Furthermore, what was created using evolutionary techniques and genetic algorithms was test data and not complete test cases or test scripts. Hence, there is a difference in complexity between automatically creating qualitative test data and complete, useful and correct test cases (where the test data is only one of several components). The idea behind the work in the OSTAG project has thus been to create complete test cases, including predefined *prerequisites*, necessary *input data*, correct *test procedures*, and useful *output data*. The complete test cases are created based on the information encountered in the developed ontology, an ontology whose main object is that of modeling the requirements defined in the requirements document. Some initial results from this challenging work are presented further on in this book chapter.

3 Ontology Development

In this section, the requirements ontology that was developed based on information from the avionics case study [56] is presented. The resulting ontology represents the requirements of a telecommunication software component pertaining to an embedded system used in an avionics system placed in a fighter air-plane. The software component in the embedded system is fully developed in compliance with the DO-178B standard. As defined in DO-178B, the requirements of the software component have been prepared, reviewed and validated. The results indicate that the ontology can successfully support the generation of test cases. But before going into any more details of the development process that was applied during the creation of the ontology, we describe some ontology methods that influenced our development process to a greater or lesser degree. Over the years, many methods have been presented for building ontologies. For example, Ontology 101 [57] proposes a very simple but practical guide for building an ontology using an ontology editing environment, such as Protégé [58]. METHONTOLOGY [59] contributes with a general framework for ontology development, which defines the main activities that people need to carry out when building an ontology, and outlines three different processes: management, technical, and supporting. The OTK Methodology [60] focuses on an application-driven ontology development. There are also other methods of ontology engineering, e.g. [61] that introduces eXtreme Design (XD) stemming from eXtreme Programming, the latter being a software development method. All of the mentioned methods focus on a collaborative, incremental, and iterative process of ontology development. Unfortunately, no one, single ontology development method was sufficient for our ontology development process.

The ontology development process followed in this project is illustrated in Figure 1. During the development of the ontology, the developers worked as a pair and followed an iterative and incremental process. In each iteration, the developers basically followed the steps suggested in Ontology 101, and considered the activities described in the supporting process in METHONTOLOGY. Lightweight *competence questions* (CQs) were used as a guidance to build the ontology. These CQs are simple and quickly prepared. In each iteration the developers had an opportunity to meet with the industry experts and discuss the issues they had encountered during the acquisition and specification steps. The developers also received feedback from the users of the ontology, and modified the ontology when needed. The development tool used was Protégé. HermiT reasoner [62] was used to check the consistency of the ontology. Finally, the ontology was written in OWL [63].

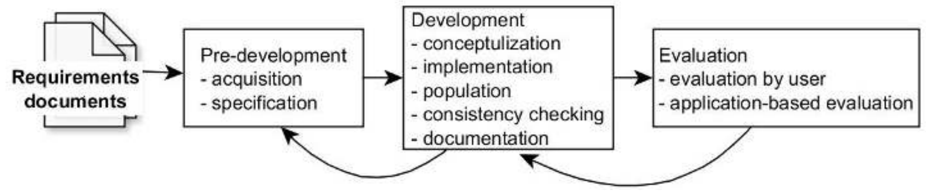

Fig. 1. The ontology development process.

The developed ontology includes three specific pieces of knowledge:

- a *meta model* of the software requirements,
- the *domain knowledge* of the application, e.g. general knowledge of the hardware and software, electronic communication standards, etc.,
- all the *requirement specification*s defined in the SRS document.

The current version of the ontology contains 42 classes, 34 object properties, 13 datatype properties, and 147 instances in total. Figure 2 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 requirement parameters, which are inputs of a requirement, requirement conditions, and results, which are usually outputs of a requirement, and exception messages. Some requirements require the system to take actions. Furthermore, there exists a traceability between different requirements, e.g. traceability between an interface requirement and a system requirement. Figure 3 illustrates an ontology fragment of the *domain knowledge* of the telecommunication software component. Figure 4 shows the ontology fragment of one particular functional *requirement specification*, in this example, the SRSRS4YY-435. The functional requirement defines that if the parity is out of its valid range, the initialisation service shall deactivate the Universal Asynchronous Receiver/Transmitter (UART), and return the result "parityCfgError". The ontology fragments for all of the remaining individual requirements of the software component, pertaining to the embedded system, are similar to the 435 requirement. In figures 2 to 4, the orange boxes represent the concepts of the ontology; the white boxes represent the instances; and the green boxes provide the data values of the datatype property for the instances.

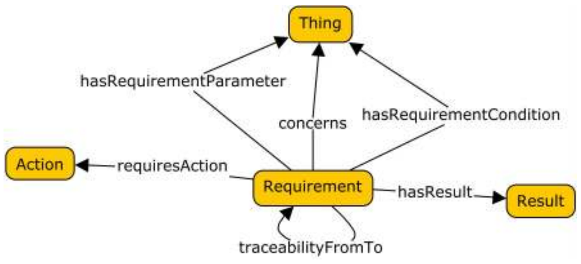

Fig. 2. The *meta model* of the software requirements in the ontology.

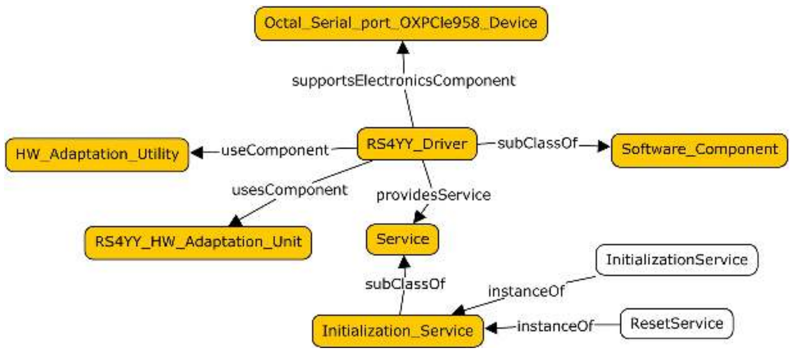

Fig. 3. Ontology fragment for the *domain knowledge*.

4 Test Case Generation

In this section, two different approaches of producing test cases will be outlined. The first approach is based on the usage of inference rules together with the requirements ontology. The second approach is based on the use of genetic algorithms to produce test cases but without the need of any source code or any executable version of the software.

4.1 Using Inference Rules to Derive Test Cases from the Ontology

In this subsection are presented the inference rules and how they have been applied to the requirements ontology to create test cases [64]. But before going into any details,

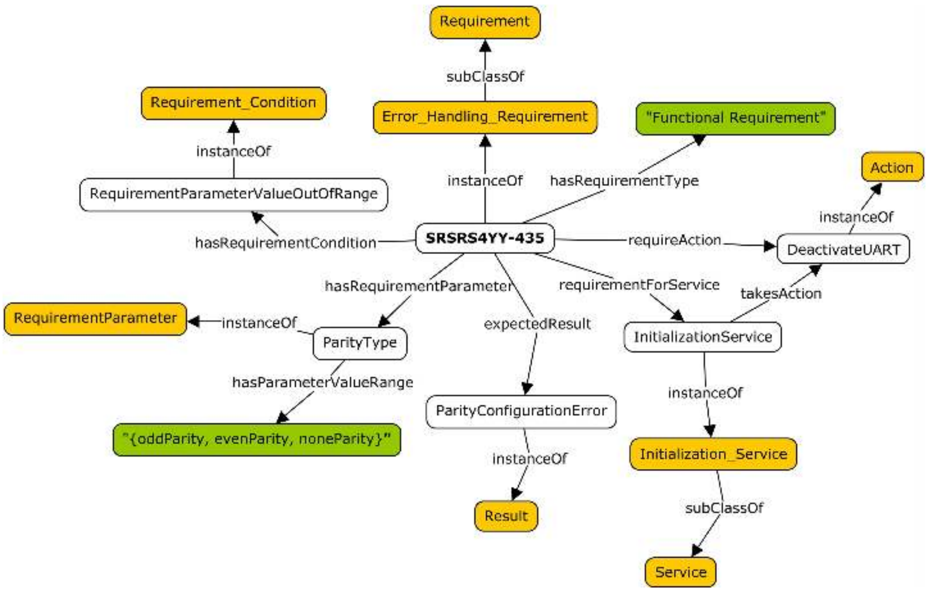

Fig. 4. Ontology fragment for a *requirement specification*, SRSRS4YY-435.

we first present a short background to the problem area of test case generation. To the best of our knowledge, there exist only a limited number of projects that rely on ontologies for software testing activities, for example [65, 66]. As defined earlier in this book chapter, an ontology represents a formal model of the knowledge captured for a specific domain. It should be stressed, however, that the creation of an ontology is only the first step in the automatic creation of software test cases. One must also contemplate the generation of the test cases, having some specific test objectives in mind. The creation of test cases can be realized in several ways, but in the OSTAG project we chose to make use of inference rules.

First of all, the ontology was translated into a syntax that was supported by the inference rules, i.e. Prolog syntax, as Prolog was chosen for coding the inference rules. We chose Prolog [67] as the language for the implementation as Prolog has means of representing the rules in a natural way and has means of accessing the entities in the ontology. Prolog also has a built-in inference engine that was used to execute the coded rules, to generate the test cases.

An OWL functional-style syntax was chosen as the starting point for the ontology conversion as this syntax was the most similar to the Prolog syntax. An ontology document in the functional-style syntax is a sequence of OWL constructs, each located on a separate line, as well as a number of prefix definitions [68]. A Python script was written for the OWL-to-Prolog translation, which processed the ontology document, line by line. Each OWL statement was tokenized and converted according to the Prolog syntax rules. The list of tokens was subsequently converted into the corresponding Prolog statement. The following OWL statements were translated: Declaration, ClassAssertion, SubClassOf, ObjectPropertyAssertion, DataPropertyAssertion, objectProperty-

Table 1. Example of the conversion of several OWL statements.

OWL functional-style syntax	Prolog syntax
Declaration(Class(Requirement))	declaration(class(requirement)).
Declaration(ObjectProperty(OSTAG:requir-ementForService))	declaration(objectProperty(requirem-entForService)).
DataPropertyAssertion(:hasParameter-ValueList :ParityType "[oddParity, noneParity, evenParity]"^^xsd:string)	dataPropertyAssertion(hasParamete-rValueList, parityType, [oddParity, noneParity, evenParity]).
ObjectPropertyAssertion(OSTAG:requirement-ForService :SRSRS4YY-435 :InitializationService)	objectPropertyAssertion(require-mentForService, srsrs4yy_435, initializationService).
ObjectPropertyDomain(OSTAG:require-mentForService OSTAG:Requirement)	objectPropertyDomain(requirementFo-rService, requirement).
AnnotationAssertion(rdfs:label OSTAG:FIFO "FIFO")	annotationAssertion(rdfs(label), fifo, 'FIFO').
Annotation(owl:versionInfo "3.0"^^xsd:decimal)	annotation(owl(versionInfo), 3.0).

Range, objectPropertyDomain, Annotation, AnnotationAssertion. Table 1 shows several examples of the translation from OWL to Prolog.

After the translation of the requirements ontology from an OWL-syntax to a Prolog equivalent, the inference rules were applied to the ontology, to produce the test cases. To do this, the testers' expertise on how they use requirements to create test cases, was represented. Only a few general guidelines for testing can be found in literature, such as boundary value testing. Also, most expertise is specific to particular types of software systems and/or particular domains. Hence, experienced testers at Saab Avionics were interviewed and existing test cases were studied, together with their corresponding requirements, in order to capture the expertise. This kind of knowledge expressed inherent strategies for test case creation and was represented as if-then rules.

In the OSTAG project, 16 requirements were examined with 20 corresponding test cases. Each requirement described some functionality of a service (function) from a driver for a hardware unit. All requirements were grouped according to services. The requirements covering six services were analysed. During the analysis of an original test case, a test case that had been manually created by a software tester, it was compared with the corresponding requirement to fully understand how the different parts of the original test case had been constructed. In the subsequent discussions with the industry software testers that participated in the study, any inconsistencies or remaining doubts were resolved. The activities resulted in a set of inference rules formulated in plain English. Each original test case consisted 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 prerequisites part of the requirements SRSRS4YY-435 is shown below:

```
IF the requirement requires deactivation of a UART controller and
    the controller has queues
THEN add the prerequisites of filling the queues with data
```

The condition (if-part) of each inference rule was 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 rules embodied instructions on how a test case part should be generated. An example of a Prolog rule that implements the previous inference rule is given below:

```
1 tc_prerequisites(Requirement, Prerequisites) :-
2     objectPropertyAssertion(requiresAction, Requirement,
          DeactivateUART),
3     objectPropertyAssertion(actsOn, DeactivateUART,
          UartController),
4     classAssertion(uart_controller, UartController),
5     setof(Queue,
          objectPropertyAssertion(hasQueue, UartController,
              Queue),
          QueueList),
6     queue_prereq(QueueList, QueueList, Prerequisites).
```

Line 1 in the example is the head of the rule consisting of the name, input argument and output argument. Lines 2-5 encode the condition of the rule as well as act as queries to retrieve the relevant entities from the ontology. Line 6 constructs the prerequisites part of the test case.

Each test case was generated sequentially, from the prerequisites part through to the results part. The generated parts were 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),
  tc_prerequisites(Requirement, Prerequisites),
  tc_inputs(Requirement, Inputs),
  tc_procedure(Requirement, Procedure),
  tc_results(Requirement, Results),
  new_tcid(TCid).
```

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

4.2 Using Genetic Algorithms to Evolve Test Cases

In the OSTAG project, the use of genetic algorithms in the generation of test cases has been investigated. Over the years and in numerous projects, evolutionary testing techniques have been used to create test cases, as described earlier in the book chapter. Genetic algorithms can be used to find an optimal set of test cases that covers *all code* in a program or *all paths* in an execution path diagram, something that requires access to the program code. This is also known as white-box testing. In black-box testing, on the other hand, where no program code is available but only an executable version of the software under testing, evolutionary techniques have been used to find erroneous configurations and input data.

In the OSTAG project, no source code or executable version of the software were available. Instead of code or path coverage, we introduced the concept of *ontology coverage* as a measure by which test cases and test case sets could be evaluated. Since the ontology is assumed to include all software requirements, a set of test cases that completely covers the ontology would therefore, among other things, guarantee that all requirements are tested. This approach is, to the best of our knowledge, something completely new which has not been previously investigated.

When using genetic algorithms, a population of individuals is evaluated and developed in the direction of some optimum. Each individual is assigned a so called fitness value, which makes it meaningful to talk about better or worse members of the population. New individuals can be created by mutations, small variations of existing individuals, or by different cross-over operations, where selected individuals, parents, give rise to new individuals, children, inheriting some of their parents' properties. The fitness of an individual often plays an important role, both when selecting individuals to mutate or to become parents, and when deciding if a new-born individual shall replace an old one.

In our project, each individual in the following description represents a set of test cases. Usually, a test case contains several parts. There are *prerequisites*, with the intention to put the machine in a certain state, *input data*, and a description of the test *procedure*. These three parts can be looked upon as being different kinds of input data. Hence, a set of test cases is just a sequence of inputs. However, a complete test case must also contain some expected *output*. But what could be judged as being expected outputs from a certain sequence of prerequisites, input data and test procedures, must be validated based on some software documentation and can, for obvious reasons, not be generated using random methods.

A mutation of an individual is a change of a single input in the sequence, cross-over means that new sequences are built taking parts of two existing sequences. A mutation that alter the length of a sequence is also introduced. The fitness value is calculated from the length of the sequence, something that we want to minimize, and the ontology coverage, which should be maximized. To give these two goals appropriate weights, the fitness of a sequence $\boldsymbol{x}$ is given by the fitness function

$$f(\boldsymbol{x}) = A \cdot C_o(\boldsymbol{x}) - L(\boldsymbol{x}),$$

where A is a (large) constant, C_o is the ontology coverage computed as (number of covered ontology instances)/(total number of instances in the ontology), and $L(\boldsymbol{x})$ the number of inputs in the sequence.

The idea of using genetic algorithms to produce test cases have so far only been tested on model examples. A typical result from a simulation is shown in figure 5. Each individual is defined by a number sequence and initiates with an initial population consisting of 100 random generated sequences. As is shown in the figure, the best of these randomly generated sequences has a length of about 520 while covering approximately 70% of the ontology. As the ontology coverage has a higher priority than the sequence

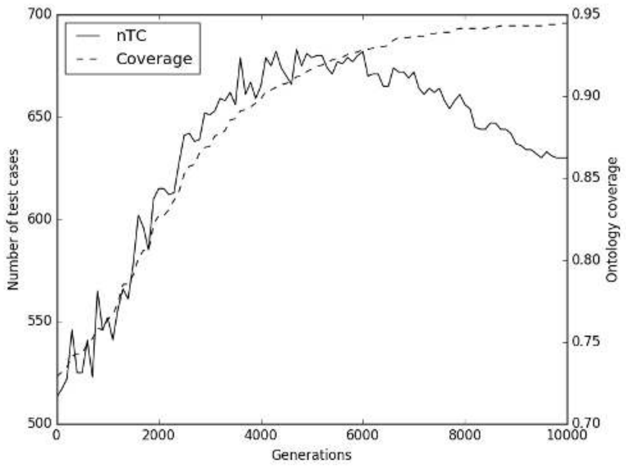

Fig. 5. Results from a genetic algorithm trial. In each generation, three new individuals (number sequences) are generated and evaluated. For each generation, the length of the best individual (nTC=number of test cases) and its coverage is shown.

length in this example, the genetic algorithm initially produces sequences with higher coverage at the expense of greater sequence lengths. However, as the algorithm proceeds and the coverage asymptotically approaches 100%, for each new generation the algorithm finds shorter sequences with equal or slightly better ontology coverage.

5 Evaluation of the Approach

The requirements ontology developed in the OSTAG project is a type of application domain ontology. Any type of application domain model has to be evaluated, to demonstrate its appropriateness for what it was contemplated, and this is also true for the developed requirements ontology. The challenge in ontology evaluation is to determine the quality features that are to be evaluated as well as the evaluation method. Many different ontology evaluation criteria, or features as we call them in this book chapter, have been discussed in literature, e.g. *consistency, completeness, conciseness, expandability* and *sensitiveness* [69], *structural measures, functional measures* and *usability-profiling measures* [70], and *accuracy, adaptability, clarity, completeness, computational efficiency, conciseness, consistency* and *organisational fitness* [71]. Which quality features to evaluate depend on various factors, such as the type of ontology, the focus of an evaluation and the person who is performing the evaluation. Not many tools exist for the handling of different key features but Lantow have presented an on-line platform, OntoMetrics, for the calculation of ontology quality metrics (or features) [72]. For an application domain ontology it is enough to evaluate the features important to the application or domain. In this project we have focused on three specific features, the *usability*, *applicability* and the *correctness* of an ontology. These three features were found to be the most important to use with the requirements ontology for the test case generation in our project.

- We define *ontology usability* as a set of attributes that describe the effort needed by a human to make use of an ontology. The evaluation of the usability is performed by typical potential users of the ontology, normally application domain experts.
- We define *ontology applicability* as the quality of the ontology being correct or appropriate for a particular application domain or purpose. The evaluation of the applicability of the ontology is carried out by a developer of a software component that uses the ontology to implement its functionality.
- We define *ontology correctness* as the degree to which the information asserted in the ontology conforms to the information that should be represented in the ontology. It is about getting accurate information and also accurately documenting the information gathered in an ontology. The evaluation of the correctness is performed by application domain experts.

The three evaluation features are described in continuation. The evaluations of the features were performed on one of the two case studies, provided by the industrial partner coming from the avionics domain. Similar evaluations as the ones presented in the following subsections are planned for the second case study, focusing on AUTOSAR.

Apart from evaluating the inherent quality of the generated requirements ontology itself, the quality of the output when making use of the aforementioned ontology, i.e. the generated test cases, should also be evaluated. To illustrate this evaluation and the results from it, one example of a generated test case is presented in the last subsection.

5.1 Evaluation of the Ontology Usability

The evaluation of the usability of a product or system is something that goes back in time. In 1986, Brooke developed a questionnaire, the System Usability Scale (SUS) [73]. During the years since then, it has been demonstrated that the SUS is applicable over a wide range of systems and types of technology and that it produces similar results as more extensive attitude scales that are intended to provide deeper insights into a users attitude to the usability of a systems. The SUS also has a good ability to discriminate and identify systems with good and poor usability [74]. In the OSTAG project we used a version of the SUS introduced by Casellas [75]. The scale, including its ten questions and the result, is presented in Table 2. The texts in the ten questions have only been slightly modified to adjust to the domain of ontologies.

The evaluation of the usability of an ontology is especially important when the ontology is going to be used by application domain experts who are normally not ontology experts. The ontology was evaluated by two persons, one being an application domain expert (in software testing) and the other being an ontology expert, in order to compare their different views on the usability of the ontology. An evaluation of only two persons will only provide an indication of the usability of the ontology. However, Tullis and Stetson [76] has shown that it is possible to get reliable results with a sample of only 8-12 users. The results from a more extensive evaluation of the usability of the ontology, including four domain experts and one ontology expert, will be published during 2017.

The statements at the odd numbered positions in Table 2 are all in positive form and the even numbered positions are all in negative form, as defined by the SUS. The reason for this alternation is to avoid response biases, especially as the questionnaire invites

Table 2. Ontology usability evaluation. (AE:Application Domain Expert, OE:Ontology Expert, score: 1=strongly disagree, 2=disagree, 3=no preference, 4=agree, 5=strongly agree).

Statements to evaluate the usability of the requirement ontology		AE	OE
1	I think that I could contribute to this ontology	3	5
2	I found the ontology unnecessarily complex	3	2
3	I find the ontology easy to understand	4	4
4	I think that I would need further theoretical support to be able to understand this ontology	2	1
5	I found that various concepts in this system were well integrated	4	5
6	I thought there was too much inconsistency in this ontology	2	2
7	I would imagine that most domain experts would understand this ontology very quickly	4	2
8	I find the ontology very cumbersome to understand	2	2
9	I am confident I understand the conceptualisation of the ontology	4	5
10	I needed to ask a lot of questions before I could understand the conceptualisation of the ontology	2	1

rapid responses by being short; by alternating positive and negative statements, the goal is to have respondents read each statement and make an effort to reflect whether they agree or disagree with it. The intrinsic details related to the calculation of the scoring can be found in [73]. When applying the scoring procedure on the result presented in the table, the SUS scores indicate that the usability result for the application domain expert was 70 while the ontology expert had a result of 82.5. This indicates that the usability of the ontology from the application domain expert's point of view was in the 57 percentile rank while the usability of the ontology from the ontology expert's point of view was in the 92 percentile rank.

5.2 Evaluation of the Ontology Applicability

According to our definition of applicability, the ontology should exhibit the quality of correctness or appropriateness when used for a particular application domain or purpose. To evaluate the applicability of the requirements ontology, it has been used for automatic generation of software test cases based on the requirements in the SRS document. The generation of test cases was done by applying inference rules to the ontology as described in section 4.1. The OWL statements from the requirements ontology were accessed by the inference rules. During the first experiment 66 distinct entities from the ontology were used for the test case construction. The test cases were generated as plain text in English. Our experiment showed an almost one-to-one correspondence between the texts in the generated test cases and the document texts provided by Saab Avionics (see subsection 5.4 for more details).

The evaluation showed that the developed requirements ontology can fulfil its purpose, that is, to support different stages of a software development process. The ontology has been used for the automation of a part of the testing process and allowed for the successful generation of test cases. The ontology allowed for a straightforward way of formulating inference rules. It was fairly easy to integrate the ontology in the OWL functional syntax in the Prolog program containing the inference rules. The OWL expressions were directly employed in the inference rules (after small syntactic changes in the translation phase) thanks to the availability of instances in the ontology. Exploring the ontology paths allowed for the capture of strategies for the test case generation. The minor deficiencies in the ontology that were discovered during the development of inference rules were addressed in subsequent iterations.

5.3 Evaluation of the Ontology Correctness

For the evaluation of the correctness of the ontology, two different tools were used by five evaluators, four industry domain experts and one ontology expert. The first tool that was used was Protégé, to be able to access the information represented in the ontology and compare it with the information written in the SRS document. The second tool that was used was a web-based application developed within the project, a tool that transformed, or *verbalised*, the information represented in the ontology into a natural language text, very much like the original text found in the SRS document. The general purpose of verbalisation tools is to make ontologies more readable for application domain experts [77, 78]. The research focus in the ontology verbalisation domain has been on expressing the axioms in an ontology in natural language (e.g. [79, 80]), or generating a summarized report of an ontology (e.g. [81]). The requirements in the SRS document coming from the avionics case study were well-structured. The verbalisation tool made use of a simple pattern-based algorithm. While simultaneously using Protégé and the verbalisation tool, the experts were asked to validate the correctness of the ontology, comparing it with the texts found in the SRS document. Each of the five experts were assigned 9 or 10 requirements each that they were asked to validate. When evaluating the results, it was striking that, in general, the evaluators used less time but managed to verify more requirements when using the verbalisation tool than if only relying on Protégé. But, as one of the industry domain experts put it, "*Note that 'human understandable' issues that are 'machine impossible' will go undetected* [if relying solely on verbalisation, authors' comments]. *In Protégé, it was possible to see when a requirement was misinterpreted. Here it is back to textual representations* [when using verbalisation, authors' comments] *that may hide the misunderstanding. Easier to read though, but changing format is sometimes good*". More details on the evaluation results will be presented in a publication during 2017.

5.4 Evaluation of the Generated Test Cases

To generate the test cases, a total of 40 inference rules were used. Together, they generated 18 test cases for 15 requirements. The corresponding test cases were reproduced in plain English, approximating the format described in the software test description (STD) document (provided by Saab Avionics). To illustrate the similarity between the two representations, one specific requirement, the SRSRS4YY-435 that was outlined in a previous section, has been chosen. The results from this evaluation can be observed in Table 3 where the text in the left column is a slightly modified excerpt from the STD document while the text in the right column is the generated output, after applying some of the inference rules to the requirements ontology. SRSRS4YY-435 is a requirement that is evaluated in one single test case (while other requirements sometimes need to

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	**Test Inputs:**
1. According to table below.	1. <parity> := min_value - 1 <parity> := max_value + 1 <parity> := 681881
2. <uartId> := <uartId> from the rs4yy_init call	2. <uartID> := <uartID> from the initializationService call
3. <uartId> := <uartId> from the rs4yy_init call	3. <uartID> := <uartID> from the initializationService call
4. <parity> := rs4yy_noneParity	4. <parity> := noneParity
Test Procedure	**Test Procedure:**
1. Call rs4yy_init	1. Call initializationService
2. Call rs4yy_write	2. Call writeService
3. Call rs4yy_read	3. Call readService
4. Recovery: Call rs4yy_init	4. Recovery: Call initializationService
Expected Test Results	**Expected Test Results:**
1. <result> == rs4yy_parityCfgError	1. <result> == parityConfigurationError
2. <result> == rs4yy_notInitialised	2. <result> == rs4yyNotInitialised
3. <result> == rs4yy_notInitialised, <length> == 0	3. <result> == rs4yyNotInitialised, <length> == 0
4. <result> == rs4yy_ok	4. <result> == rs4yyOk
...	...

be tested in more than one test case), in this occasion test case STDRS4YY-133. As can be observed in the table, there is an almost one-to-one correspondence between the texts in the two columns, something that was positively commented on by the people at Saab Avionics. Even more so, on some occasions the generated test case texts indicated a discrepancy to the corresponding test case texts found in the STD document. These discrepancies were presented to and evaluated by personnel from Saab Avionics and on occasions, the observed discrepancies indicated a detected error in the STD document. Hence, this evaluation of the correctness of the generated test cases helped improving the overall quality of the STD document.

6 Conclusions

In this book chapter we have presented a method to convert software requirements to a formal model in the form of a (requirements) ontology. Once the ontology has been produced, inference rules can be applied to it to produce software test cases. The first experiments indicated that, by using 40 inference rules, 18 test cases for 15 requirements could be generated as plain text in English. The examination of the results showed an almost one-to-one correspondence between the texts in the generated test cases and the manually produced texts provided by Saab Avionics. As the conducted experiments were limited in size, additional experiments with an increased number of inference rules are required to demonstrate the full potential of the process.

The test case generation process has demonstrated that the quality of the generated test cases improved. Minor errors that had gone undetected by the human test case designers were identified and corrected. This result highlights the benefits of automating the test case generation process. In the near future, additional types of quality metrics are going to be evaluated, such as the time savings from automating the test case generation process through real-life time studies, and the coverage of the requirements in the ontology to demonstrate that all requirements in the requirements specification document have been considered and tested.

7 Future Work

The ontologies have so far been developed manually in the OSTAG project, just as a proof of concept of the initial ideas. However, manual ontology development is not an easy or trivial task. It is a time-consuming, expensive, error-prone and labor-intensive activity. During the execution of the OSTAG project, ontology and domain experts manually processed all the information and participated throughout the manual ontology development process, therefore making it a fairly expensive process. Hence, it is essential to be able to (semi-)automatically create ontologies in order to save time and resources. Ontology learning is the research field that deals with the (semi-)automatic construction of ontologies. In the near future, we will investigate how ontologies can be (semi-)automatically developed from an SRS document. Different approaches and tools will be evaluated for this purpose, such as natural language processing and machine learning techniques.

Preliminary studies have showed that different levels of requirements are written in different ways, e.g. low-level requirements are structured while high-level requirements are not. The avionics case study relies on low-level requirements while the AUTOSAR case study relies on high-level requirements. The high-level requirements in the AUTOSAR standard are documented in a less structured, more complex text format as compared to the avionics case study. The created ontology development process is not capable of processing these different types of requirement in an optimal way. Hence, we will focus our investigation on how to extract and annotate the meaning of an AUTOSAR requirement in the best way, and how to represent the meaning of an AUTOSAR requirement to efficiently support the test case generation methods. A preliminary study [82] has indicated that exploiting standard ontological resources for natural language processing, such as FrameNet [83], could be a solution of the first task. A lightweight ontology, or embedding rich meta-data within requirements documents, could be a solution of the second task. Yet another approach could be to limit the complexity and the expressiveness that can be found in several requirements, such as in AUTOSAR, by relying on requirement boilerplates (on a syntactic level) or requirement patterns (on a semantic level) when writing the requirements [84–86]. The third important task is *how to acquire and represent test case generation strategies in a more general way*. One approach to tackle this problem is to introduce several layers of inference rules to make the knowledge more modular and to gather the case specific knowledge in one layer. Another approach is to use algorithms instead of a set of inference rules. An algorithm would allow for representation of more general strategies for test case generation. Further studies will have to be carried out to implement and evaluate the suggested approaches.

Within the research field of ontology development is situated the crucial activity of building high quality ontologies. The challenge is the difficulty of defining quality, and currently there exists no common definition of ontology quality, something that was elaborated on in a previous section. Different quality characteristics can be defined, depending on the scope and purpose of an ontology. In this paper we proposed usability, applicability and correctness as three key quality features of a requirements ontology in relation to the verification and validation of the ontology. The ontology correctness was evaluated using Protégé and a verbalisation tool. To extend the usefulness of the verbalisation tool, we plan to generalize the verbalisation process in such a way that the tool can be used to evaluate other types of ontologies. To provide good tools for ontology quality evaluation or automate the process is something that is important for the effectiveness and efficiency of ontology development. The focus of our future work is on the investigation of practical methods and tools for the evaluation of additional quality features, apart from the three features presented previously.

With regard to the use of genetic algorithms, we have come to the following conclusion; the goal is to cover the ontology, meaning among other things that all requirements are covered. A genetic algorithm produces test cases using random methods, and each randomly created test case must be evaluated. The evaluation of a test case includes an investigation of every ontology instance to see whether this instance is covered or not by the test case. This investigation can often require an amount of work comparable to the work required to manually construct a test case for each of the instances. In such cases, the use of genetic algorithms brings about no substantial value. However, for large programs, where the same input data can cover a large number of instances in non-predictable and non-obvious ways, or when the same input data that tests one instance can serve as prerequisites for testing other instances, genetic algorithms might help reducing the number of test cases required to cover the ontology.

Acknowledgments. The research reported in this book chapter has been carried out in the project ”Ontology-based Software Test Case Generation”, and was financed by grant KKS-20140170 from the Knowledge Foundation in Sweden.

References

1. Gartner worldwide IT spending forecast 2017, 1st quarter 2017 (2017) http://www.gartner.com/newsroom/id/3568917. Accessed July 7th, 2017.
2. CapGemini, Sogeti: World Quality Report 2016-17 (2016) 80 pages.
3. Judge Business School, Cambridge University: Cambridge university study states software bugs cost economy $312 billion per year (2013) http://markets.financialcontent.com/stocks/news/read/23147130/Cambridge_U-niversity_Study_States_Software_Bugs_Cost_Economy_$312_Billion_Per_Year. Accessed: July 7th, 2017.

4. homepage, T.: (http://www.testing-whiz.com) Accessed: July 7th, 2017.
5. homepage, H.U.: (https://saas.hpe.com/en-us/software/uft) Accessed: July 7th, 2017.
6. Enoiu, E., Sundmark, D., Čaušević, A., Pettersson, P.: A comparative study of manual and automated testing for industrial control software. In: Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on, IEEE (2017) 412–417
7. de Santiago Júnior, V.A., Vijaykumar, N.L., Guimarães, D., Amaral, A.S., Ferreira, É.: An environment for automated test case generation from statechart-based and finite state machine-based behavioral models. In: Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on, IEEE (2008) 63–72
8. Harel, D.: Statecharts: a visual formalism for complex systems. Science of computer programming 8 (1987) 231–274
9. MacMinn, P.: Search-based software test data generation: a survey. Software Testing, Verification and Reliability 14 (2004) 105–156
10. Harman, M.: The current state and future of search-based software engineering. 2007 Future of Software Engineering (2007) 342–357
11. Armour, P.G.: Software: hard data. Communications of the ACM 49 (2006) 15–17
12. de Souza, É.F., de Almeida Falbo, R., Vijaykumar, N.L.: Knowledge management initiatives in software testing: a mapping study. Information and Software Technology 57 (2015) 378–391
13. Souza, É.F.d., Falbo, R.d.A., Vijaykumar, N.L.: ROoST: Reference Ontology on Software Testing. Applied Ontology 12 (2017) 59–90
14. Studer, R., Benjamins, V.R., Fensel, D.: Knowledge engineering: principles and methods. Data & Knowledge Engineering 25 (1998) 161–197
15. Gruber, T.R.: Toward principles for the design of ontologies used for knowledge sharing. International Journal of Human-Computer Studies 43 (1995) 907–928
16. Borst, W.N.: Construction of Engineering Ontologies for Knowledge Sharing and Reuse. CTIT Ph.D.-thesis series No. 97-14. Universiteit Twente, Enschede (1997)
17. Happel, H.J., Seedorf, S.: Applications of ontologies in software engineering. In: Proceedings of Workshop on Semantic Web Enabled Software Engineering (SWESE) on the ISWC, Citeseer (2006) 5–9
18. Ruy, F.B., de Almeida Falbo, R., Barcellos, M.P., Costa, S.D., Guizzardi, G.: SEON: a software engineering ontology network. In: Knowledge Engineering and Knowledge Management: 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings 20, Springer (2016) 527–542
19. Wang, Y., Bai, X., Li, J., Huang, R.: Ontology-based test case generation for testing Web services. In: Proceedings of Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07), IEEE (2007) 43–50
20. Nguyen, C.D., Perini, A., Tonella, P.: Ontology-based test generation for multiagent systems. In: Proceedings of 7th International Joint Conference on Autonomous Agents and Multiagent Systems. Volume 3. (2008) 1315–1320
21. Sneed, H.M., Verhoef, C.: Natural language requirement specification for Web service testing. In: Web Systems Evolution (WSE), 2013 15th IEEE International Symposium on, IEEE (2013) 5–14
22. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: Proceedings of Conference on the Future of Software Engineering, ACM (2000) 35–46
23. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos: representing knowledge about information systems. ACM Transactions on Information Systems (TOIS) 8 (1990) 325–362
24. Greenspan, S., Mylopoulos, J., Borgida, A.: On formal requirements modeling languages: RML revisited. In: Proceedings of 16th international conference on Software engineering, IEEE Computer Society Press (1994) 135–147

25. Uschold, M., Gruninger, M.: Ontologies: principles, methods and applications. The Knowledge Engineering Review 11 (1996) 93–136
26. Dobson, G., Sawyer, P.: Revisiting ontology-based requirements engineering in the age of the semantic Web. In: Proceedings of International Seminar on Dependable Requirements Engineering of Computerised Systems at NPPs. (2006) 27–29
27. RTCA: Software Considerations in Airborne Systems and Equipment Certification. (1992)
28. : AUTOSAR homepage (2017) https://www.autosar.org/. Accessed July 7th, 2017.
29. Mussa, M., Ouchani, S., Al Sammane, W., Hamou-Lhadj, A.: A survey of model-driven testing techniques. In: Proceedings of 8th International Symposium on 9th International Conference on Quality Software (QSIC '09). (2009) 167–172 24-25 August 2009, Jeju, South Korea.
30. Nguyen, C.D., Marchetto, A., Tonella, P.: Combining model-based and combinatorial testing for effective test case generation. In: Proceedings of 2012 International Symposium on Software Testing and Analysis, ACM (2012) 100–110
31. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey on automated software test case generation. Journal of Systems and Software 86 (2013) 1978–2001
32. Kaur, A., Vig, V.: Systematic review of automatic test case generation by UML diagrams. International Journal of Engineering Research & Technology (IJERT) 1 (2012) 17 pages.
33. Enoiu, E.P., Čaušević, A., Ostrand, T.J., Weyuker, E.J., Sundmark, D., Pettersson, P.: Automated test generation using model-checking: an industrial evaluation. International Journal on Software Tools for Technology Transfer 18 (2016) 335–353
34. Holt, N.E., Briand, L.C., Torkar, R.: Empirical evaluations on the cost-effectiveness of state-based testing: an industrial case study. Information and Software Technology 56 (2014) 890–910
35. Pinheiro, A.C., Simão, A., Ambrosio, A.M.: FSM-based test case generation methods applied to test the communication software on board the itasat university satellite: A case study. Journal of Aerospace Technology and Management 6 (2014) 447–461
36. Sneed, H.M.: Testing against natural language requirements. In: Quality Software, 2007. QSIC'07. Seventh International Conference on, IEEE (2007) 380–387
37. Demuth, B., Sneed, H.M.: Ein Modell für natursprachliche Anforderungsdokumente. Informatik-Spektrum 39 (2016) 362–372 in German.
38. de Santiago Júnior, V.A.: SOLIMVA: A methodology for generating model-based test cases from natural language requirements and detecting incompleteness in software specifications, 264 p. PhD thesis, Thesis (Doctorate at Post Graduation Course in Applied Computing)National Institute for Space Research, São José dos Campos, SP, Brazil (2011)
39. de Santiago Júnior, V.A., Vijaykumar, N.L.: Generating model-based test cases from natural language requirements for space application software. Software Quality Journal 20 (2012) 77–143
40. De Souza, É.F., de Santiago Júnior, V.A., Vijaykumar, N.L.: H-Switch Cover: a new test criterion to generate test case from finite state machines. Software Quality Journal 25 (2017) 373–405
41. Lin, J., Fox, M.S., Bilgic, T.: A requirement ontology for engineering design. Concurrent Engineering 4 (1996) 279–291
42. Mayank, V., Kositsyna, N., Austin, M.: Requirements engineering and the semantic Web, part II. representaion, management, and validation of requirements and system-level architectures. Technical report, University of Maryland (2004)
43. Siegemund, K., Thomas, E.J., Zhao, Y., Pan, J., Assmann, U.: Towards ontology-driven requirements engineering. In: Proceedings of Workshop semantic web enabled software engineering at 10th international semantic web conference (ISWC), Bonn. (2011) 14 pages.

44. Moroi, T., Yoshiura, N., Suzuki, S.: Conversion of software specifications in natural languages into ontologies for reasoning. In: Proceedings of 8th International Workshop on Semantic Web Enabled Software Engineering (SWESE'2012). (2012) 15 pages.
45. Papadakis, N., Stravoskoufos, K., Baratis, E., Petrakis, E.G., Plexousakis, D.: PROTON: a Prolog reasoner for temporal ontologies in OWL. Expert Systems with Applications 38 (2011) 14660–14667
46. Laera, L., Tamma, V.A., Bench-Capon, T., Semeraro, G.: SweetProlog: A system to integrate ontologies and rules. In: Proceedings of 3rd RuleML workshop Rules and Rule Markup Languages for the Semantic Web. Volume 3323 of LNCS., Springer (2004) 188–193
47. Fogel, L.J., Owens, A.J., Walsh, M.J.: Artificial intelligence through simulated evolution. (1966)
48. Rechenberg, I.: Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution. Frommann-Holzbog, Stuttgart, 1973. Step-Size Adaptation Based on Non-Local Use of Selection Information. In Parallel Problem Solving from Nature (PPSN3) (1994)
49. Holland, J.H.: Outline for a logical theory of adaptive systems. Journal of the ACM (JACM) 9 (1962) 297–314
50. DeMillo, R.A., Offutt, A.J.: Experimental results from an automatic test case generator. ACM Transactions on Software Engineering and Methodology (TOSEM) 2 (1993) 109–127
51. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithms. Software Testing Verification and Reliability 9 (1999) 263–282
52. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary test environment for automatic structural testing. Information and Software Technology 43 (2001) 841–854
53. Sthamer, H., Wegener, J., Baresel, A.: Using evolutionary testing to improve efficiency and quality in software testing. In: Proceedings of 2nd Asia-Pacific Conference on Software Testing Analysis & Review. (2002)
54. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37 (2011) 649–678
55. Mahajan, M., Kumar, S., Porwal, R.: Applying genetic algorithm to increase the efficiency of a data flow-based test data generation approach. ACM SIGSOFT Software Engineering Notes 37 (2012) 1–5
56. Tan, H., Muhammad, I., Tarasov, V., Adlemo, A., Johansson, M.: Development and evaluation of a software requirements ontology. In: Proceedings of 7th International Workshop on Software Knowledge-SKY 2016, in conjunction with the 8th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management-IC3K 2016, SCITEPRESS (2016) 11–18
57. Noy, N.F., McGuinness, D.L.: Ontology development 101: a guide to creating your first ontology. Technical report, Stanford Knowledge Systems Laboratory (2001) KSL-01-05.
58. Gennari, J.H., Musen, M.A., Fergerson, R.W., Grosso, W.E., Crubézy, M., Eriksson, H., Noy, N.F., Tu, S.W.: The evolution of Protégé: an environment for knowledge-based systems development. International Journal of Human-computer studies 58 (2003) 89–123
59. Fernández-López, M., Gómez-Pérez, A., Juristo, N.: METHONTOLOGY: from ontological art towards ontological engineering. In: Proceedings of AAAI97 Spring Symposium, American Asociation for Artificial Intelligence (1997)
60. Fensel, D., Van Harmelen, F., Klein, M., Akkermans, H., Broekstra, J., Fluit, C., van der Meer, J., Schnurr, H.P., Studer, R., Hughes, J.: On-To-Knowledge: ontology-based tools for knowledge management. In: Proceedings of eBusiness and eWork. (2000) 18–20
61. Presutti, V., Daga, E., Gangemi, A., Blomqvist, E.: eXtreme design with content ontology design patterns. In: Proceedings of 2009 International Conference on Ontology Patterns-Volume 516, CEUR-WS.org (2009) 83–97

62. Shearer, R., Motik, B., Horrocks, I.: HermiT: a highly-efficient OWL reasoner. In: Proceedings of 5th Int. Workshop on OWL: Experiences and Directions. Volume 432. (2008) 91 pages.
63. Bechhofer, S.: OWL: Web ontology language. In: Encyclopedia of Database Systems. Springer (2009) 2008–2009
64. Tarasov, V., Tan, H., Ismail, M., Adlemo, A., Johansson, M.: Application of inference rules to a software requirements ontology to generate software test cases. In: OWL: Experiences and Directions–Reasoner Evaluation: 13th International Workshop, OWLED 2016, and 5th International Workshop, ORE 2016, Bologna, Italy, November 20, 2016, Revised Selected Papers. Volume 10161., Springer (2017) 82
65. Nasser, V.H., Du, W., MacIsaac, D.T.: Knowledge-based software test generation. In: Proceedings of 21st International Conference on Software Engineering and Knowledge Engineering, Boston, U.S.A. (2009) 312–317
66. Freitas, A., Vieira, R.: An ontology for guiding performance testing. In: Proceedings of 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT). (2014) 400–407
67. Bratko, I.: Prolog Programming for Artificial Intelligence. 4th edn. Pearson Education (2011)
68. Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3C. 2nd edn. (2012)
69. Gómez-Pérez, A.: Ontology evaluation. In: Handbook on Ontologies. Springer (2004) 251–273
70. Gangemi, A., Catenacci, C., Ciaramita, M., Lehmann, J.: Modelling ontology evaluation and validation. In: Proceedings of European Semantic Web Conference, Springer (2006) 140–154
71. Vrandečić, D.: Ontology evaluation. In: Handbook on Ontologies. Springer (2009) 293–313
72. Lantow, B.: OntoMetrics: putting metrics into use for ontology evaluation. In: Proceedings of 8th International Joint Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management-IC3K 2016, Volume 2, KEOD, SCITEPRESS (2016) 186–191
73. Brooke, J.: SUS-a quick and dirty usability scale. Usability Evaluation in Industry 189 (1996) 4–7
74. Brooke, J.: SUS: a retrospective. Journal of Usability Studies 8 (2013) 29–40
75. Casellas, N.: Ontology evaluation through usability measures. In: Proceedings of OTM Confederated International Conferences "On the Move to Meaningful Internet Systems", Springer (2009) 594–603
76. Tullis, T.S., Stetson, J.N.: A comparison of questionnaires for assessing website usability. In: Proceedings of Usability Professional Association Conference, Citeseer (2004) 1–12
77. Jarrar, M., Keet, M., Dongilli, P.: Multilingual verbalization of ORM conceptual models and axiomatized ontologies. Technical report, Vrije Unversiteit Brüssel (2014) 13 pages.
78. Keet, C.M., Chirema, T.: A model for verbalising relations with roles in multiple languages. In: Knowledge Engineering and Knowledge Management: 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings 20, Springer (2016) 384–399
79. Kaljurand, K., Fuchs, N.E.: Verbalizing OWL in Attempto controlled English. In: Proceedings of OWLED'07. Volume 258. (2007) 10 pages.
80. Kop, C.: How to summarize an OWL domain ontology. In: Proceedings of Fourth International Conference on Digital Society (ICDS'10), IEEE (2010) 106–111
81. Liang, S.F., Stevens, R., Scott, D., Rector, A.: Automatic verbalisation of SNOMED classes using OntoVerbal. In: Proceedings of Conference on Artificial Intelligence in Medicine in Europe, Springer (2011) 338–342

82. Zimmermann, O.: Modelling complex software requirements with ontologies. Master's thesis, Universität Rostock (2017)
83. Baker, C.F., Fillmore, C.J., Lowe, J.B.: The Berkeley FrameNet project. In: Proceedings of 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1, Association for Computational Linguistics (1998) 86–90
84. Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Zojer, H., Panis, C.: DODT: increasing requirements formalism using domain ontologies for improved embedded systems development. In: Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2011 IEEE 14th International Symposium on, IEEE (2011) 271–274
85. Arora, C., Sabetzadeh, M., Briand, L.C., Zimmer, F.: Requirement boilerplates: transition from manually-enforced to automatically-verifiable natural language patterns. In: Requirements Patterns (RePa), 2014 IEEE 4th International Workshop on, IEEE (2014) 1–8
86. Böschen, M., Bogusch, R., Fraga, A., Rudat, C.: Bridging the gap between natural language requirements and formal specifications. In: Joint Proceedings of REFSQ-2016 Workshops, Doctoral Symposium, Research Method Track, and Poster Track (REFSQ-JP 2016). CEUR Workshop Proceedings, CEUR-WS (2016) 1–11