Automation of Software Testing Process using Ontologies

Vladimir Tarasov, He Tan and Anders Adlemo
School of Engineering, Jönköping University, Jönköping, Sweden

Keywords: Knowledge Representation, Ontologies, Inference Rules, Test Case Generation, Automated Testing.

Abstract: Testing of a software system is a resource-consuming activity that requires high-level expert knowledge. Methods based on knowledge representation and reasoning can alleviate this problem. This paper presents an approach to enhance the automation of the testing process using ontologies and inference rules. The approach takes software requirements specifications written in structured text documents as input and produces the output, i.e. test scripts. The approach makes use of ontologies to deal with the knowledge embodied in requirements specifications and to represent the desired structure of test cases, as well as makes use of a set of inference rules to represent strategies for deriving test cases. The implementation of the approach, in the context of an industrial case, proves the validity of the overall approach.

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 3.2% in 2019, to reach a total of USD 3.8 trillion (Gartner, 2018). As software products and systems permeate every aspect of our lives, we become more and more dependent on their correct functioning. Consequently, quality concerns become much more vital and critical as end-users get more dependent on software products and services. 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 all these efforts, errors have an annoying tendency to remain undetected in the code. Accordingly to Capgemini World Quality Report 2018-19 (CapGemini et al., 2018), the budget allocation for quality assurance and testing, as percentage of IT expenditures in the software industry, has come down in recent years but still accounted for 26% in 2018.

One way to reduce the cost related to software testing has been to automate as many of the aforementioned activities as possible. As far as test management and test script execution go, this is a mature field where commercial products assist software testers in their daily work, like TestingWhiz (TestingWhiz, 2018) or HPE Unified Functional Testing (HPE-UFT, 2018). 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%) (Enoiu et al., 2017).

Despite successful achievements in automation of script execution and white-box testing, there is still a lack of automation of black-box testing of functional requirements. Because such tests are mostly created manually, which requires high-level human expertise, modern methods from the area of knowledge engineering are up to the challenge.

This paper proposes an approach to automate software testing by modelling the testing body of knowledge with formal ontologies and reasoning with inference rules to generate test cases[1]. The main contribution of this paper is to demonstrate that the use of ontologies allows for automation of the full process of software testing, from the capture of domain knowledge in software requirements specifications (SRS) to the generation of software test cases. The proposed framework combines OWL ontologies, representing the knowledge from SRS and test specifications, with inference rules, representing the assembled knowledge of expert software testers and testing documents, and with ontological representation of generated test cases.

The rest of this paper is structured as follows. Section 2 describes the related work. The automated testing process framework is presented in Section 3. Section 4 presents an implementation of the framework for a testing task in the avionics industry. It details

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

Tarasov, V., Tan, H. and Adlemo, A.
Automation of Software Testing Process using Ontologies.
DOI: 10.5220/0008054000570066
In *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2019)*, pages 57-66
ISBN: 978-989-758-382-7
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

the proposed approach and implementation for each step in the framework, including ontologies providing the meta-data for software requirements and test cases, inference rules capturing strategies for test case generation, and the test scripts generation strategy. Section 5 presents the evaluations of the implemented approach. The conclusions of the study are given in Section 6.

2 RELATED WORK

Until recently, the use of ontologies in software testing has been one of the least explored areas of software engineering (Ruy et al., 2016) and has thus not been discussed as much as their use in other stages of the software life-cycle process. In (Happel and Seedorf, 2006), 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. (Wang et al., 2007; Nguyen et al., 2008; Sneed and Verhoef, 2013).

One mature area where ontologies have been successfully applied is requirements engineering. There exists a clear synergy effect between the ontological modelling of domain knowledge and the modelling of requirements performed by requirement engineers (Dobson and Sawyer, 2006). Recently, a renewed interest in utilizing ontologies in requirements engineering has surged due to the appearance of semantic web technologies (Happel and Seedorf, 2006; Dobson and Sawyer, 2006). Most of the research deals with inconsistency and incompleteness problems in requirement specifications through reasoning over requirements ontologies.

The ontologies, presented in this paper, support an automated testing process. First, a requirements ontology provides SRS's meta data and supports extracting information from requirements documents. Second, a test case ontology describes the desired structure of test case descriptions and is used to guide the test case generation process. In the end, the populated test case ontology supports the test scripts generation.

3 A FRAMEWORK FOR TESTING PROCESS AUTOMATION USING ONTOLOGIES

A typical process of black-box testing of functional software requirements comprises two activities. During the first activity, software testers design test cases based on SRS and their own expertise from previous work on testing software systems. The second activity is to develop test scripts. Finally, the tests are carried out, either manually or using a test execution tool, based on the automated execution of test scripts. In this paper we present a framework to automate such a testing process using ontologies. The framework is shown in Fig. 1.

The requirements are often described in well-structured or semi-structured textual documents. First, a requirements ontology is built to represent the structure of software requirements. With the help of the ontology, the requirements information is extracted from the text documents and then used to populate the ontology. The populated ontology serves as an input for the test case generator.

In situations where the testers' expertise is less structured, the information is acquired through interviews with experienced testers and examination of existing software test description (STD) documents. The acquired testing strategies are represented with inference rules. The rules are built on top of ontologies, following the W3C recommendation on a layered infrastructure of Semantic Web. The advantage is that the ontologies can be reused for different applications when requirements specifications and test cases are expressed in a similar way. The rules utilize the populated requirements ontology for checking conditions and querying data to generate test cases.

Furthermore, a test case ontology is provided to specify what should be contained in test cases and how each test case should be structured. The test case ontology is used in the step of test case generation. The ontology is populated when test cases are generated. Finally, the populated test case ontology is employed to generate test scripts.

4 IMPLEMENTATION OF THE FRAMEWORK

In this section we demonstrate an implementation of the framework. The implementation is to support testing of the components of an embedded sub-system within an avionic system. The case data were provided by the fighter aircraft developer, Saab Avion-

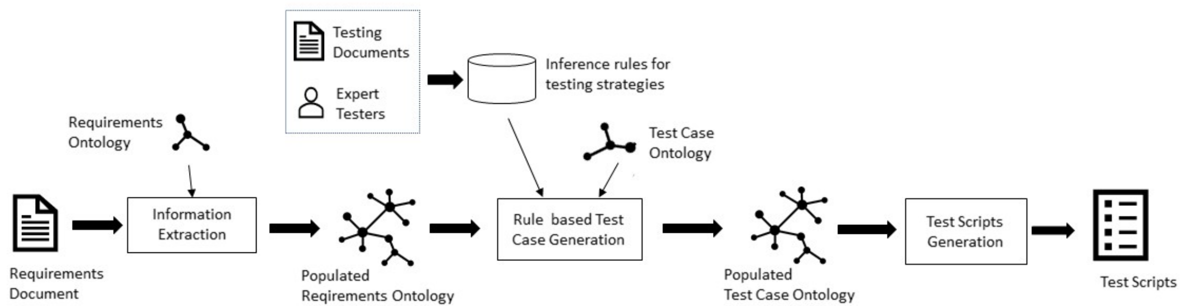

Figure 1: A framework for automation of testing process using ontologies.

ics. 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 DO178B (RTCA, 1992). The requirements of the entire system, or units making up the system, must be analyzed, specified and validated before initiating the design and implementation phases. The software test cases are manually created and also have to be manually inspected. The requirements and test cases that were used in the framework implementation were provided in text documents and the results were manually validated by avionic industry domain experts.

4.1 Requirements and Test Case Ontologies

Although the requirements are written in natural language, the requirements documents are well structured. The *requirements ontology* (see Fig. 2) was built by ontology experts based on the structure of the textual documents provided by Saab Avionics. Each requirement has an unique ID and consists of at least:

1. Requirement parameters, which are inputs of a requirement,
2. Requirement conditions,
3. Results, which are usually outputs of a requirement and exception messages.

Some requirements require the system to take actions.

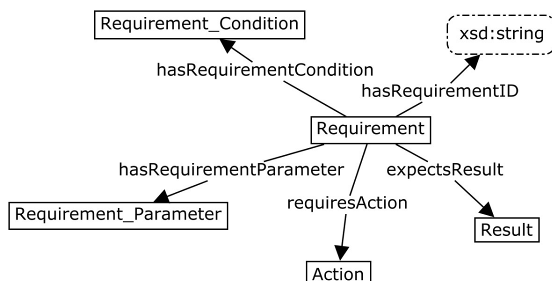

Figure 2: The key entities in the *requirements* ontology.

More details about the ontology can be found in (Tan et al., 2016).

The *test case ontology* (see Fig. 3) was also built by ontology experts based on the structure of test case descriptions created by the industrial partner. Each test case has a unique ID, addresses one requirement, and has prerequisite conditions. There is a list of ordered steps needed to be followed in each test case. Each step consists of input, procedure and expected result. Therefore, the classes `Test_Input`, `Test_Procedure` and `Test_Results` are defined as the subclasses of OWLList (Drummond et al., 2006), which is a standard representation of a sequence in OWL-DL.

4.2 Population of the Requirements Ontology from the Requirements Documents

Fig. 4 shows a fragment of the populated ontology for one particular functional requirement, SRSRS4YY-431. The requirement states that 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 Fig. 4, the rectangles repre-

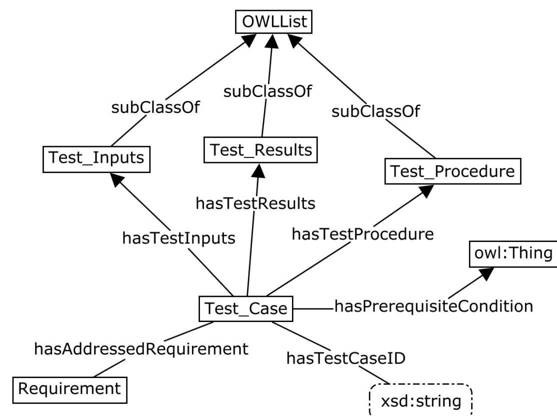

Figure 3: The key elements in the *test case* ontology.

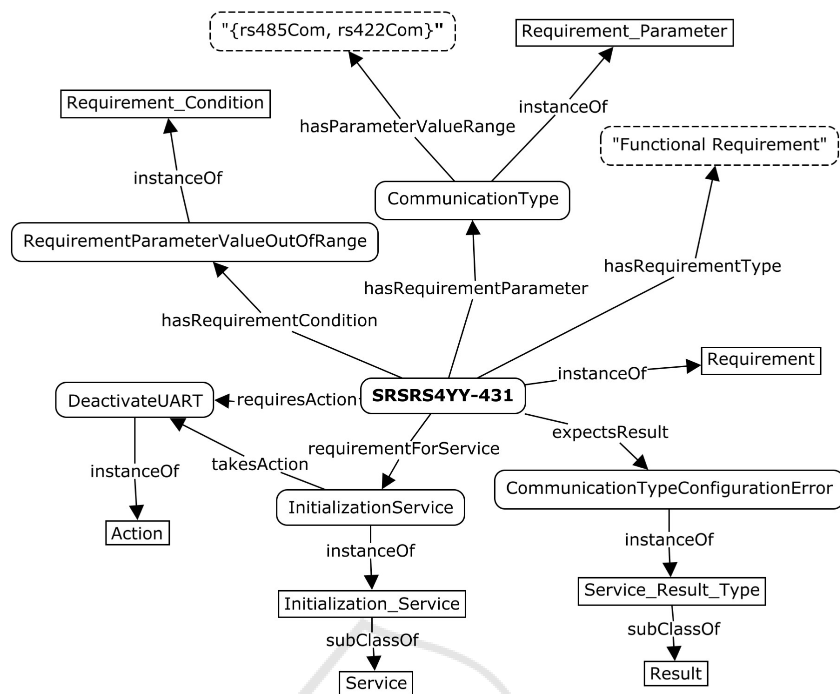

Figure 4: Ontology fragment for the SRSRS4YY-431 requirement specification.

sent the concepts of the ontology; the rounded rectangles represent the individuals (instances); and the dashed rectangles provide the data values of datatype property for individuals.

The populated ontology contains 147 individuals in total. In the implementation presented in this paper, the information was manually extracted from the requirements document and used to populate the ontology. To fully automate the testing process, the population can be accomplished automatically using different methods (Gangemi et al., 2017; Petasis et al., 2011). For the well structured requirement specifications provided by Saab Avionics, we have proposed an approach based on BNF grammar as the solution (Ismail, 2016).

4.3 Approach to Test Case Generation based on Inference Rules

Test cases are generated through deriving information from the populated requirements ontology with the help of inference rules. The Prolog programming language (Sterling and Shapiro, 1994) was chosen for coding of inference rules because (1) it has both built-in inference engine and standard programming facilities, (2) it has means to represent inference rules in a natural way, and (3) it has means to access the entities in the ontology. The popular tool for business rules, DROOLS, was not selected as a solution because of its flaws and deficiencies, as reported in (Kaczor et al., 2011) and (Ostermayer and Seipel, 2012)

The particular Prolog system used for the implementation was SWI-Prolog (Wielemaker et al., 2012). The first necessary task is to translate the ontology into the Prolog syntax to prepare the ontology for the use by the inference rules. 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 was selected as it has the closest resemblance to the Prolog syntax.

An ontology document in the functional-style syntax is a number of prefix and import definitions followed by a sequence of OWL constructs enclosed in the Ontology statement (Motik et al., 2012). In turn, 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 (Sterling and Shapiro, 1994).

In an ontology document in the functional-style syntax, every construct is represented by one statement that resides on a single line. During the OWL-to-Prolog translation, a Python script processes the ontology document line by line to translate each line into the corresponding Prolog statement. The following steps are carried out for every OWL statement:

- Read an OWL statement,
- Tokenize the statement and convert each token into lowerCamelCase notation because Prolog atoms start with a lower case letter,
- Convert the list of tokens into a Prolog clause in the form of a fact, encoding OWL prefixes as Prolog terms.

The translated OWL statements are annotations, declarations, and axioms including assertions on properties, classes, restrictions and facts. An example of a translation of an OWL class assertion (statement #1) into a corresponding Prolog clause (statement #2) is given below.

```
1: ObjectPropertyDomain(:hasPart owl:Thing)
2: objectPropertyDomain(hasPart, owl(thing)).
```

The second necessary task to solve to derive test cases from the requirements ontology is to represent testers expertise on how they use requirements to create test cases. Such expertise embodies inherent strategies for test case creation, knowledge that can be expressed in the form of heuristics represented as if-then rules. This kind of knowledge is acquired from two sources. First, literature on software testing contains few general guidelines, e.g. boundary value testing. These general test strategies apply to all domains. Second, expert testers were interviewed to capture their expertise that is specific to particular types of software systems and/or particular domains. Such testing knowledge needs to be acquired for each domain type. Additionally, existing test cases and their corresponding requirements were examined and analysed. The details of the test case generation with inference rules are provided in (Tarasov et al., 2017).

Each requirement describes some functionality of a service (function) from a driver for a hardware unit. The latter is represented by an embedded avionic system component. All the requirements are grouped into services. The analysis process included the comparison of an original test case, previously created manually by a software tester, with the corresponding requirement. The analysis goal was to fully understand how different parts of the original test case had been constructed. After that, a number of discussions with the industry software testers participating in the project were arranged to clear any inconsistencies or remaining doubts.

As a result, a set of inference rules were formulated in plain English. The test cases provided by the industrial partner are structured as follows: prerequisite conditions, test inputs, test procedure, and expected test results. For each of the test case parts, a group of inference rules were formulated. The example below shows an inference rule for the test procedure part of the requirement SRSRS4YY-431:

```
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 using the ontology instances (individuals) representing the requirement and connected hardware parts, input/output parameters for the service and the like. The instructions for generating a test case part are expressed in the action part (then-part) of the rule.

The acquired inference rules are implemented in the Prolog programming language with the help of Prolog rules (a Prolog rule is analogous to a statement in other programming languages). After the OWL-to-Prolog translation, the ontology can be loaded as part of the Prolog program, and the ontology entities can be directly accessed by the Prolog code. 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:

```
% construct TC procedure
1 tc_procedure(Requirement, [Service,
        WriteService, ReadService,
        recovery(Service)]) :-
   %check condition for calls #2-4
2  action(Requirement, deactivateUART),
   %get service individual for calls #1,4
3  service(Requirement, Service),
   %get individuals of the required services
4  type(WriteService, transmission_service),
5  type(ReadService, reception_service).
  %check the required action
6 action(Requirement, Action) :-
   objectPropertyAssertion(requiresAction,
       Requirement, Action).
  %retrieve the service of a requirement
7 service(Requirement, Service) :-
   objectPropertyAssertion(
       requirementForService,
       Requirement, Service).
  %check the type of an instance
8 type(Individual, Class) :-
   classAssertion(Class, Individual).
```

Line 1 in the example is the head of the rule consisting of the name, input argument and output argument, which is the constructed procedure as a Prolog list. The list is constructed from the retrieved ontology entities and special term functors. Line 2 encodes the condition of the heuristic. Lines 3-5 are the queries to retrieve the relevant entities from the ontology. The predicates 6-8 are helper ones that perform

the actual retrieval of the required entities from the Requirements Ontology.

Each test case is generated sequentially, from the prerequisites part to the results part. The generated parts are collected into one structure (Prolog term).

4.4 Population of the Test Case Ontology

During the generation phase, all created test cases are stored in the Prolog working memory as a list. When the test case list is complete, the next phase of ontology population starts. The test cases in the list are processed consecutively. For each test case, an instance is created with object properties relating it to the addressed requirement and test case parts. After that, instances with object properties are created for the four test case parts: prerequisites, test inputs, test procedure and expected test results. If the parts contain several elements, OWL lists are used for representation.

The main Prolog predicate, performing the ontology population, is shown in Fig 5. It utilizes two predicates from the ontology population layer: `ontology_comment` and `ontology_assertion`. The former is used to insert auxiliary comments in the ontology. The latter asserts OWL axioms representing test case elements in the test case ontology. The four predicates that start with "`populate_w_`" create OWL statements for the test case parts by processing lists associated with each part. Each of these predicates is passed the name prefix and initial number to construct instances of an OWL list representing this test case part.

Finally, the newly asserted axioms are serialized in the ontology source file by the ontology serialization layer. It takes care of translating the OWL assertions from the Prolog syntax into the OWL functional-style syntax with the help of a definite clause grammar.

4.5 Test Scripts Generation

In order to carry out testing, test cases need to be transformed into executable procedures. Such procedures are usually programs written in a language like Python or C. However, our project partner, Saab Avionics, follows a strict quality assurance process. According to their process, all test cases have to be thoroughly inspected as plain text by the quality assurance team before they are signed of for actual execution. For this reason, test cases were translated into plain text in English in the OSTAG project. The translation into plain text is implemented by the process of verbalization of the test case ontology. The verbalization starts with the test case instance and then iterates through the OWL lists representing the four test case parts. In a similar way, the test cases from the test case ontology can be transformed into actual executable procedures in a programming language.

In the OSTAG project, the 37 test cases were translated into plain text descriptions according to the software test description specification provided by Saab Avionics. An example of a test case description is given in Table 1. In some occasions during these translations, minor discrepancies were detected by the experts and consequently corrected in the corresponding document.

5 EVALUATION

In the following section is outlined how a populated ontology was evaluated. The evaluation of the testing strategies that lie behind the generation of the previously described inference rules is also described, as is the evaluation of the test cases that were generated by applying the inference rules on the test case ontology.

In the OSTAG project, 32 requirements (out of 38) and 37 corresponding test cases (out of 58) were examined. The reason for not evaluating the totality of the requirements was that the remaining 6 requirements were written with a different syntax. As a consequence, the presented semi-automatic test case generation would need to be modified. Furthermore, the main goal of the project was to present a proof-of-concept only. This included the semi-automatic generation of the same test cases as the ones that had been manually generated by test design experts. Hence, no additional, unforeseen test cases were generated. Furthermore, the completeness of the test cases was limited to this one-to-one validation between the semi-automatically and the manually generated test cases, only.

5.1 Ontology Evaluation

The correctness of the populated requirements ontology was evaluated where the correctness is defined as the degree to which the information asserted in the ontology conforms to the information that need be represented in the ontology (Tan et al., 2017). It is the most crucial feature when the ontology is to provide the knowledge base to generate test cases. Ideally, the correctness verification is an activity realized by a domain expert who not only grasps the details of the represented information itself but also has sufficient knowledge of ontologies to be able to perform

```
populate_w_tc(Ontology, tc(description(TC, Requirement, Service),
              PrerequisitesList, InputsList, ProcedureList, ResultsList)) :-
    ontology_comment(Ontology, TC, Requirement, Service),
    % Test case with data and object properties
    ontology_assertion(Ontology, declaration(namedIndividual(TC))),
    ontology_assertion(Ontology, classAssertion(test_case(TC))),
    ontology_assertion(Ontology, dataPropertyAssertion(hasTestCaseID(TC))),
    ontology_assertion(Ontology,
        objectPropertyAssertion(hasAddressedRequirement(TC, saab(Requirement)))),
    % Prerequisites
    ontology_comment(Ontology, TC, 'Prerequisites'),
    populate_w_prereq(Ontology, TC, PrerequisitesList),
    % Test Inputs
    ontology_comment(Ontology, TC, 'Test Inputs'),
    format(string(InputsListElem), '~w_inputs', TC),
    ontology_assertion(Ontology, objectPropertyAssertion(hasTestInputs(TC, InputsListElem))),
    populate_w_inputs(Ontology, InputsList, InputsListElem, 1),
    % Test Procedure
    ontology_comment(Ontology, TC, 'Test Procedure'),
    format(string(ProcListElem), '~w_procedure', TC),
    ontology_assertion(Ontology, objectPropertyAssertion(hasTestProcedure(TC,ProcListElem))),
    populate_w_procedure(Ontology, ProcedureList, ProcListElem, 1),
    % Expected Test Results
    ontology_comment(Ontology, TC, 'Expected Test Results'),
    format(string(ResultsListElem), '~w_results', TC),
    ontology_assertion(Ontology, objectPropertyAssertion(hasTestResults(TC,ResultsListElem))),
    populate_w_results(Ontology, ResultsList, ResultsListElem, 1), !.
```

Figure 5: The main Prolog predicate to populate the test case ontology.

the evaluation. In most situations this kind of human evaluator, with knowledge of ontologies, is not available. During the evaluation, an ontology verbalization tool was provided to the domain experts. The tool first verbalizes the ontology into a natural language text. Thereafter, the text can be read and understood by the domain experts, who have no knowledge of ontologies, and be compared with the information contained in the source information that was used to construct the ontology. The correctness of the ontology was verified by five domain experts from Saab Avionics, i.e. that the information asserted in the ontology was equivalent to the information written in the requirements documents.

5.2 Testing Strategy Evaluation

One part in the evaluation of the generated test cases consisted in the evaluation of the test case generation (TCG) strategies that a test designer apply during the creation of a test case based on explicit information found in, e.g., an SRS document or the competence acquired by a test designer in the software testing domain.

A strategy usually concerns only one element in one test case part, like an input or expected result, but can apply to several requirements. Each TCG strategy gives instructions on how a certain element of a TC can be constructed from the given requirement definition, other specifications, general testing recommendations and the designers competence in the testing domain.

The evaluation of the strategies consisted in the test designers comparing the text of already existing test cases with the texts found in the SRS, other relevant documents and their own knowledge on how to best write a test case. The evaluation of the strategies was thus performed by indirectly validating the quality of the inference rules through the quality of the test cases generated by the same inference rules.

An example of a strategy that is related to the expected test results and valid for seven different requirements (that resulted in seven different test cases) is:

```
When a device is disabled or uninitialised,
a service call results in not-init.
```

The result after applying the strategy above can be observed in the right column in Table 1 under Expected Test Results, point 2 and 3.

Another strategy example, that is also applicable to several requirements/test cases, is:

```
A recovery service call is necessary as the
last call for error handling requirement.
```

The result after applying the strategy above can be observed in the right column in Table 1 under Test Procedure, point 4.

Table 1: 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. <parity> := rs4yy_noneParity	**Test Inputs:** 1. <parity> := min_value - 1, <parity> := max_value + 1, <parity> := 681881 2. <uartID> := <uartID> from the initializationService call 3. <uartID> := <uartID> from the initializationService call 4. <parity> := noneParity
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_parityCfgError 2. <result> == rs4yy_notInitialised 3. <result> == rs4yy_notInitialised, <length> == 0 4. <result> == rs4yy_ok	**Expected Test Results:** 1. <result> == parityConfigurationError 2. <result> == rs4yyNotInitialised 3. <result> == rs4yyNotInitialised, <length> == 0 4. <result> == rs4yyOk
...	...

The result of the evaluation of all of the test strategies demonstrated a clear resemblance between the existing and the generated test cases, thus indirectly validating the quality of the test case generation strategies represented by the previously described inference rules.

5.3 Test Case Evaluation

To generate the test cases, a total of 40 inference rules were used. Together, they generated 37 test cases for 32 requirements. The corresponding test cases were reproduced in plain English, approximating the format described in the STD document (provided by Saab Avionics). To illustrate the similarity between the two representations, one specific requirement, the SRSRS4YY-435, that has a strong resemblance to the already presented SRSRS4YY-431, is described in this section. The results from this evaluation can be observed in Table 1 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 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 minor 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

This paper has presented an ontology-based framework for software test automation. The techniques from the area of knowledge engineering underpin the proposed framework. Knowledge representation with formal ontologies is utilized through the whole process chain of test automation, from the representation of software requirements to the representation of generated test cases. The use of OWL ontologies as logical models allows for reasoning, which was employed to generate test cases. To this end, the strategies for test case creation were represented with inference rules that were coded in the Prolog programming language. The Prolog inference engine powered the application of the rules to the requirements ontology to derive test cases that were used to populate the test case ontology. The preservation of the OWL axioms in the Prolog program allowed for meta-reasoning over the predicate statements in the ontol-

ogy, e.g. SubClassOf or ClassAssertion, and thus for the alleviation of the constraints of first-order logic. Test scripts can be created from the test case ontology in the form of plain text descriptions or executable scripts. The performed study has shown that via the application of ontology engineering and logic programming, the software testing process can be automated to a considerable extent.

The proposed framework has been implemented and evaluated in the OSTAG project. The requirements ontology was created to represent the software requirements specification for hardware controlling software in the avionics industry. The test case ontology was constructed based on the software test description. The inference rules represented the knowledge acquired from the software testers and provided test cases. 37 test cases were generated for 32 requirements. During the evaluation, the requirements ontology, testing strategies and generated test cases ware verified by domain experts from Saab Avionics. The evaluations performed on the framework demonstrated the validity of the overall approach. In particular, it showed the correctness of the ontology and the test cases as well as the validity of the test case generation strategies. In future work the framework will be evaluated for more testing tasks in different application areas.

The requirements applied in this paper are written in natural language but are still well structured. That being said, preliminary studies have showed that different levels of requirements are written in different ways. For example low-level requirements, like the example presented in this paper, are usually well-structured while high-level requirements are not. Thus, the created ontology development process is not capable of processing these different types of requirement in an optimal way. A preliminary study (Zimmermann, 2017) has indicated that exploiting standard ontological resources for natural language processing, such as FrameNet (Baker et al., 1998; Ruppenhofer et al., 2006), could be a solution when extracting and annotating the meaning of a high-level requirement in the best possible way while a lightweight ontology, or embedding rich meta-data within requirements documents, could efficiently support test case generation methods. Yet another approach could be to limit the complexity and the expressiveness that can be found in several high-level requirements by relying on requirement boilerplates (on a syntactic level) or requirement patterns (on a semantic level) when writing the requirements (Farfeleder et al., 2011; Arora et al., 2014; Böschen et al., 2016).

Although the framework presented in this paper proved to be effective for test automation, we identified two bottlenecks of the approach during our work on the industrial application. The first relates to manual ontology construction and the second to test strategies acquisition. Both activities are resource intensive when carried out manually. While general test strategies need to be acquired once, there are many domain specific strategies. Thus, our future work will focus on overcoming these deficiencies. Firstly, as already mentioned in Section 4.2, the work has been initiated on ontology learning, to semi-automatically create requirements ontologies from semi-structured documents. Secondly, the acquisition of inference rules for test case generation will be investigated by learning from examples with the help of inductive logical programming and other machine learning methods. It might also be of interest to apply the method on other, diverse applications, to prove its validity.

ACKNOWLEDGMENTS

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

REFERENCES

Arora, C., Sabetzadeh, M., Briand, L. C., and Zimmer, F. (2014). Requirement boilerplates: transition from manually-enforced to automatically-verifiable natural language patterns. In *Requirements Patterns (RePa), 2014 IEEE 4th International Workshop on*, pages 1–8. IEEE.

Baker, C. F., Fillmore, C. J., and Lowe, J. B. (1998). 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*, pages 86–90. Association for Computational Linguistics.

Böschen, M., Bogusch, R., Fraga, A., and Rudat, C. (2016). 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, pages 1–11. CEUR-WS.

CapGemini, HP, and Sogeti (2018). World quality report 2018-19. 72 pages.

Dobson, G. and Sawyer, P. (2006). 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*, pages 27–29.

Drummond, N., Rector, A. L., Stevens, R., Moulton, G., Horridge, M., Wang, H., and Seidenberg, J. (2006). Putting OWL in order: Patterns for sequences in OWL. In *OWLED*. Citeseer.

Enoiu, E., Sundmark, D., Čaušević, A., and Pettersson, P. (2017). A comparative study of manual and automated testing for industrial control software. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 412–417. IEEE.

Farfeleder, S., Moser, T., Krall, A., Stålhane, T., Zojer, H., and Panis, C. (2011). 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*, pages 271–274. IEEE.

Gangemi, A., Presutti, V., Reforgiato Recupero, D., Nuzzolese, A. G., Draicchio, F., and Mongiovì, M. (2017). Semantic web machine reading with fred. *Semantic Web*, 8(6):873–893.

Gartner (2018). Gartner global IT spending forecast 2019, 3rd quarter 2018. http://www.gartner.com/en/newsroom/press-releases/2018-10-17-gartner-says-global-it-spending-to-grow-3-2-percent-in-2019. Accessed December 20th, 2018.

Happel, H. J. and Seedorf, S. (2006). Applications of ontologies in software engineering. In *Proceedings of Workshop on Sematic Web Enabled Software Engineering (SWESE) on the ISWC*, pages 5–9.

HPE-UFT (2018). HPE-UFT homepage. https://saas.hpe.com/en-us/software/uft. Accessed: Dec. 20th, 2018.

Ismail, M. (2016). Ontology learning from software requirements specification (SRS). In *European Knowledge Acquisition Workshop*, pages 251–255. Springer.

Kaczor, K., Nalepa, G. J., Łysik, Ł., and Kluza, K. (2011). Visual design of Drools rule bases using the XTT2 method. In *Semantic methods for knowledge management and communication*, pages 57–66. Springer.

Motik, B., Patel-Schneider, P., and Parsia, B. (2012). *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax*. W3C, 2nd edition.

Nguyen, C. D., Perini, A., and Tonella, P. (2008). Ontology-based test generation for multiagent systems. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, volume 3, pages 1315–1320.

Ostermayer, L. and Seipel, D. (2012). Knowledge engineering for business rules in Prolog. In *Proc. Workshop on Logic Programming (WLP)*.

Petasis, G., Karkaletsis, V., Paliouras, G., Krithara, A., and Zavitsanos, E. (2011). Ontology population and enrichment: State of the art. In *Knowledge-driven multimedia information extraction and ontology evolution*, pages 134–166. Springer-Verlag.

RTCA (1992). *Software Considerations in Airborne Systems and Equipment Certification*.

Ruppenhofer, J., Ellsworth, M., Schwarzer-Petruck, M., Johnson, C. R., and Scheffczyk, J. (2006). Framenet II: Extended theory and practice.

Ruy, F. B., de Almeida Falbo, R., Barcellos, M. P., Costa, S. D., and Guizzardi, G. (2016). 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*, pages 527–542. Springer.

Sneed, H. M. and Verhoef, C. (2013). Natural language requirement specification for Web service testing. In *Web Systems Evolution (WSE), 2013 15th IEEE International Symposium on*, pages 5–14. IEEE.

Sterling, L. and Shapiro, E. Y. (1994). *The art of Prolog: advanced programming techniques*. MIT press.

Tan, H., Muhammad, I., Tarasov, V., Adlemo, A., and Johansson, M. (2016). Development and evaluation of a software requirements ontology. In *7th International Workshop on Software Knowledge-SKY 2016 in conjunction with the 9th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management-IC3K 2016*.

Tan, H., Tarasov, V., Adlemo, A., and Johansson, M. (2017). Evaluation of an application ontology. In *Proceedings of the Joint Ontology Workshops 2017 Episode 3: The Tyrolean Autumn of Ontology Bozen-Bolzano*. CEUR-WS.

Tarasov, V., Tan, H., Ismail, M., Adlemo, A., and Johansson, M. (2017). *Application of Inference Rules to a Software Requirements Ontology to Generate Software Test Cases*, volume 10161 of *Lecture Notes in Computer Science*, pages 82–94. Springer International Publishing, Cham.

TestingWhiz (2018). TestingWhiz homepage. http://www.testing-whiz.com. Accessed: Dec. 20th, 2018.

Wang, Y., Bai, X., Li, J., and Huang, R. (2007). Ontology-based test case generation for testing web services. In *Autonomous Decentralized Systems, ISADS'07. Eighth International Symposium on*, pages 43–50. IEEE.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):6796.

Zimmermann, O. (2017). Modelling complex software requirements with ontologies. Master's thesis, Universität Rostock.