Automatic Conformance Test Generation Based on a Verified Model of a Bus System Standard

Armin Lechlera, Tomas Maciasa, Agus Atmosudiroa*

*aInstitute for Control Engineering of Machine Tools and Manufacturing Units, Seidenstr. 36, Stuttgart, 70174, Germany

* Corresponding author. Tel.: +49-711-685-82440; fax: +49-711-685-72440. E-mail address: agus.atmosudiro@isw.uni-stuttgart.de

Abstract

In automation domain, the conformance of field bus devices in regard to the corresponding bus system standard is ensured by deploying conformance tests. The correctness of the tests is therefore decisive. A common practice to develop conformance test manually is fault-prone due to the fact that the test engineer can misinterpret the standard or the standard itself has defects. Therefore, in this present work we argue that a conformance test development should be based on formally verified requirements and propose an approach to automatically generate conformance tests based on the verified model of a bus system standard in order to assure the test correctness.

© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).
Peer-review under responsibility of the International Scientific Committee of the 8th International Conference on Digital Enterprise Technology - DET 2014 – “Disruptive Innovation in Manufacturing Engineering towards the 4th Industrial Revolution”

Keywords: Model-based engineering; Automatic test generation; Conformance test

1. Introduction

To ensure the conformity of field bus devices to the corresponding bus system standard, a so called conformance test is introduced. This conformance solution measures the device conformity by testing device behaviour regarding standardized functionalities that must be fulfilled.

The quality of the tests, i.e. the correctness and the coverage, is therefore decisive for the evaluation of the device conformity to the standard and hence the interoperability between devices from different manufacturers.

The test correctness can be achieved by eliminating possible error sources during test development, which can be categorized into two classes: First, programmatic errors during test implementation and second, conceptual errors due to misinterpretation of the standard, or the standard itself has a defect. While programmatic errors can be significantly reduced by deploying a test development tool, the conceptual error still remains an issue.

The objective of this paper focuses on the approach to ensure test correctness based on formal modelling and verification of the standard. By deriving tests from the verified model of the standard, the correctness of the generated tests can be ensured.

This paper is organized as follows: Section 2 introduces background information related to the process of developing conformance tests. Section 3 presents the requirements for formal verification in automation domain and Section 4 presents the related works. Section 5 describes our approach followed by the implementation in Section 6. Section 7 and 8 provide the summary and the future work.

2. Background Information

2.1. Conformance test manual development process

In the domain of automation, the bus system plays an important role. It enables communication between automation devices from different manufacturers. Such a regulation is known as bus system standard or norm (hereafter referred to as standard). It describes the desired system behaviour and specifies system restrictions. In this regard, a standard is considerably similar to a requirement widely known in the
software engineering. In both cases, they provide the basis for developing the corresponding tests.

However, standard documents \[1,2,3\] are described in a hybrid form consisting of natural language description, as well as complementary or impartial tables and pictures. As opposed to this, requirement documents are mainly composed in natural language \[4\]. Despite the inherently ambiguous and incomplete nature of the natural language, composing requirements in natural language is advantageous because it allows the involvement of stakeholders with different backgrounds \[5\].

Driven by an increased demand for technological advances or issues that have been encountered, bus system standards underlie continuous development. Fig. 1 depicts the workflow for conformance test and device development in dependency of standard development. The standard development essentially consists of three milestones: draft, final draft and finished. In order to timely provide a test environment to the device manufacturer as a reference for the device implementation, the test development takes place parallel to standard development. As soon as the standard documents reach the draft state, the conception of conformance tests begins. At this point, the functionalities to be tested are defined. Upon reaching the final draft state, the conception of conformance tests is further refined by specifying the test parameters such as test constraints, actions to be executed and the expected results. Subsequently, the test concept is implemented in a specific test framework in form of test codes.

To ensure the quality of the standard, the standard needs to be inspected. For this purpose, informal inspection techniques such as Ad-hoc review \[6\], Checklist Based Reading \[6\] and Perspective Based Reading \[7\] are applied. However, informal inspection techniques, even if properly applied, cannot guarantee to detect all the defects in the standard due to the reliance on the reader inspection skill \[8\]. Any undetected defects can then propagate in the conformance tests and subsequently in the devices.

Moreover, conceptual errors may further occur if the test developer did not correctly understand the standard so that a correct device behaviour is considered faulty. This leads to unnecessary efforts to clarify this particular problem and, consequently, to question the reliability of the remaining tests. Based on these insights, we argue that a conformance test development should be based on formally verified standards and propose an approach to automatically generate conformance tests in order to eliminate conceptual errors.

3. Requirements for formal verification in automation domain

In the narrow sense, modelling and verifying a standard means investigating the truth of a statement in the bus system standard, based on a model of the standard \[9\]. In this context, the model is a conceptual formal or non-formal representation capturing structural and behavioural aspects in the standard in a more or less abstract way. The conceptual model can facilitate deeper understanding of the standard as well as simulating desired goals and restrictions made in the standard.

The verification of the standard aims to support early detection of defects such as ambiguity or incorrectness. By definition, incorrectness is described as a combination of inconsistency and incompleteness \[10\]. The advantage of a formal model or modelling language with formal properties, such as mathematical model or finite state machine (e.g. UML State Machine), as opposed to non-formal model is the semantic definiteness that allows a definite representation of a requirement. By implication, this means that any unmappable requirement, i.e. the requirement that has multiple formal representations due to the possibility of different interpretations (ambiguous), can be attributed to a defect in the requirement. Moreover, during formal modelling assumptions are consequently not made. Therefore, modelling incomplete requirements containing omitted or implicit information will result into an unknown or undefined object in the formal model, which in turn can be attributed to a defect in the requirement.

While ambiguity and incompleteness can already be revealed during the modelling process, detecting inconsistency requires behavioural analysis. As defined in \[10\], inconsistency occurs when two or more requirements contradict each other so that all requirements cannot be fulfilled at the same time. This definition implies the necessity to analyse the relationship between the models in order to detect inconsistency.

The further advantage of a formal model is the possibility to automate certain analysis steps which in turn can increase the reliability in the verification result. Some of the important methods for automated analysis are simulation and
prototyping representing informal methods as well as model-checking and deductive methods representing formal methods [9]. In spite of high accuracy, prototyping is less favoured due to time and cost, while the deduction method requires a deep understanding in mathematical theory. Simulation is favourable for the usability and scalability but provides lower reliability in the verification result whereas model checking provides high reliability but is less readable [9].

The communication of Bus systems in automation domain mainly follows the Master-Slave Architecture, where the achievement of a goal always includes both master and slave device(s) based on master command. For this purpose, the master device triggers the execution of a functionality by transmitting command values and the slave device reacts accordingly. Based on the slave reaction, the master calculates new command values. This finite process is repeated until the desired goal is achieved. Due to the process dependency between master and slave, the consistency checking of functionality described in standard requires simultaneous observation of master and slave model and cannot be carried out monolithically for one specific model. This requirement can be ideally fulfilled by deploying a simulation to analyse multiple and concurrent behaviours. But since simulation provides lower reliability in the verification result, an adjustment or enhancement of the method is required.

4. Related work

4.1. Methods for modelling and verifying requirements

The process of modelling requirements includes behaviour elicitation and text processing. Different researchers [11,12] have presented approaches to obtain object-oriented model from natural language text. Both approaches concentrate on parsing natural language text to obtain the so-called Part of Speech (POS) tags, as well as the syntax function of each element within the sentence. POS tags correspond to a grammar category, e.g. noun, (modal) verb, determinant etc. On the other hand, the syntax function builds the cornerstone for recognising or mapping text elements onto desired model elements such as classes, attributes or associations. Nevertheless, these approaches did not address behavioural information.

Aceituna et al. [8] approached a method to manually translate functional requirements into a state transition diagram and exposes assumptions made in the requirements by manual identification of missing functionality and ambiguity. In this method, incompleteness is defined as any missing element that leads to a disconnection of two states (missing transition), whereas ambiguity is a result of a phraseological implication that leads to an unknown state. Kof's verification approach [13] is based on computational linguistics by extracting ontology from requirements consisting of a list of domain specific concepts and the corresponding taxonomic ("is-a") and non-taxonomic relations to identify communicating objects. The extracted ontology is then to be manually validated and modelled into a message sequence chart to reveal missing objects, i.e. message sender / receiver or the message itself resulting from ambiguous or incomplete requirements.

As opposed to formal methods for ambiguity and incompleteness, there are fewer methods for checking inconsistency. Zowghi et al. [10] provided a framework based on mathematical theory, i.e. non-monotonic logic, to prove the consistency of requirements at each step of the requirement evolution. In the concluding chapter, Zowghi et al. mentioned the negative opinion of practitioners regarding the practicability of mathematical theory for consistency checking. This is due to the fact that formal proofing method requires a deep understanding of advanced mathematics which is not always given in the bus system domain. Deepthimahanti et al. [14] implemented a tool that focuses on automated modelling of a natural language requirement into UML models, i.e. use-case, analysis class, collaboration and design class diagram. During the generation of the UML models, irrelevant classes and identification of aggregation / composition relationship among objects need to be eliminated manually. It can be argued that the elimination is not necessarily attributable to a defect in the requirement document but also possibly to a tool limitation. Moreover, a guideline or automated support is desirable to examine the relationship between modelled requirements, which is not given by the approach.

Acharya et al. [15] approached a method for consistency checking based on formal modelling and model testing. First, the requirements are modelled into a class diagram by means of a formal specification language called Raise Specification Language (RSL) from which so-called consistency conditions are automatically derived. The consistency conditions serve as a guideline and are then tested against manually written test cases which check the robustness of the consistency conditions and the respective requirements. The main drawback of the approach is, in our view, the manual creation of the test cases. Apart from being ineffective, the created test cases are not guaranteed to be error free. This can lead to an overlook of defects in the requirements or to confusion of the requirements engineer and consequently, to untrustworthiness of the remaining tests, if a test considers a correct requirement as false. Therefore, we consider testing to be an unsuitable method for inconsistency checking.

None of the approaches previously explained addresses the detection of all kinds of defect classes, i.e. ambiguity, incompleteness and inconsistency at the same time, for standards described in a hybrid form [8] [13, 14]. Additionally, these manual verification approaches are not reliable to detect incompleteness. For instance in [8], incompleteness cannot be exposed when the missing transition is succeeding in a missing state. Moreover, the approach cannot detect partial incompleteness, i.e. when a transition or state is not completely missing but rather only partial operations belonging to a transition or state are missing. The remaining approaches neither provide a high confidence in the model consistency [15] nor do they provide a widely usable proofing method [10].
4.2. Model based test generation

Regarding test generation, the advantage of having a formal model and especially a modelling language with formal properties is that the model provides all information required for the test generation in an implementation-friendly form. The provided information includes the precondition for the test execution, the actions to be executed by the systems involved, as well as the expected results based on the executed actions. However, the biggest benefit of generating tests based on a verified model is the elimination of conceptual errors mentioned in the introduction, since neither the model needs to be interpreted nor does it contain defects.

For a model-based test generation, there are two aspects that need to be considered: Firstly, generating test data to achieve a high level of branch coverage [16], and secondly, translating the information contained in the model into a domain-specific executable code. Since branch coverage is not considered in this paper, the following discussion will focus only on the latter. In this context, translating information includes matching keywords contained in the model with procedures implemented in the domain specific framework to obtain executable codes. Here, challenge arises because procedures of the same name can require different argument values and return different values. In this regard, the more readable the model is, the easier is the translating process.

Little and Miller [17] developed an algorithm for translating informal arbitrary keyword commands into an executable code based on a function tree to be used for web applications. The algorithm covers two aspects: Firstly, it corrects the spelling of the keywords and subdivides these on word boundaries into tokens by means of spelling dictionary. Here, return type, number of arguments as well as function name are extracted. Secondly, it searches recursively for functions that match the extracted information. Thummalapenta et al. [18] developed an algorithm which converts informal test specification into executable code and applied backtracking-based search to resolve ambiguity in the test specification. The algorithm is designed to be generally applicable and has been tested in web as well as enterprise applications. Here, the algorithm is claimed to achieve an 82 % success rate.

In both approaches, the input keywords are informal and therefore can be ambiguous and arbitrary. In this case, translating the keywords into the framework specific procedures can result in no or multiple matches which makes the deployment of an algorithm for the translation process necessary. However, in our case the ambiguity has been eliminated during the modelling process. By applying consequent wording every keyword will have a definite match and the translating process can be simplified by means of a dictionary.

5. Proposed Approach

The goal of our approach is to automatically generate conformance tests based on a verified model of a bus system standard, which is composed as a combination of natural language descriptions, as well as tables and pictures. While pictures always exist as a complement to a natural language description, tables can either exist in complement to or independent of a natural language description. Independent tables represent the more structured variety of natural language description and already provide useful meaning if interpreted independently. On the other hand, complementary tables can be considered as a data structure and only have useful meaning if coupled with the natural language description. Hence, the modelling process is rather impossible to be carried out fully automatically without human intervention. In this sense, a semi-automatic approach following a certain rule is applied to model parts of the standard described in natural language with or without complementary tables and pictures and a manual approach for modelling independent tables.

Fig. 2 depicts the workflow for the automatic test generation of the proposed approach. For conceptually modelling the standard, a UML State Machine is chosen for its readability characteristic, which is of high relevance for the test generation. During the modelling process, similar to the approach in [8], assumptions are consequently not made in order to expose ambiguity and incompleteness. As mentioned in the previous chapter, the functionalities described in standards normally imply the behaviour of a master and slave devices, as well as the exchanged data in this regard. For the
consistency checking of the model it is therefore necessary to observe both models simultaneously and enable the communication between the models. Based on this requirement, we deploy a simulation-based method for the consistency checking, which allows the observation of master and slave model at the same time. For this purpose, the conceptual model is adjusted into an executable model. The communication between these models is simulated by means of an XML file which substitutes the telegram used to enable the communication between master and slave device.

Since simulation provides a lower confidence level in the verification results, we combine simulation with model-checking method into model checking-based simulation in order to provide widely usable verification method and obtain reliable verification results. For the simulation purpose, the UML State Machine needs to be adjusted to obtain an executable (simulatable) model. The verification concept is based on reachability analysis. Here, all transitions and states are checked whether they can be reached without having any deadlock. Based on the executable model a test case list is generated, where a transition between two states is considered as a test case. For generating framework specific test codes, keywords contained in the test case list are translated by means of a dictionary.

5.1. Modelling concept

The foundation of modelling a standard into UML State Machine is to determine the behaviour of a functionality defined in the standard where tasks, options and conditions of an object regarding the fulfilment of a requirement are elicited. In this sense, independent tables are to be modelled in an ad-hoc manner while natural language description with or without complementary tables and pictures are to be modelled based on the following steps: (1) normalisation, (2) parsing, (3) analysis and (4) modelling.

During the normalisation process the document is prepared by restructuring the text following a concrete set of rules. This restructuring includes the relocation and addition of punctuations and commas to split complex sentences, as well as the resolution of pronouns (it, its, this, etc.). Moreover, different sentence structures found in the document are rearranged in a consistent structure: “Noun phrase (NP) + Modal verb (MD) + Verb phrase (VP)”.

In this context, noun phrase (NP) is the subject of a clause, i.e. the first part of a sentence consisting of an optional determinant and a noun which can refer to master or slave or any object that belongs to master or slave. The existence of the modal verb (MD) is optional where a verb phrase (VP) can consist of one or more verbs and objects as well as any additional complements to define an action. This structure can be broken by adverbs or conjunction, e.g. always and if, at the beginning of the sentences or between MD and VP.

Given the normalised sentences, the parsing process defines the grammar category of each word occurrence within a sentence and the syntactical relation between words of different grammar category which together form a syntax structure. The extraction is required in order to analyse the behavioural information of sentences, i.e. if an object has tasks, options or conditions. For this purpose, similar to [12], a catalogue is introduced in our approach to match normalised sentences with a definite behaviour. The syntax structure which composes our catalogue is defined in Table 1:

Table 1. Catalogue of syntax structure

<table>
<thead>
<tr>
<th>Behaviour</th>
<th>Syntax Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td>Subject + MD (shall/must/should/need/have to) + VP</td>
</tr>
<tr>
<td></td>
<td>Subject + MD + Adverbs + VP</td>
</tr>
<tr>
<td>Options</td>
<td>Subject + MD (may/can/could/might) + VP</td>
</tr>
<tr>
<td></td>
<td>Subject + MD + Adverbs + VP</td>
</tr>
<tr>
<td>Conditions</td>
<td>Conjunction (if/when) + Subject + VP</td>
</tr>
<tr>
<td></td>
<td>Conjunction (if/when) + Subject + Adverbs + VP</td>
</tr>
</tbody>
</table>

After analysing the normalised sentence, the behaviour of a sentence is definitely determined and can be transformed into UML state machine in the modelling phase. In this regard, tasks refer to required operations while options refer to optional tasks. Tasks and options relate both to operations that are to be performed within a state whereas a state can contain more than one operation. On the other hand, conditions translate to Boolean expressions which will initiate a state change if certain events, external and or internal, occur. Therefore, conditions are to be modelled as transitions in the UML state machine.

5.2. Verification concept

The proposed approach to verify the standard is divided into two main steps: (1) manual method to expose ambiguity and incompleteness and (2) model-checking based simulation for inconsistency and partial incompleteness checking. In the first step, ambiguity and incompleteness are exposed manually during the modelling process by consequently not making assumptions. In addition, we define a guideline to create a list of incompleteness candidates. It suggests that conditions should have “otherwise” information and a state machine should end in a closed-loop. If this is not the case, the corresponding state or transition is considered as a defect candidate which is to be further examined in the simulation process.

For inconsistency and partial incompleteness checking, a model-checking based simulation is applied, based on a reachability analysis. The goal is to reveal whether all states and transitions can be reached or performed without any deadlock. For this purpose, the corresponding models of the master and slave are examined, i.e. simulated, concurrently. To provide a communication medium between the master and slave model during the simulation we introduce a telegram model which emulates the capability of the bus telegram. The telegram model is designed according to the bus telegram
structure so that the master and slave model can write and read certain bit value in a determinate placeholder.

5.3. Test generation concept

Generally, a test environment consists of a device under test (DUT) and a test device which performs actions that examine the behaviour of the DUT regarding a specific functionality. Our test generation concept is based on a transition coverage criterion where each transition of a model corresponds to a test case. This assumption is based on the principle that by covering all possible transitions of a model, the necessity condition is fulfilled and therefore the behaviour of a DUT representing by this model is tested. To generate a test case each group of "state+transition+state" is analysed. The source state represents the starting point of the test and the transition corresponds to actions that are to be executed by the test device. Lastly, the target state relates to the expected values, which decide whether the DUT passes or fails the test. To derive a test from the model, the information contained in the corresponding states and transitions has to be obtained. This information, considered as keywords, is then translated into procedures of the specific framework.

In addition, a look-ahead strategy is introduced to obtain a proper test generation result (Fig. 3). On one hand, it searches for all properties, e.g. variables or objects, included in the model. These found properties are assigned as test condition so that the generated test is always executed under correct circumstances. On the other hand, the look-ahead strategy takes measures to assure that the evaluation of the expected results is carried out at the right moment. This takes place by executing procedures derived from the proceeding transitions of the target state to avoid the premature exit from the target state.

6. Implementation

In the present work, a standard of a bus system called Sercos is used [1]. Particularly, a release candidate of communication protocol document, which regulates the system communication, is chosen. To validate our elicitation rules defined in Section 5.1, a class of the Sercos standard without tables and pictures is used. First, the text is normalised according to the rules and parsed using a text processing tool called GATE [19] using Stanford Parser [20] to obtain the syntactical relation between words of different grammar category. Afterwards, the syntactical relation is analysed and translated into behaviour according to the catalogue defined in Table 1. For this purpose, we developed a tool which analyses the syntactical relation of a sentence and creates a final output containing only task, options or conditions ordered chronologically. The extracted behaviour is then manually modelled as an UML State Machine representing the conceptual model of the particular class of the standard.

In the first verification step, some incompleteness in the standard has been exposed due to missing states and transitions in the conceptual model. At this point, the model may contain abstract operations. In order to execute the model, such an abstract operation needs to be adjusted by assigning a value to it. For instance, the operation “Read first of sequence” is adjusted to “FoS = ReadTelegram.SCH("FoS")”. In consideration of the test generation process, a consequent wording is applied for the adjustment process. In the further verification step, the guideline defined in Section 5.2 is considered to create incompleteness candidates. The executable model is simulated based on reachability analysis by means of a simulation engine called AMUSE. Here, we implemented the telegram model in XML files structured after the Sercos telegrams. These allow the master model to transmit command values to the slave model and respectively allowing the slave model to acknowledge the master command. In order to synchronise the communication between the models, we additionally implemented a token mechanism.

The result of the simulation is evaluated to examine if all transitions and states can be reached without a deadlock. The Table 2 provides the simulation results of eight classes which have been verified in the present work. The detected defects in the class Sercos Messaging Protocol were resulted from incompleteness of the standard. Here, the missing transitions were listed in the incompleteness candidates created after the guideline. The remaining detected defects were resulted from inconsistency of the standard.

Table 2. Detected defects after simulation

<table>
<thead>
<tr>
<th>Class</th>
<th>Detected defects</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sercos Messaging</td>
<td>5</td>
<td>4 missing transitions,</td>
</tr>
<tr>
<td>Protocol</td>
<td></td>
<td>1 missing action</td>
</tr>
<tr>
<td>Connection Mechanism</td>
<td>0</td>
<td>Transition not performable</td>
</tr>
<tr>
<td>Communication Phase Switch</td>
<td>0</td>
<td>Transition not performable</td>
</tr>
<tr>
<td>SVC Handling</td>
<td>0</td>
<td>Deadlock in a state</td>
</tr>
</tbody>
</table>
Provided the verified model of the standard, we extended our tool to generate test cases from the executable model by evaluating the C# file AMUSE generated for the simulation. Since we applied a transition coverage criterion, the tool extracts information of each group of "state+transition+state", whereas the look-ahead strategy is applied before the tool translates each statement. With the look-ahead strategy, the tool goes through the C# file to obtain a list of properties used as test conditions and a list of the prevention procedures to avoid a premature exit from the target state. These are important for the correct operation of the test case. Thereafter, our tool maps the found keywords from the C# file to the corresponding framework procedures. Since the vocabulary used in the model is restricted due to the consequent wording applied for the executable model, the mapping is done statically. Therefore, it is possible to define 1:1 command mapping for each keyword.

To validate our tool we automatically generate all the test cases of a consumer state machine defined in Connection Mechanism class based on transition coverage criterion. The generated test cases were later compared against the already existing manually created test cases. The result was satisfactory for almost all the cases and achieved 13 of 16 correct test cases (81% success rate). The deviation of the remaining test cases results from the fact that the conformance solution software (including the framework) is not real-time capable. Therefore, the test cases have to be designed in such way that the time constraint between the conformance software and hardware is compensated. Nevertheless, this information was not given in the model.

7. Summary

In the present work we presented an approach to automatically generate test cases from a standard. The goal is to eliminate programmatic and conceptual errors that can occur during a manual test development. The correctness of the generated tests is assured by verifying the correctness of the standard. For this purpose, the standard, which is composed in a hybrid form, is first modelled by means of a modelling language with formal properties (UML State Machine). The process of modelling the standard includes behaviour elicitation and text processing. For this purpose, the text from the standard is normalised to restructure the text and parsed to obtain the syntactical relation between words. The obtained syntactical structure is then analysed to derive the behavior of each sentence.

The method is applied to elicit behavior of parts in the standard. The result reveals that the method was unable to derive a satisfying behaviour if the input data is a combination of text, diagram and table. In this case, we manually modelled the standard. As opposed to this, the method was satisfyingly able to derive the behaviour if the input data is a pure text. From the results of the modelling processes, some conclusions were drawn regarding the writing style of a standard: Functionalities shall be described sequentially and chronologically; "otherwise" information and any other obvious information shall not be omitted and relied to human interpretation. Functionality of an object shall be described independently, although references to each other as well as their parameters can be made.

To verify the model a model-checking based simulation is applied. Additionally, a guideline to create a list of a defect candidate is provided, which aims to support the verification process. Table 2 provides the result of the verification. The verified model was translated after a dictionary to generate tests based on a transition coverage criterion. A success rate of 81% was achieved.

Summarizing, the implemented model-driven verification mechanism is a suitable and complete approach for verifying a standard in an early stage, without the need of expertise knowledge or high mathematical formalism, and derive test cases automatically.

8. Future Work

The implemented tool for translating the generated keywords from the model into framework specific test codes uses static mapping. This implementation relies on the fact that the vocabulary used in the model was known and invariant. Although this approach is reliable for this specific application, the creation of the dictionary needs to be done manually and is therefore time consuming (especially if the keywords need to be translated in various frameworks). A possibility to optimize the translation process could be achieved dynamically by applying more complex searching and matching algorithms. This could result in robust translation tool, which is independent from the vocabulary employed. Another dynamic translation approach could be based on artificial intelligence. The concept is to develop a learning system which analyses the framework based on the pattern of existing test codes that are written manually and are proven to be correct. Lastly, to achieve a 100% success rate of the generated test cases, boundary conditions such as time constraint between involved systems must be considered.

References


