2018 IEEE International Conference on Software Quality, Reliability and Security Companion

Functional Dependency Detection for Integration Test Cases

Sahar Tahvili*, Marcus Ahlberg†, Eric Fornander†, Wasif Afzal§, Mehrdad Saadatmand*,
Markus Bohlin*, Mahdi Sarabi‡

*Research Institutes of Sweden - RISE SICS Västerås AB, Sweden
† Royal Institute of Technology (KTH), Stockholm, Sweden
‡ Bombardier Transportation AB, Västerås, Sweden
§ Mälardalen University, Västerås, Sweden
Email: {sahar.tahvili, mehrdad.saadatmand, markus.bohlin}@ri.se
† {mahlbe, ericfo}@kth.se
§ wasif.afzal@mdh.se
‡ mahdi.sarabi@rail.bombardier.com

***Abstract*—This paper presents a natural language processing (NLP) based approach that, given software requirements specification, allows the functional dependency detection between integration test cases. We analyze a set of internal signals to the implemented modules for detecting dependencies between requirements and thereby identifying dependencies between test cases such that: module 2 depends on module 1 if an output internal signal from module 1 enters as an input internal signal to the module 2. Consequently, all requirements (and thereby test cases) for module 2 are dependent on all the designed requirements (and test cases) for module 1. The dependency information between requirements (and thus corresponding test cases) can be utilized for test case prioritization and scheduling. We have implemented our approach as a tool and the feasibility is evaluated through an industrial use case in the railway domain at Bombardier Transportation (BT), Sweden.**

***Index Terms*—Software Testing, Dependency, Software Requirement, Internal Signals, NLP, Optimization**

1. Introduction

One of the main goals of software testing is detecting as many critical bugs in the system under test as possible, thus improving software quality. Having an efficient testing process may lead to an earlier fault detection [1] and thereby achieving better quality of the software product earlier. The concept of efficiency in a typical testing process can be interpreted in different ways, where all steps of the software testing life cycle (STLC) can be considered as candidates for efficiency improvement [2], [3]. One particular research direction that concerns test case selection, prioritization and scheduling has been researched to a greater extent as a way to improve testing efficiency. Previously, we formulated the problem of test case scheduling and prioritization as a multi criteria decision making problem [4], [5], [6]. We showed that in our specific context of vehicular integration level testing, multiple criteria affect test prioritization such as dependencies between test cases (and requirements), test cases execution time and requirement coverage. However, without automatically or semi-automatically inferring such criteria, the multi criteria decision making solution can be expensive and labour-intensive. More so, new test cases might be generated continuously (e.g., as part of a continuous integration environment), thus demanding the multi criteria decision making solution to adapt quickly to support timely decision making. One of the most influential criterion in our context is the existing dependencies between test cases. Identifying such dependencies requires a systematic approach that has to consider artifacts ranging from system architecture to testing data. In addition, a small change in the system under test may lead to generation of new test cases and therefore the dependency between test cases continuously changes. Unfortunately, there does not exist many explicit ways to capture such dependencies automatically as this problem is shown to be challenging as well as time and resource consuming [7]. However, identification of dependencies in test cases, especially at an early stage, promises to improve testing efficiency since dependent test cases are obvious candidates to be scheduled in a way that effectively test integration between implemented functions. In [8], we have shown through simulation that identifying the dependencies between test cases in the early stage of a testing process can help save test execution cost. Given the importance of identifying dependencies between test cases, this study answers the following research goal:

How can the functional dependency between integration test cases be detected at an early stage of the software testing process?

In order to answer the above research goal, we introduce a natural language processing (NLP) based approach that detects functional dependencies between manual integration test cases through software requirements specification. This requires analyzing multiple related artifacts such as the test specification, software requirement specification and the relevant signal information between the functions under test. The proposed approach is implemented as an aiding

978-1-5386-7839-8/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS-C.2018.00047

tool in Python. In order to show the feasibility of the proposed approach, an industrial testing project in the railway domain at Bombardier Transportation AB in Sweden is selected as a case study.

2. Background and Related Work

Creating independent test cases is strongly recommended for several purposes e.g. executing and rerunning test cases individually in different order, faster debugging and also easier extending and restructuring of test cases. However, for testing the interfaces between modules, we need to create a set of test cases which tests various parts of the combined interfaces. In reality, those test cases are functionally dependent on each other and has an effect on each other's execution. The concept of dependency is important in a wide range of testing contexts and dependency detection has become a research as well as in industrial challenge today [9]. Moreover, several kinds of dependencies between test cases have been identified by researchers, at the different levels of a testing process:

- **Functional dependency:** represents the interactions between the system functionality and their run sequences [10]. For instance, function F_2 is allowed to be executed if their required preconditions are already enabled by function F_1, thus the function F_2 is dependent on the function F_1. Consequently, all test cases which are designed to test F_2 should be executed any time after the assigned test cases for testing F_1.
- **Temporal dependency:** represents the execution sequence between test cases. It occurs mostly in the context of real time systems. If a test case TC_2 temporally depends on test case TC_1 then TC_2 should be tested exactly after TC_1.
- **Abstract dependency:** occurs in the models which have a hierarchical decomposition of the model elements [11]. Test cases are hierarchically dependent on each other based on the decomposition structure of the model.
- **Causal dependency:** can be assumed as another kind of temporal dependency, where specific data items need to be created by TC_1 before TC_2 is executed.

For detecting the mentioned dependencies between test cases, several methods have been proposed. An empirical study conducted by Bell et al. [12] concerns dependency between both manual and automated test cases in practice. The study is performed on 5 software issue tracking systems, though analyzing 96 real-world dependent tests. Ryser and Glinz [11] proposed a graphing language, *SCENT*, which can be used to document scenarios (a sequence of user interactions with a system). By documenting the scenarios of a system with *SCENT*, it was shown that the documentation has a consistent language and still interpretative for humans. Bates and Horwitz [13] utilized program dependence graphs (PDG) and adequacy criteria to identify components of a modified program in regression testing. The proposed approach reduces the overall time required for creating new test files. Moreover, a similar approach to [13] is proposed by Rothermel and Harrold [14] where changed def.-use pairs are identified by slicing the program dependence graphs. Caliebe et al. in [15] proposed an algorithm for prioritizing and selecting test cases based on the dependencies between components of an embedded system. By analyzing the system structure and architecture, a component dependency model is set up and a system graph is constructed. Applying path searching methods in the graph, test cases can be selected that are based on the dependencies between components in the system. The logical dependency between structured requirements is exploited by Arlt et al. [7] to automatically detect the redundant test cases. The main goal of their approach is reducing test suites, based on the results of executed test cases, in such way that a dependent test case will be failed if a corresponding independent test case fails. Acharya et al. [16] proposed a greedy approach for prioritizing test cases in component-based software development environment. From an object interaction graph, which is generated from the UML sequence diagrams for interdependent components, the value of an objective function is evaluated when selecting the test cases for execution. Depending on the numbers of inter and intra component object interactions, which are obtained from traversing the graph, the value of the objective function is calculated.

3. Dependency Detection at Integration Testing

In this section, we describe the details of our proposed approach to detect dependencies between manual integration test cases. In this paper, we just focus on detecting the functional dependency between test cases, where a functionally dependent test case is not allowed for execution until its independent counterpart test case be executed first. Functional dependencies are common in vehicular integration testing when the interactions between modules need to be tested [17]. Our proposed approach takes the software requirement specifications (SRS) as an input and provides the functional dependencies between test cases as the output. In most companies, the interaction between requirements can go undetected in the integration testing phase. The reasons are many such as fragmented view of the system, too much focus on testing individual requirements without being aware of the possible interactions and partially implemented functions where interactions cannot even be tested. In some cases, even the requirements specifications are a gray- box, meaning that interactions are not specified completely, or it is not possible to keep track of interactions without an automated tool support. In some companies, the requirements are considered as a black-box, where the testes do not receive any requirement specifications and they get directly the module to test. However, in some testing process, the requirements are designed based on assumptions, where the project designers simply assume some requirements in the design phase [18]. In this paper, the black-box of requirements is opened and analyzed by an NLP-based approach.

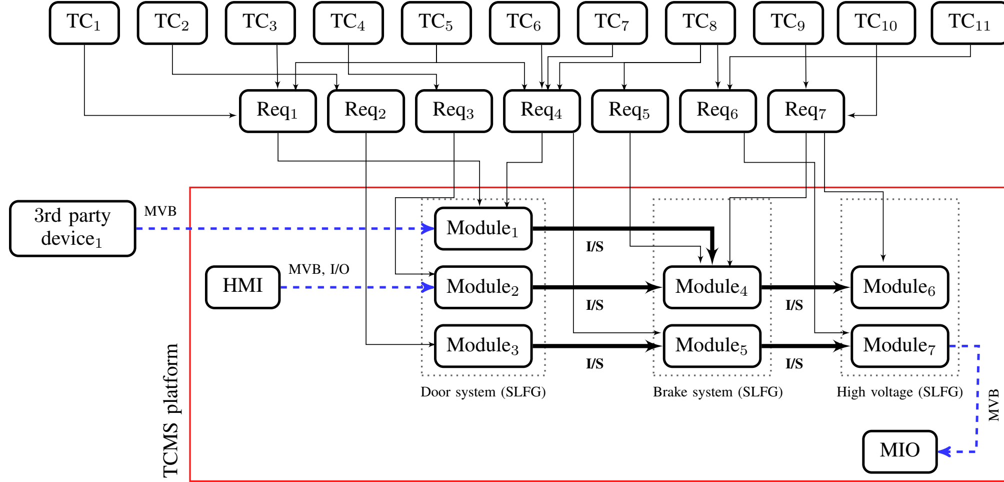


Figure 1: The input-output signals to the TCMS platform

3.1. Basic concepts definitions

Before describing the approach in detail, the following definitions set the foundations for understanding the problem.

Definition 1. A typical software module is a separated (sometimes interchangeable) component of a software. The program functions of a software are broken down into several modules, which should fulfill the project requirements [19].

In a vehicular context, the inputs and outputs of a module are usually given in the form of variety of signals. The number of involved signals are different, driven mainly by the functionality supported by the module. In our context, the following signals are the most common signals for testing the modules:

- **MVB:** stands for Multi-Function Vehicle Bus. Signals of MVB type are documented as an actual signal code together with a short description. The MVB signals are meant to connect different vehicles and usually use for data transfer between a hardware module and software module [20].
- **Digital I/O:** stands for Modular Digital Input-Output unit. I/O is documented with the actual signal codes as a communication protocol between two information processing system or humans [21].
- **Internal Signal (I/S):** is an internal communication protocol between two (usually software) modules. The internal signals are documented in natural language text. Assume $module_4$ and $module_2$ represent a software module in a brake and door system respectively. If a set of conditions (which are described in the corresponding requirement specification) are satisfied in $module_2$, an internal signal will be send to $module_4$.

Usually, MVB and Digital I/O signals interact with hardware (from-to hardware) and just the internal signals are transferred between the software modules. Thus, analyzing the internal signals between modules can provide clues about the dependency between modules and thereby their corresponding requirements and test cases.

Definition 2. A software requirement specification (SRS) describes a part of a function in a module. In other words, a module is described fully by a set of SRSs.

Moreover, a test case tests a set or just one of the SRSs, in order to verify compliance with a specific requirement. To provide a clear overview of the relationships between the modules, *input-output* signals, requirements and test cases, the Train Control Management System (TCMS) platform at Bombardier Transportation is chosen as an example in this work. The TCMS is a modular train-borne distributed control system platform, which controls the flow of information between the different sub-systems such as doors, heating, braking and air supply [22]. TCMS includes computer devices, software, human and machine interfaces, digital and analogue input-output (I/O) and is also able to integrate with third party devices. Figure 1 represents a part of TCMS platform with several modules from different sub-systems, where the various type of *input-output* signals go to the modules. The figure also shows the relationships between the requirements and test cases. The following sub level functional groups (SLFGs) are selected to visually describe the structure and relationships between the modules: door system, brake system and high voltage. By the modules in Figure 1, we mean just the software modules corresponding to the mentioned SLFGs. However, the HMI (Human Machine Interaction) and MIO (Multiple Input Output) represent two different hardware modules, which are located inside the TCMS and the 3rd party device is also a hardware module, located outside of TCMS. As illustrated in Figure 1, the hardware modules (inside and outside of the TCMS) are just sending the MVB and I/O signals (the blue dashed arrows) to the software modules. The bold black arrows represent the internal signals (I/S) between the software modules. As we can see in Figure 1 the *output internal signal* (I/S) from *Module*$_1$ is an *input internal signal*

for $Module_4$. This relationship represents the dependency between modules, where $Module_4$ depends on $Module_1$ and should be tested after $Module_1$. In other words, if all defined conditions in $Module_1$ are true, the internal signal will be generated as an *output* from $Module_1$. $Module_4$ receives the released *output signal* from $Module_1$ as an *input signal*. Moreover, the requirements and test cases (shown as located outside of the TCMS platform in Figure 1) are connected to the modules. As shown in the Figure 1, several requirements can describe a module and several test cases can describe a requirement. Since $Module_4$ depends on $Module_1$, all requirements meant for $Module_4$ should be tested after the assigned requirements for $Module_1$. Thus, Req_5 and Req_7 in Figure 1 are depending on Req_1 and Req_4. The number of test cases required to test a requirement can change during the testing process. Testers may create one test case for testing two (or more) requirements or two (maybe more) test cases can test just one requirement. Therefore, it may happen that a large number of test cases can be created for testing the software modules at integration testing. By using the dependency information between the software modules and the requirements, the dependency between test cases can be detected. Considering the above example, test case TC_9, TC_{10} should be tested after test cases TC_5, TC_6, TC_7. Moreover, test case TC_8 is a mutual test case, which tests both Req_4 and Req_5 and hence Req_4 depends on Req_5, then TC_8 can be tested first. For detecting the dependencies between test cases, we just consider the internal signals between the software modules and no other signals. As is obvious in the discussed example and also in Figure 1, our approach for dependency detection is a bottom-up approach such that the dependency between two software modules leads to detection of dependent test cases. Capturing the information for creating Figure 1 requires analyzing different resources. Partially, the signal information between modules is described textually in the software requirement specifications or visually in the software architecture. However, the signal information between modules can also be gained based on expertise and experience of the test engineers. In the early stage of a TCMS project, the core team analyzing the functional vehicle design specifications writes an SRS document. The SRSs are written by the requirement-engineering team members as early as the needed input is available to the project. The requirement adjustments are performed continuously during the project life-cycle. Each requirement from the SRS is assigned to an SLFG. The requirement is then implemented as a part of one module within the SLFG, or as a part of several modules within the same SLFG. Moreover, the traceability between the requirements and the test cases is generated for each software release and is stored in the SRS documents. During the project, some of the SRSs might be removed, merged or new SRSs might be added to the project.

Table 1 represents a sample of a software requirement specification (SRS) for three requirements illustrated in Figure 1, where various signals (*input-output*) have been described textually. We need to consider that the information inserted in Table 1 are gathered from three different SRSs documents files, merged as one table, in order to avoid repetition. The different type of signals (MVB, I/O) are distinguished in the *Interface* column for each requirement. We can see in Table 1, there is a shared internal signal between requirements Req_1, Req_5, and also between requirements Req_5 and Req_7. The internal signal `44-A34.X11.4.DI3` enters as an *input signal* to the requirement Req_5 and exits from the requirement Req_1 as an *output internal signal*. Moreover, the internal signal `95-B27.X01.5.PI10` exits from Req_5 and enters as an *internal input signal* to Req_7. The dependency between these three requirements can be presented as: $Req_1 \rightarrow Req_5 \rightarrow Req_7$ where Req_1 can be considered as an independent requirement in this case. Hence, the requirement Req_5 and Req_7 should be tested after requirement Req_1, then all test cases which have been created for requirement Req_1, has a higher priority for execution for those test cases which test requirement Req_5 and Req_7. As Table 1 shows, in some industrial cases such as ours (Bombardier Transportation), the signal information is available in textual form, described in SRSs. Therefore, in this work, we propose an NLP-based approach for analyzing the requirements specifications, in order to match the *internal output signals* with the same *internal input signals*. The dependency detection process is rerun if a new requirement is added or if some requirements are merged into one requirement. In practice, as explained before, the internal signals are communicated between the modules and not between the requirements. The inserted information in Table 1, represents how the modules corresponding to the mentioned requirements communicate with each other. Moreover, in our case under study at Bombardier Transportation, the traceability matrix for the requirements and test cases is available. In other words, all generated test cases are tagged to the requirements. This information is stored in the test specification and can be traced through performing the text analysis as part of NLP.

TABLE 1: Software requirement specification example - BT

Nr.	BTI/SFC Requirement	Interface
1	Doors-Req_1 (v.1)	input: MVB SDr_3EmRel output: Internal Signal 44-A34.X11.4.DI3
2	Brake-Req_5 (v.3)	input: Internal Signal 44-A34.X11.4.DI3 output: Internal Signal 95-B27.X01.5.PI10
3	High voltage-Req_7 (v.1)	input: Internal Signal 95-B27.X01.5.PI10 output: I/O $iDcuPd_{128}$

3.2. Implemented Method Details

In order to detect the functional dependency between test cases automatically, we have implemented our approach as an aiding tool in the early stage of a testing process. We divide our approach into three main phases: 1- requirement specification analysis 2- test specification analysis and 3- dependency visualization. Since both manual test cases and requirement specifications have a textual form, the natural language processing techniques are utilized for extracting the necessary signals information, which is then used for the matching process to identify dependencies. To map the requirements and test cases from the available documentation, an implementation in Python is done, having three steps of *test case extractor, requirement extractor* and finally, *test case and requirement combiner*. These three

steps take care of the first two phases of our approach consisting of requirement specification analysis and test specification analysis, while the last phase is visualization.

1) **Requirement extractor:** This step extracts the necessary information regarding the requirements from the requirements specification documentation. The documentation is first exported to *.xlsx* files from IBM Rational DOORS[1] database and the library package *xlrd* is used to parse the exported files. The requirements specification documentation is written in a semi-natural language, thus a specific algorithm was implemented to extract the required information (Algorithm 1). The extracted information is summarized as follow:
   - Requirement name
   - Version number
   - Input signals
   - Output signals

Algorithm 1 Requirements extraction

```
1: Set R to an empty list
2: for each each line in the documentation excel file do
3:     Read requirement name and input and output
4: end for
```

2) **Test case extractor:** This step extracts the necessary information from the test specification document. The documentation is also exported as *.xlsx* files from the DOORS database. In order to parse the extracted test specifications, a library package called *xlrd*[2] is used. The test specification is also written in a semi-natural language format by testers, therefore a specific algorithm need to be implemented for tracking the relevant information. The extracted information (see Algorithm 2) is consisted of the following:
   - Test case name
   - Requirements tested by the test case

Algorithm 2 Test case extraction

```
1: Set T to an empty list
2: for each line in the documentation excel file do
3:     Read test case name and requirements tested and append it to T
4: end for
```

3) **Test case and requirement combiner:** This step consists of two Algorithms (Algorithm 3, Algorithm 4), where result from the *test case extractor* and the *requirement extractor* are combined by *combiner* for dependency detection. It is done by first normalizing all signals to only alphanumeric lower-case characters (in order to eliminate inconsistent use of spaces, commas and dots). Then it creates a dependency graph with the requirements based on the signals (if output signal of requirement Req_1 is the same as the *input signal* for requirement Req_2, then Req_2 is seen as dependent on Req_1, see Algorithm 3). This data is then extracted in *JSON* format to a visualization web page.

Algorithm 3 Dependency detection between requirements

```
1: Set of documentation (inputsignals, outputsignals) of requirements, R
2: for each r1 in R do
3:     Set r1.dependencies to an empty list
4:     for each r2 in R do
5:         for each i in r1.input − signals do
6:             if i in r2.output − signals and r1 not r2 then
7:                 Add r2 to r1.dependencies
8:             end if
9:         end for
10:    end for
11: end for
```

Moreover, the *combiner* connects the test cases to their corresponding requirements (see Algorithm 4). Also, a dependency graph containing both requirements and test cases will be provided. Afterward, the graph is pruned to a new dependency by replacing each requirement with a new set of edges between test cases. Assume requirement Req_1 is tested by test case TC_1 and requirement Req_2 (which is dependent on Req_1) is tested by TC_2, this will be represented as an edge from TC_1 to TC_2 in the new graph. The resulting graph is also exported in a *JSON* format to a visualization web page.

Algorithm 4 Dependency detection between test cases

```
1: Set of documentation (inputsignals, outputsignals) of requirements, R
2: Set of documentation (requirementstested) of requirements, T
3: for each t in T do
4:     for each r in t.requirements − tested do
5:         Add t to r.tested − by
6:         Assert r.tested − by is non empty for all r in R
7:     end for
8:     for each t in T do
9:         Set t.dependencies to an empty list
10:        for each r in t.requirements − tested do
11:            for each r2 in r.dependencies do
12:                for each t2 in r2.tested − by do
13:                    if t2 not in t.dependencies and t2 not t then
14:                        Add t2 to t.dependencies
15:                    end if
16:                end for
17:            end for
18:        end for
19:    end for
20: end for
```

- **Visualization web page:** each visualization web page visualizes certain data (from *JSON* format) in different network graphs using the package *vis.js* which are presented later in this paper.

4. Empirical Evaluation

The feasibility of the proposed approach is analyzed through studying an on-going testing project at Bombardier Transportation (BT) in Sweden. The selected project is

1. Dynamic Object Oriented Requirements Management System
2. https://pypi.python.org/pypi/xlrd

called BR490[3] which in an underground subway train project in Stockholm. The guidelines of Runeson and Höst [23] for conducting and reporting the industrial case study research in software engineering is used as an inspiration for writing this section.

4.1. Unit of Analysis and Procedure

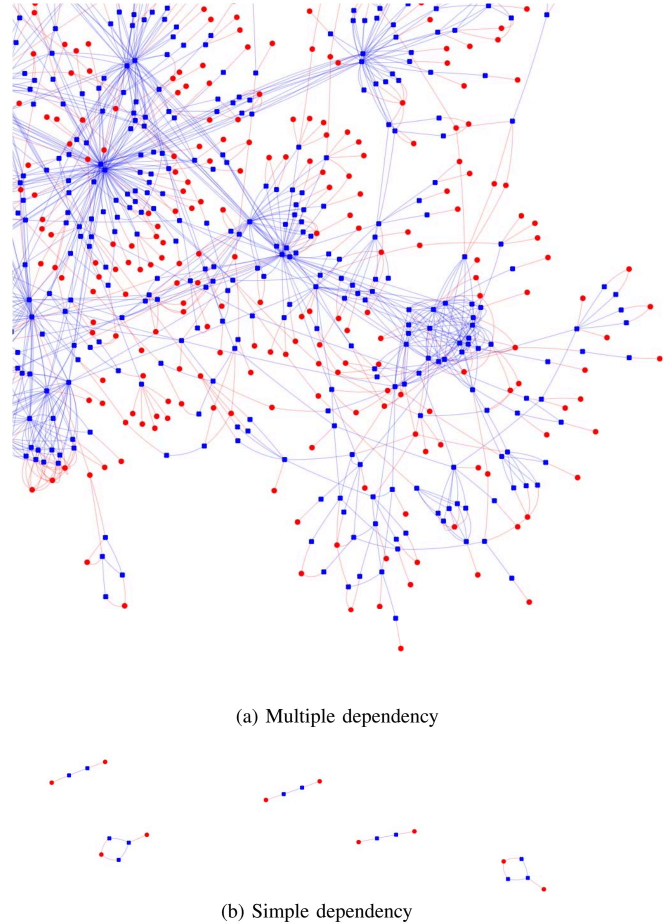
The unit of analysis in the study is the integration testing activity performed at the sub-level function group level at BT. Several steps are carried out to conduct this case study. **1.** A total number of 3938 requirements specifications (SRSs) from 17 different sub-level function groups are extracted from DOORS database at BT. **2.** The implemented method (as described in Section 3.2) is utilized for analyzing the extracted SRSs. The dependency between requirements are detected for 3201 SRSs while 737 SRSs are detected as being independent (as the method did not recognize any matching *internal signals* communicating with them). **3.** A total number of 1748 test specifications are extracted from DOORS database. The matched test cases to the corresponding dependent requirements are detected through analyzing the test specifications. **4.** The results of dependency between test cases are presented to the BR490 project team members. **5.** The testers' and engineers' opinions about the detected dependencies between test cases are collected and analyzed.

4.2. Case study report and results

Previously in Table 1, an example of the software requirement specification was provided. A typical SRS at BT consists of different pieces of information including the signal information. In order to evaluate the feasibility of our proposed approach and to help testers at BT visualize the dependencies between integration test cases, all SLFGs have been analyzed. The number of required SLFGs for testing a project depends on several factors such as the project size and importance from the customer perspective. A total of 17 SLFGs are assigned for testing in the BR490 project including: *air supply, automatic train protection, auxiliary power supply, bogies, brake* among others. As mentioned before, the approach recognizes the matched *input-output* signals for a total of 3201 requirements specifications for all SLFGs.

This fact indicates that there are no signal matches (an *output signal* for a corresponding *input signal*) for a total of 737 requirements in the BR490 project. This information might be interpreted as being 737 independent requirements existing in the analyzed project. In consultation with the testers and engineers at BT, a set of wrongly spelled meanings were found in the requirements and test case specification documents. Thus, the data in DOORS database sometimes contain ambiguity, uncertainty and spelling issues. Algorithm 3 searches for exactly same names for $input - output$ signals for detecting dependencies. In any case that, no *output signal* matches are found for an internal *input signal*, the corresponding requirement

3. The S-Bahn Hamburg BR490 three-car electric multiple units in production at Bombardier Hennigsdorf facility.



(a) Multiple dependency

(b) Simple dependency

Figure 2: The dependency between the requirement (blue nodes) and the test cases (red nodes) in the BR490 project

is counted as an independent requirement. However, by missing one letter in the name of a signal, no signal matches will be found, even if the signal enters (or exits) to several requirements. Moreover, the results of dependent requirements are visualized, in order to help the testers at BT for designing and creating better test cases from the requirements in the future that take dependencies into account. By using the dependency information between the requirements, an early prioritization of a number of decisions become possible. For example, the dependency information between requirements can help the test managers and testers to determine which candidate requirements should be included in a certain release [24]. For instance, it may very well happen that the independent requirements can be planned early for implementation. Moreover, following the dependencies between requirement, testers may want to focus on most dependent (and hence complex) requirements for more rigorous testing. They can thus plan for releasing such requirements first [25]. As stated before, in our context the dependency between requirements provides insights into the dependency between test cases, which is our ultimate objective. For testing BR490 project, a total number of 1748 test cases are created until the writing of this study.

Since the BR490 is an ongoing testing project, the testers might create more test cases, remove (or merge) some of the assigned test cases during the project. The relationships and dependencies between the requirements and test cases in BR490 project can be visualized in Figure 2, where blue nodes represent the requirements and the red nodes are test cases. The relationship and dependencies between requirements and test cases can either be categorized as *simple dependency* or *multiple dependency*. Figure 2b represents a simple dependency situation, when one test case (the red nodes) is created for testing one unique requirement (the blue nodes). This kind of dependency can be also interpreted as one-to-one dependency. However, Figure 2a shows the possible complicated nature of dependencies between test cases, where more than one test case is meant to test a requirement. The multiple dependency situation between the requirements and test cases in Figure 2a can be categorized as an instance of the shortest path problem, which can be solved by the graph traversal and path-finding algorithms. For instance, a single red or blue node (a test case or a requirement) will be assumed as the source node and the shortest paths from the source to all other nodes in the Figure 2a provides a shortest path tree. This can help answer different questions, for instance, minimal number of test cases required to cover a certain set of requirements. Further decisions can also be supported depending on the intended goal, such as to follow a certain execution path to satisfy one or more objectives. In addition to the visualization, the dependency between test cases are presented to testers at BT as a numeric value. Table 2 shows a part of our result showing dependencies between test cases. In

TABLE 2: Independent and dependent test cases and the number of requirements which are tested by each test case

Nr.	Test case ID	Requirement coverage	Dependent on	Output
1	IVVS_SBHH_ATP-IVV-51	38	0	0
2	IVVS_SBHH_Battery-IVV-96	24	0	0
3	IVVS_SBHH_Linevoltage-IVV-5	21	1	0
4	IVVS_SBHH_Drive-IVV-27	20	0	0
5	IVVS_SBHH_Trainradio-IVV-02	6	13	1
6	IVVS_SBHH_Linevoltage-IVV-3	4	12	0
7	IVVS_SBHH_Braketest-IVV-21	3	9	0
8	IVVS_SBHH_Speed-IVV-04	1	0	46
9	IVVS_SBHH_Speed-IVV-06	4	0	45
10	IVVS_SBHH_TC-IVV-07	2	0	36

Table 2 we can see how many test cases need to be executed before testing another particular test case. The independent test cases have the 0 value in the *Dependent on* column in Table 2. Moreover, the number of test cases that can be tested after each test case is inserted in the *Output* column. The requirement coverage (the number of assigned requirements to one test case) is calculated and stored in the *Requirement coverage* column in Table 2. For instance, the test case number 1 is an independent test case (the dependency value is equal to 0), which does not require execution of any other test cases before it. Furthermore, the number of 38 requirements are assigned to this test case. The testers can plan to prioritize execution of this test case first given its requirements coverage (in certain industrial contexts, the requirement coverage is assumed as the most important criterion for test case selection and prioritization [5]). The tester can also test this test case at any time due to its independent nature. On the other hand, test case number 5 is dependent on 13 other test cases with the requirements coverage equal to 6 and just one test case being dependent on it. Thus, test case number 5 is not a good candidate for first cycle execution given its dependency on 13 other test cases. However, after executing test case number 8, a total of 46 test cases will be available for execution and therefore this test case can be also considered for early execution in the testing cycle. As we can see in Table 2, test cases number 9, 10 are two independent test cases, where a total of 45 and 36 test cases are dependent on these two test cases respectively. The names of some dependent test cases to the independent test case numbers 9 and 10 are shown in Table 3.

TABLE 3: A test schedule example

Test Schedule		
Nr.	Independent Test Cases	Dependent Test Cases
1	IVVS_SBHH_Speed-IVV-06	IVVS_SBHH_Braketest-IVV-09 IVVS_SBHH_Battery-IVV-13 ⋮ IVVS_SBHH_Linevoltage-IVV-19
2	IVVS_SBHH_TC-IVV-07	IVVS_SBHH_Speed-IVV-05 IVVS_SBHH_Air-IVV-04 ⋮ IVVS_SBHH_ATP-IVV-12

Table 3 can be utilized as a test execution schedule, where the testers can execute independent test cases first and then dependent test cases can be ranked for execution based on the extent of dependencies. Lastly, we need to consider that the test cases in Table 2 can be prioritized and ranked in different ways, depending on the test objectives and the company's test policies. Sometimes running most test cases in a limited time period is required while in some other cases, a high requirements coverage is more sought after. Previously [26], we showed how the execution time for manual test cases can be predicted in the early stage of a testing process. The proposed approach in [26] performs a regression analysis on the similar previously executed test cases. Adding an estimation of the execution time for test cases to Table 2, can help testers to prioritize the test cases for execution in a more efficient way.

5. Discussion and Future Extensions

The main goal with our approach is to enable automatic detection of functional dependency between test cases by analyzing the dependency relationship of the requirements they target. The dependency information can be used in early stages of a testing process for test planning in order to make efficient use of available testing resources. The dependency information between the requirements that is identified as part of our approach also has the potential to be used for designing the product release plans. As soon as all requirements are captured, the test managers can modify the designed release plan for the final product. The Algorithm 3 presented in subsection 3.2 can be utilized for this purpose. Knowing the complexity of the requirements based on the dependencies between them can provide an estimation about the overall essential time which is needed

for testing each requirement. Moreover, prioritizing the requirements for implementation can lead to earlier release of a software product and also an efficient usage of the testing resources. The traceability graph illustrated in Figure 1 is not available for all testing processes. At the testing level in most companies, test cases are the only available information and the dependencies should be detected by analyzing just the structure of test cases. In the future, the provided results in this paper can be used as the ground truth for functional dependency detection between test cases. Any other proposed approach for dependency detection such as text analysis on the test specification need to be compared with the illustrated dependency graph in Figure 2. Finally, we designed and implemented a decision support system for test scheduling, which captures the inserted information in Table 2 and automatically ranks test cases for execution.

6. Summary and Conclusion

In this paper, we introduced an approach for detecting the functional dependency between manual integration test cases. The approach works by first analyzing the structure of the software requirement specifications. Additionally, a set of signals communicating between the software modules are identified, where the internal signals' inputs and outputs represent the dependencies between modules and thereby the requirements and test cases. Two software modules are considered dependent on each other if and only if the internal *output signal* from one of them is required by another one as an internal *input signal*. Based on the identified dependency between requirements, the dependency between the test cases are then determined. The proposed approach is evaluated through applying it on an industrial use case from Bombardier Transportation in Sweden, which shows the feasibility of the approach.

7. Acknowledgements

ECSEL & VINNOVA (through projects MegaM@RT2 & TESTOMAT) and the Swedish Knowledge Foundation (through the projects TOCSYC (20130085) and TestMine (20160139)) have supported this work.

References

[1] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, D. Sundmark, and S. Larsson, "Towards earlier fault detection by value-driven prioritization of test cases using fuzzy topsis," in *13th International Conf. on Information Technology : New Generations*, 2016.

[2] P. Abrahamsson, O. Salo, J. Ronkainen, and J. Warsta, "Agile software development methods: Review and analysis," *CoRR*, vol. abs/1709.08439, 2017.

[3] W. Afzal, "Metrics in software test planning and test design processes," Master's thesis, Blekinge Institute of Technology, Ronneby, Sweden, 2000.

[4] S. Tahvili, M. Saadatmand, and M. Bohlin, "Multi-criteria test case prioritization using fuzzy analytic hierarchy process," in *The 10th International Conf. on Software Engineering Advances*, 2015.

[5] S. Tahvili, M. Saadatmand, S. Larsson, W. Afzal, M. Bohlin, and D. Sundmark, "Dynamic integration test selection based on test case dependencies," in *The 11th Workshop on Testing: Academia-Industry Collaboration, Practice and Research Techniques*, 2016.

[6] S. Tahvili, *A Decision Support System for Integration Test Selection*, October 2016, Licentiate Thesis Dissertation, Mälardalen University, Sweden.

[7] S. Arlt, T. Morciniec, A. Podelski, and S. Wagner, "If a fails, can b still succeed? inferring dependencies between test results in automotive system testing," in *2015 IEEE 8th International Conf. on Software Testing, Verification and Validation*, 2015.

[8] S. Tahvili, M. Bohlin, M. Saadatmand, S. Larsson, W. Afzal, and D. Sundmark, *Cost-Benefit Analysis of Using Dependency Knowledge at Integration Testing*, 2016.

[9] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th International Conf. on Software Engineering*, 2006.

[10] S. e. Z. Haidry and T. Miller, "Using dependency structures for prioritization of functional test suites," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 258–275, Feb 2013.

[11] J. Ryser and M. Glinz, "Using dependency charts to improve scenario-based testing management of inter-scenario relationships: Depicting and managing dependencies between scenarios," 2000.

[12] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[13] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *Proceedings of the 20th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[14] G. Rothermel and M. J. Harrold, "Selecting tests and identifying test coverage requirements for modified software," in *Proceedings of the International Symposium on Software Testing and Analysis*, 1994.

[15] P. Caliebe, T. Herpel, and R. German, "Dependency-based test case selection and prioritization in embedded systems," in *2012 IEEE Fifth International Conf. on Software Testing, Verification and Validation*, 2012, pp. 731–735.

[16] D. P. M. A. Acharya and N. Panda, "Model based test case prioritization for testing component dependency in cbsd using uml sequence diagram," *International Journal of Advanced Computer Science and Applications*, vol. 1, no. 3, pp. 108–113, 2010.

[17] J. Gosling, G. Bracha, and G. Steele, *Java Language Specification*. Pearson Education, Limited, 2005.

[18] H. Bhasin, J. Hamdard, and E. Khanna, "Black box testing based on requirement analysis and design specifications," 2014.

[19] A. S. for Testing and Materials, *Form and Style for ASTM Standards*. American Society for Testing and Materials, 1983. [Online]. Available: https://books.google.se/books?id=abUkAQAAIAAJ

[20] H. Kirrmann and P. A. Zuber, "Mobile Systems IV," ABB Corporate Research, Tech. Rep., 03 2001.

[21] M. Abd-El-Barr, *Fundamentals of Computer Organization and Architecture*, ser. Wiley Series on Parallel and Distributed Computing. Hoboken: Wiley, 2005.

[22] "Mitrac train control and management system," http://transportation.bombardier.com/content/dam/Websites/bombardiercom/supporting-documents/BT/Bombardier-Transportation-MITRAC-Game-changing-Electronics.pdf, accessed: 2018-03-22.

[23] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 2008.

[24] L. Lehtola, M. Kauppinen, and S. Kujala, "Requirements prioritization challenges in practice," in *Product Focused Software Process Improvement*, F. Bomarius and H. Iida, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 497–508.

[25] P. Berander and A. Andrews, *Requirements Prioritization*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 69–94.

[26] S. Tahvili, M. Saadatmand, M. Bohlin, W. Afzal, and S. H. Ameerjan, "Towards execution time prediction for test cases from test specification," in *43rd Euromicro Conf. on Software Engineering and Advanced Applications*, August 2017.