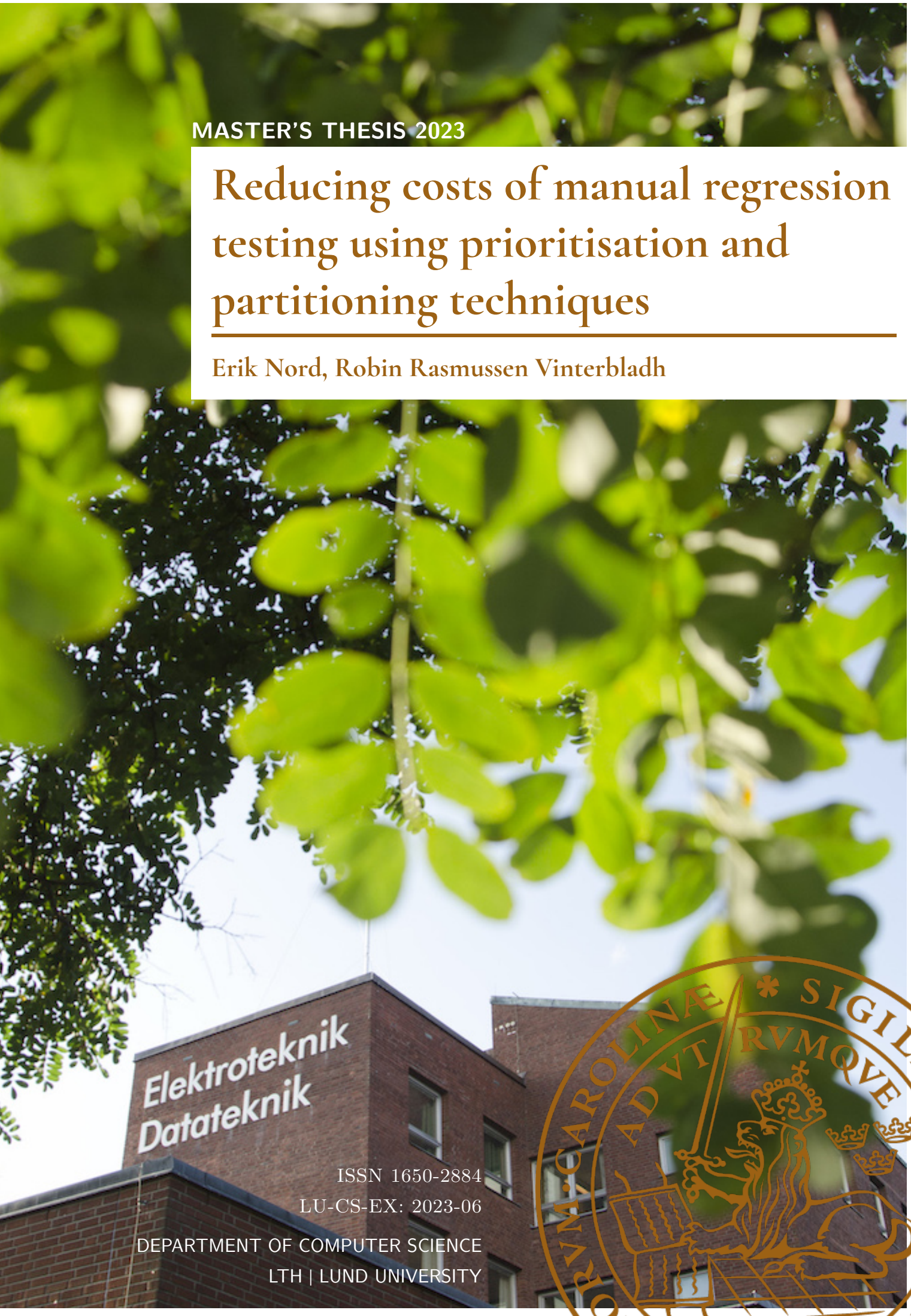


MASTER'S THESIS 2023

Reducing costs of manual regression testing using prioritisation and partitioning techniques

Erik Nord, Robin Rasmussen Vinterbladh

ISSN 1650-2884
LU-CS-EX: 2023-06

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-06

Reducing costs of manual regression testing using prioritisation and partitioning techniques

Kostnadsreduktion av manuell regressionstestning genom prioriterings- och partitioneringstekniker

Erik Nord, Robin Rasmussen Vinterbladh

Reducing costs of manual regression testing using prioritisation and partitioning techniques

Erik Nord
tpi13eno@student.lu.se

Robin Rasmussen Vinterbladh
ro3413vi-s@student.lu.se

March 29, 2023

Master's thesis work carried out at Sony Nordic (Sweden).

Supervisors: Peter Jansson, peter.jansson@sony.com
Per Runeson, per.runeson@cs.lth.se

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Context: This thesis was conducted alongside a team at Sony suffering from a classic regression testing problem. Regression test suites grows larger over time which leads to regression testing as an activity becoming increasingly costly. Automatic regression testing is cost-effective enough for high testing frequency. Manual regression testing is expensive, however equally necessary for test coverage.

Objective: Our objective is to assist Sony in reducing the costs for regression testing. Using data sources already available to them, we introduce regression test selection (RTS) techniques for manual regression test sessions. Data sources are historic test case performance and traceability between functional requirement specifications (FRS) to test cases.

Method: Using design science, we present two RTS techniques for solving the conceptualised problem: one primary RTS technique selecting a sub-set of the whole test suite based on historical performance and another complementary RTS technique based on FRS partitioning. RTS techniques in this thesis were evaluated with a data simulation over random generated data, a comparative study using real-world data, and continuous field study evaluating adaptations.

Results: We present a modified history-based RTS technique with extra emphasis on amplifying priority when test cases with low fail rates fail. This primary RTS performed at least as good as other empirically evaluated history-based RTS techniques. Our complementary FRS partitioning RTS technique consistently added extra cost to our selection, sometimes with and sometimes without any additional value.

Conclusions: Empirically evaluated RTS techniques are seldom designed primarily for manual regression testing. Automatic and manual regression testing differs in the frequency of testing sessions. RTS techniques focused on manual regression testing must take this difference in frequency into account. The added value from requirements partitioning depends on the completeness and traceability aspects of the underlying FRS, as well as the performance of the primary RTS technique.

Keywords: regression test selection, regression test prioritisation, design science, historical data, requirement specification

Acknowledgements

We would like to thank Lunds Tekniska Högskola (LTH) for providing us with a supervisor and examinator. We would also like to thank Sony Nordic, Lund, for providing us with the opportunity to conduct our thesis work alongside one of their teams.

Specifically, we would like to thank Per Runeson (supervisor, LTH) for continuously providing us with great wisdom and feedback throughout the entire thesis work. Additionally, we would like to thank Emelie Engström (examinator, LTH) for helping by giving us early guidance in finding relevant literature.

Also, we would like to thank Eskil Lundgren (test architect, Sony) for helping us with technical aspects during our time at Sony. We would also like to thank Peter Jansson (supervisor, Sony) for his amazing tips on how to create the best milk foam for espresso drinks and for his continuous feedback on our thesis work as well as being proponent for our project. Finally, we want to thank the entire team at Sony Nordic, Lund, for their patience with all of our questions and for providing us with valuable feedback on our work.

Contents

Chapter 1

Introduction

Testing is arguably one of the most important activities to increasing product quality in software engineering. This is why software development projects often have some form of testing integrated in their development cycle. Integrating newly developed features or any change to a feature into a system introduces some risk of unintentionally breaking that system. These faults are called "regressions" and the effort of detecting them is called regression testing. It is often the regression testing that takes up the majority of the time spent while testing in a software development project [21].

If we follow the principle to always maintain our regression test suite to verify new changes added to a system, we will end up with a regression test suite that continuously grows larger as the system itself grows larger. Overall, adding new test cases along with features this way is generally a good thing, because it is a direct strategy to maintain the same test coverage as the system grows larger in both size and complexity. Naturally, this will increase the time necessary for regression testing over time and the cost of running regression tests only becomes worse once you consider manual regression testing, that is, test cases that have to be executed by humans rather than by machines in automated testing frameworks. There can be many reasons to why some test cases cannot be automated. Any complex software project will likely at some point require manual regression test executions. This leads to manual regression testing being an incredibly time-consuming, and therefore costly activity.

In this thesis work, the case company seems to suffer the same problems: regression testing costs scaling poorly. During our thesis work, we will tackle the problems of regression testing costs by introducing regression test selection (RTS) techniques for a team developing Android TV solutions at Sony in Lund. Our case initiates from a state where the team can afford daily executions of their automatic regression test suites. However, these test cases are not sufficient. They also require manual regression testing efforts to fully regression test their system. They struggle to find the time to execute all manual regression test cases at the same

frequency as automatic test cases and instead, they strive to execute all manual regression test cases at least once every sprint cycle (spanning weeks).

There will always be a trade-off in regression testing between confidence and costs. A select-all-test-all strategy will naturally maximise confidence in testing outcome, but it will also maximise the costs. Inversely, select-none-test-none will completely remove confidence in testing outcome, but it will take zero time. In academia, there are efforts where RTS techniques are presented and empirically tested and applied to industry cases [3, 5, 11, 14]. RTS techniques seek to select a sub-set of all regression test cases that according to some metric is deemed likely to identify regressions. Previously performed systematic literature reviews reveal that prioritisation of the test cases are commonly based on change request content, recent changes in software requirements specification or historical test performance [1, 6]. More often than not, RTS techniques presented in academia are specifically made with automatic testing in mind, as noted by Ali et al. [1] in their literature review and the research gap between automatic RTS and manual RTS techniques is acknowledged in a recent study by Haas et al. [9].

In our thesis work we strive to improve Sony's cost-effectiveness by optimising their RTS from a select-all-test-all strategy to a new technique proposed by us. Hemmati et al. [11] suggests that adaptations of any RTS technique that is originally designed with automatic regression tests in mind could realistically be applied to manual regression tests. A test case, manual or automatic, is at its most primitive definition a verification that any well-defined input and procedure should always yield a pre-defined expected result. Before we began our thesis, we have set up research questions that will act as guidelines throughout our work over what we want to explore from an academic standpoint. All research questions are driven by the initial context outlined in this introduction, formulated as follows:

RQ1 What are the differences in prioritisation of automatic versus manual regression tests for the purpose of cost reduction?
RQ2 How does utilising the requirement specification in a selection algorithm for manual regression tests affect the efficiency?
RQ3 Is our implementation *perceived* as valuable and useful to the team at Sony?

We propose a new RTS technique that is specifically designed to be used for manual regression testing. Our proposed RTS technique will use historic regression test data. In conjunction with our proposed RTS technique, we also propose a new technique called software requirement specification partitioning (REQP), intended to complement the output of the initial selection done by an RTS technique. Our REQP technique utilises and therefore requires that each test case tracks what functional requirement is verified by the test case. We require that both manual and automatic test cases have requirements tracking, which allows us to use failing requirement specification items from automatic regression testing and flag what manual test cases have a high risk for failure, then add those high risk manual regression test cases to our selection.

This thesis is structured in such way where related work and background is presented in Chapter 2. We present our method in Chapter 3 which consists of three executive and result-generating phases. Our execution of the three phases are presented in Chapters 4, 5 and 6 respectively. Then we discuss our method, findings and validity in Chapter 7 before concluding by answering our research questions in Chapter 8.

Chapter 2

Definitions and Related Work

In this chapter we present previous work that relates to ours. Before we go into details of related work, we introduce some concepts used in this thesis. In Section 2.1 we introduce our definition of regression testing and in Section 2.2 we introduce software requirement specifications and how these specifications relate to regression verification. Finally, general related literature, related literature in regression test prioritisation (RTP) and related literature in regression test selection (RTS) is presented in Section 2.3.

2.1 Regression testing

When releasing new versions of a system it is important that the newly added features do not break any of the previously implemented functionality, and detecting these unintentionally introduced faults is what regression testing is used for. A regression test case is defined such that if the test case passes both before and after a change was integrated to the system, we say that no regressions have been found [8]. Regression testing is all about routinely and frequently testing the system's core functionalities to check for regressions, and thus ensuring that older functionality works even as a system evolves over time. Regression testing can be either automatic or manual, depending on the system and what kind of functionality that is being tested. Manual regression test cases tends to be more expensive to execute, both in terms of time, and therefore cost, but also due to the risk of human errors.

Each test case, manual or automatic, has an input vector, a testing procedure and an expected outcome associated with it. For a manual test case the input is defined as pre-conditions that should be fulfilled before the execution of the test case can begin. The procedure is a set of instructions, often written in natural language, rather than code. It describes the sequence of actions a tester should take to properly execute the test case. The expected result describes

what the outcome of the testing should be and any outcome that differs from the expected result is considered a failure.

2.2 Software requirement specifications

Requirements engineering is a branch in software engineering that describes the process of all stakeholders agreeing upon what features and aspects should be included into any software implementation [18]. This is done during planning phases for traditional software development methodologies, or early-on in the iteration/sprint for agile development methodologies. The resulting software requirement specification (SRS) is a document that describes these agreed upon features and aspects. An SRS represents the basis for agreement between the stakeholders [26].

There are standards for how an SRS should be expressed and structured. The language should be on a reasonably technical level where it can be used directly by developers to help them understand what features was originally asked for by the ordering stakeholders (product owners or customers). Structurally SRS documents are divided into two main categories: *functional* and *non-functional* requirements [13]. In our work, we will observe and use SRS describing the software under regression testing, however we will only focus on the functional requirement specifications (FRS) part.

Ideally the FRS section will contain all functional, feature driven, requirements necessary for implementing a piece of software. Practically, the FRS is a living document that gets versioned and changed over time like any other configuration item. In order to produce these discrete versions with as much quality as possible the FRS itself has to be structured in a way that enables version control. Which at its most basic level means that every requirement inside the FRS is identified with unique identifiers, so that they can be referenced by external tools and by test cases during testing activities.

Identifiers are commonly selected in such way where they have semantic meaning, which exposes the underlying structure among the requirements. Apart from the semantically sensitive identifiers, other quality factors are concerned when specifying functional requirements. IEEE standard 830-1998 defines these quality factors in detail [12]. Summarised by Lauesen [18], he describes factors such as *completeness*, *traceability* and *verifiability* among others. These three quality factors will be most important for this thesis work.

Completeness is achieved when the functional software requirements cover all the project owner's or customer's needs. Traceability describes the possibility to point out what regression test case is related to what functional requirement in the FRS and vice versa. Ideally every requirement is verifiable and is verified via one and only one FRS-traced regression test case. In our work, we assume that any FRS being used in our methods contains requirements that excel in these three quality aspects.

2.3 Regression Test Optimisations

Yoo and Harman [28] define three major branches of optimisation techniques for regression testing, namely: test case prioritisation, selection and minimisation. Test case prioritisation seeks to prioritise each test case based on certain metrics (e.g. code coverage, historic performance or perceived importance). The result of prioritisation can determine what order one should execute the test cases in order to achieve optimal results. Test case selection seeks to select a subset of all test cases to execute. The selection can be based on metrics such as code coverage, code changes or prioritisation based on any prioritisation technique. By only executing a subset of all test cases a lot of time can be saved, however you risk missing regressions if the selection criteria fail. Finally, test case minimisation seeks to remove test cases from the test suite, either because they are redundant or obsolete. Reducing the number of test cases in the test suite will naturally make it faster to execute them all, and all of these techniques have been proven to be cost-effective when applied correctly. We will focus on test case selection and prioritisation in our work since minimisation of the manual test suite is something that is already done by Sony.

Through our own literature research we have found that literature often assumes automatic testing. This is further confirmed by Ali et al. [1] in their systematic literature review where they mention that few papers are specifically on manual testing, and they conclude that more research should be made on this area. Haas et al. [9] talk about this research gap as well. One issue they bring up is that it is unclear which automatic optimisation techniques are applicable to manual testing. A problem in applying automatic optimisation techniques for manual testing is that some data might be missing, data that is commonly or easily available for automatic testing might be difficult to gather for manual testing. Another issue with manual testing is the fact that they are often written in plain text to facilitate execution of the test case by humans. The problem with this is that test cases written in plain text often tend to not follow software engineering best practices and therefore introduce test smells. Hauptmann et al. [10] discuss this in their paper, and they introduce a set of test smells for natural language test cases. Since changing how manual test cases from the manual regression test suite are presented to a new format was done during our thesis work, this paper helped us make design decisions regarding test smells, mostly for "inconsistent wording" and "ambiguous tests".

2.3.1 Test case prioritisation

Khalilian et al. use historical data for test case prioritisation: how often have a certain test case failed in the past, when was the test case last executed and so on [14]. The prioritisation technique made by Khalilian et al. was used in a case study by Engström et al. [5]. While the practitioners in that case were not satisfied with the specifics of the implementation, the authors still showed that history-based prioritisation improves the ability to detect faults early. This relates closely to our work since we can see many similarities between our case and the case presented by Engström et al., making this highly relevant.

In literature a review made by Ali et al. [1], only a few reviewed papers mention utilisation of tracking made from software requirements to test cases. They concluded that parts of those papers are lacking in various aspects. They are either lacking because they do not mention

how the tracking from requirements to test cases were made or how they can be generated. Krishnamoorthi et al. propose a change-based prioritisation technique using system requirements [16]. They find that by using prioritisation techniques that observe changes in system requirements, it is possible to achieve a better fault detection rate compared to random execution order. This is one of the papers critiqued for expressing the necessity of linking system requirements to source code and/or test cases but does not express how this linking should be made. Lastly, Lachmann et al. propose a selection technique based on a genetic algorithm optimised such that all generations selects a subset of test cases that will cover all requirement specifications [17].

Li and Boehm [19] bring up using a value based method for prioritising test cases. They claim that traditional testing methodologies usually treat all aspects of software as equally important. This leads to a purely technical issue leaving the close relationship between testing and business decisions unlinked and the potential value contribution of testing unexplored. They propose that rather than only prioritising based on metrics like code coverage or fault detection ratio prioritisation should also take into account the following aspects:

- Business / mission value: captured by business case analysis with the prioritisation of success-critical stakeholder value propositions.
- Testing cost: captured by expert estimation or based on historical data or past experience.
- Defect criticality: captured by measuring the impact of absence of an expected feature, not achieving a performance requirement or the failure of a test case.
- Defect-proneness: captured by expert estimation based on historical data or past experiences, design or implementation complexity, qualification of the responsive personnel, code change impact analysis etc.

An important lesson learned in this paper was that the value based prioritisation is not independent of other prioritisation techniques, on the contrary, it is meant to supplement them to add more value to the testing process. This paper is relevant to our case since it has been noted that some problems with the purely historical selection algorithms are that they sometimes prioritise test cases that are of little importance even though they fail. By adding a value based prioritisation criteria it might be possible to make up for some drawbacks with purely history-based selection.

2.3.2 Test case selection

Böhme et al. [4] present a way of using a partition based regression verification method, where the code is partitioned based on the input, if it reaches the same syntactic changes and whether it propagates the same differential state to the output. Basically for a certain range of input values, if the code that is traversed is the same and the output is within a certain range those inputs can be considered to be one partition of the program. By then testing one of these inputs and checking that the output is within the expected range before testing the same input in a different version of the program you can demonstrate the absence of regressions between the two versions, if the output remains the same [4]. This study made us think about how we can possibly partition the test cases at Sony in a semantic fashion. It

led us to start considering utilising the connection between test cases and the requirement specification to create the "partitions".

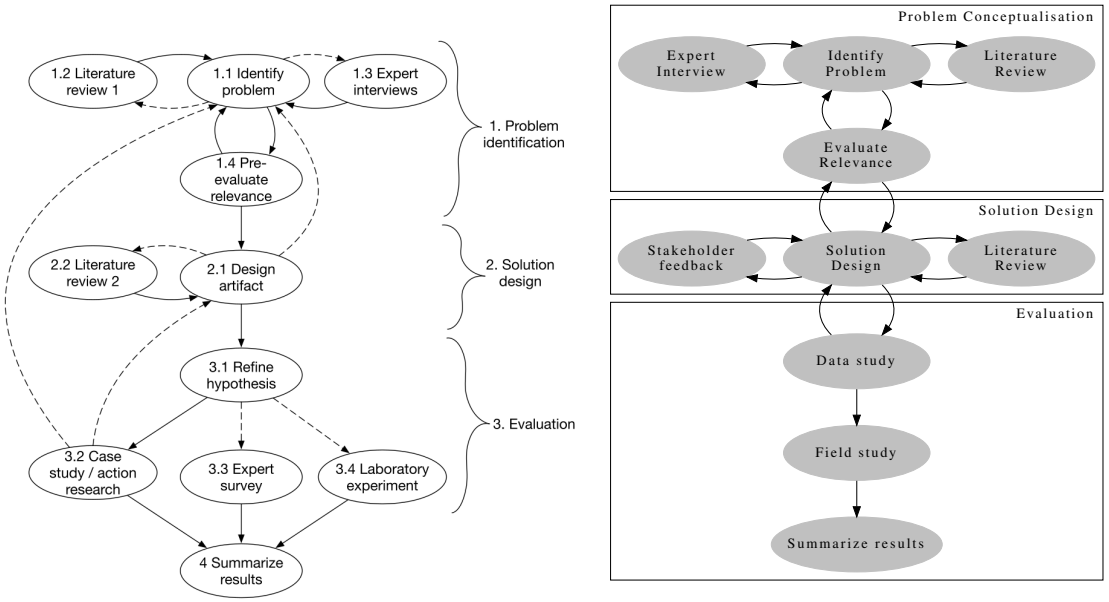
For manual verification regression testing Buchgeher et al. [3] present an approach using version control systems to find what parts of the code have been changed. Then by utilising data on what code coverage each test case have they can match so that they select and execute the test cases that specifically target the changed parts of the system. This is important in the case they are studying since the system is very large and difficult to test automatically so by only performing a part of the test suite they can save a lot of time and cost [3]. This paper is related to our case since we both try to deal with manual regression verification, even though we don't have access to code coverage, they still bring up a lot of interesting thoughts and problems they encountered in their implementation which makes it a valuable learning experience for us, so we can avoid those problems.

Chapter 3

Methodology

It is not obvious how selection and prioritisation should be made for the best results. Engström et al. observe that every project and every team has to come up with what works best for them [6]. They conclude that there is no ultimate strategy that is strictly better than all others. We interpret their conclusions to mean that before any optimisations on regression testing efforts can be done, a good problem conceptualisation at the case level is required in order to identify as many stakeholder needs as possible. By specifically looking for methods where a pronounced problem conceptualisation phase is key for success we found that Runeson et al. presents *Design Science* as a paradigm – an abstract framework with emphasis on *problem conceptualisation* to create a *solution design* and then *validation* of that solution [22]. Offermann et al. presents a concrete example of *Design Science* as a methodology [20]. Further, Wohlin et al. present the possibility of *Design Science* as a paradigm and as a methodology to coexist [27].

In this chapter, we present our design science adaptation that is based on Offermann's concrete example of *Design Science* as a methodology. We present descriptions for our methods used for the three design science phases and in Figure 3.1 we present an overview of our methodology compared to Offermann's. All methods used throughout this thesis work are outlined in this chapter's sections 3.1 through 3.3.



(a) Original method. **(b)** Our method.

Figure 3.1: Design science methodology work flow, extracted from Wohlin et al. [27] and in turn derived from Offermann et al. [20]. Leftmost graph shows the original method's structure and the rightmost graph shows the modified method that we will be working with.

3.1 Problem conceptualisation

The goal of problem conceptualisation is to ensure that a problem that has been identified is of practical relevance, and to find information regarding possible solutions. Technological rules specify how to achieve a certain goal by applying a specific intervention in a context, often formulated as a recommendation of how to use the designed intervention [22]. The problem conceptualisation phase are driven by research questions that represent theoretical outcomes of a conducted design study, and this phase will result in formulation of technological rules that captures the solution as a whole. In Section 4.2, we formulate two technological rules and further break them down into three design requirements. Once all design requirements are fulfilled, both of our technological rules will be possible to evaluate. This phase corresponds to the top section of our model presented in Figure 3.1b.

The issue of manual regression testing taking up more and more time had already been identified at Sony. Given the problem was already identified and verified as a known problem in our initial related work, we will start our thesis work with the problem conceptualisation step, and it will consist of further literature reviews and expert interviews.

Relevant literature was researched to see what previous efforts have been made to solve similar problems. Literature databases such as Lubsearch was used to find research related to regression test selection and prioritisation. We searched using keywords, for example: "*manual regression testing, regression test optimisation, -prioritisation, -selection*". Every identified paper

was then evaluated for relevance based on our intuition, however these relevancy scores were updated over time (some increased, some decreased) as we learned more about our problem. We filtered our results based on if they were peer-reviewed or not. From the peer-reviewed papers that we identified as relevant to our case we found more related literature by looking at what works were cited in them, and what works referenced them. Moreover, Runeson (supervisor) and Engström (examiner), who are heavily involved with testing research, were consulted for relevant literature on the subject.

Two expert interviews were conducted, they provided us with a better understanding of the problem at hand, in order to potentially make better design decisions for our solution design. All of our conducted interviews were semi-structured with different key-people currently working in the team at Sony. By semi-structured we mean that all interviews were in a casual setting with heavy back-and-forth conversation without a time box. All interviews were conducted one by one in order to improve our chances of getting individual answerers from every team members points of view. Topics and questions were the same for all of our interviews. Interview questions are presented in Section 4.1 together with a summary of all the interviewee's answers.

3.2 Solution design

In our solution design phase, which corresponds to the second section of Figure 3.1b, the goal was to implement a solution that fulfils all identified design requirements established during the problem conceptualisation phase. We achieved this by working agile, utilising a "kanban board" to stay organised.

Kanban is a tool commonly used during agile development [23]. Because we wish to keep it as simple as possible, we used Kanban to break down the problem to smaller tasks that we could more easily organise and implement. Instead of time-estimation which is a common practice in agile planning, we focused on our order of implementation. The main focus is to identify which features could be developed in parallel and which had to be developed in a sequential order. Our Kanban board was relatively simple, only containing columns: backlog, in progress, review and done.

Beyond usage of Kanban we also adapted the agile concept of frequent face to face conversations. Every week we held meetings with the stakeholders from Sony and every other week we held meetings with stakeholders from both from Sony and LTH. These meetings aimed to ensure that the work was progressing as intended and that the final product would achieve the stakeholder's requirements.

As problems appeared during our solution design, literature was consulted to find inspiration to solve said problems. Some problems that appeared were not covered in literature. For those problems, we had to actively make our own design decisions, sometimes based on feedback from stakeholders, and sometimes we had to come up with our own original solutions. Details regarding our solution design phase including all major design decisions can be read about in Chapter 5. Our design decisions are discussed in Chapter 7, and they were the basis for conclusions made regarding our research questions.

3.3 Evaluation

The evaluation phase aims to verify that our solution design actually solved the problem from our problem conceptualisation. By evaluating qualitative and quantitative data from our work we could underpin a discussion that ultimately answered our research questions. This phase corresponds to the third section of Figure 3.1b. Three methods are introduced in this report and all of them exist to evaluate different aspects of our solution. All their specific details are presented in Chapter 6, but a brief overview and how they relate to our research is presented in this section.

We did not have access to a large set of test data at Sony, therefore, we decided to use a data simulation study with random generated data in order to evaluate the performance of our solution. Since we are working with manual regression testing, while literature often assumes automatic regression testing, we identified a need to evaluate the performance of different RTS techniques in the context of manual testing. To achieve such an evaluative method for history-based RTS techniques, large amounts of historic data were required. Since we realised that we do not have much historic test data available to us at the beginning of our thesis work, this was the direct reason why we decided to use random data. This provided us with the opportunity to gather data with the aim of answering our research question **RQ1**.

We needed an evaluative method where we could assess the added value in comparison to the increased costs of using a complementary RTS technique. One of our goals was to implement a complementary RTS technique utilising traceability between test cases and the FRS. However, we could not find any cases in literature where this traceability was utilised in the way we propose. Therefore, we propose a method for comparing costs and values of using two RTS techniques together by calculating increase in efficiency, cost and the overall performance for the RTS techniques when utilised together. Evaluating these aspects directly supported us in finding conclusions for **RQ2**.

Lastly, we needed some way of evaluating the reception, adaptation and perceived value of our interventions. Our work implemented new additional steps to be taken for the Sony team's manual regression testing efforts. We mentioned perceived added value in **RQ3**, and quantifying perception is challenging. Therefore, we made qualitative observations during the course of our work whilst having minimal explicit interventions. I.e. we do not conduct exhaustive questionaries with all team members nor uphold focus groups that explicitly evaluates our work. This is a method referred to by Storey et al. as a *field study* [24].

Chapter 4

Problem Conceptualisation

This chapter presents activities performed during the Problem Conceptualisation phase as described in Section 3.1. Section 4.1 covers the interviews that were conducted, the questions that were asked and the answers that were elicited. Based on the information received during the interviews and during literature research, two technological rules were established and a number of design requirements that our solution design should satisfy were created from it. These are presented in Section 4.2, together with initial ideas on how to satisfy each requirement.

4.1 Expert Interviews

Interviews were planned and conducted to give a better understanding of the current state of manual testing but also to understand what goals and expectations they have for a potential solution. Questions and topics were outlined before interviews were held, and they were designed to elicit as much of the problem context as possible. The interview questions and topics are presented in Table 4.1 together with the section where the answer to each question can be found.

Two experts relevant to the problem were interviewed; a scrum master with past heavy involvement on the teams' testing process and the former test architect. The former test architect recently left their role as test architect, but because this was one month before interviews were held, they were still the most knowledgeable person about the historic state of the teams' testing efforts. All questions or some derivative of the questions adapted by the flow of conversation were covered in each interview. Answers to the questions were noted by us authors, and they are presented as a summary below.

#	Section	**Question**
1	4.1.1	Do you have any set processes and/or rules for how you execute manual tests right now?
2	4.1.2	Is there any present selection strategy for what tests to select?
3	4.1.3	Do you usually estimate cost in time for the manual tests? And are you able to hold this budget each sprint?
4	4.1.3	Are all the regression tests equally relevant/important?
5	4.1.3	The tests you have today, do you perceive them as enough?
6	4.1.4	Give us some insights with your current way of representing/defining manual tests.
7	4.1.5	Do you have any input, technical or other, about the idea of using the specifications-to-test mapping that you already have in testing efforts?
8	4.1.5	Are you happy with how you collect historic manual testing data today?
9	4.1.5	Are you happy with today's capabilities to overview the testing data (manual or automatic)?
10	4.1.5	Besides for the historic test data, can you think of other metrics that you find interesting to track for a selection algorithm?

Table 4.1: A table displaying all the questions covered by the semi-structured interviewing sessions and in what section their answers are presented.

4.1.1 Manual testing process

Both interviews reveal that there is a set process on how the manual testing was executed. At the time of interviewing, the process was recently changed, about a few months prior, and was therefore considered fairly new. During every sprint start a test lead is selected by the team, their responsibility is to ensure that testing activities are done during the sprint. It was pointed out in the interviews that being test lead does not mean you perform all testing activities on your own, rather you are responsible for following up on and delegating who executes what test cases for that sprint. Before the process changed, testing activities were done by the same person every sprint, in contrast to the rotating testing responsibility currently in use. Before testing is done, a version is selected and denoted as the official version to test. In the interview with the scrum master, it was mentioned that some minor issues have appeared over time. Debates on how to select versions to test were mentioned as an existing issue while implying that different team members have conflicting philosophies on what baseline to test from.

4.1.2 Test selection strategy

When it comes to the current selection strategy of manual testing both interviewees mentioned no formal selection strategy. Instead, in every sprint they select all existing test cases for execution. The scrum master pointed out that in addition to well-defined scripted manual test executions each sprint there is also time allocated for exploratory testing. What areas of the system that should be selected for exploratory testing is chosen by expert intuition every sprint, from that sprint's test lead, based on what areas they have worked on recently.

4.1.3 Test cost, importance and scope

No time budget is done on test-by-test basis. Instead, time spent on executing all manual test cases are recorded for every sprint. Over time, this creates a rough estimation baseline for the team to aid them in time-budgeting for their next sprint. Both interviewees mentioned that the current manual test suite is not enough. There are feature gaps in it, that is, features that are never tested but should be tested, so the addition of more manual tests will be necessary in the near future. However, the test architect mentioned that a lot of the current manual test cases should perhaps be revisited to see if they can be replaced with automatic test cases or removed entirely in the case of duplicates. Both interviewees agreed that manual test cases have different levels of importance. The scrum master pointed out that certain test cases are just not allowed to fail, for example if they are connected to parts of the system where they are legally responsible for the functionality to exist and work as intended. The test architect noted that a manual, static prioritisation for each test case used to exist in the past. This is something they aim to re-introduce in the future in order to reflect the fact that some test cases have higher importance.

4.1.4 Manual test case definitions and representations

For manual test case definitions, the test architect expressed that the definitions in place was just enough to enable testers to execute the test cases. He explains that in their current text-based representation of manual test cases' optional metadata, such as a brief human-readable summary of a test case's purpose can exist. Mandatory data such as the expected outcomes and testing procedure is also supported. However, no automation for validating that every test case contains all mandatory data fields was mentioned. The test architect did comment that such automation of validating mandatory data fields is of interest. Every test case definition is authored by the developers in plain text documents where testing procedures and expected outcomes have been documented. Concerns and shortcomings about this text based documentation format was conveyed by both interviewees. Since all manual test cases are checked in to the code repository, ensuring that test cases are human-readable was a concern. Human readability would facilitate efficient code reviewing of manual test case changes. The major concern with the plain text format was the above-mentioned absence of an automatic verification of metadata. Worries about metadata containing inconsistencies due to human error such as typos and ambiguous terminology was mentioned. Having the test case definitions represented in a machine parsable format was discussed as a potential solution during both interviews. The idea was that machine-readable formats can better ensure that every test case is defined under the same constraints.

4.1.5 Testing data overview and metrics

Both interviewees expressed discontent over how the manual testing data is currently collected and presented since it is done manually, which makes it prone to human errors. This makes it difficult to properly track and analyse the data. The test architect wants to collect and present this data similarly to the automatic tests, to facilitate comparisons between them

and to allow any programs that they use for analysis of automatic data to also be used on the manual testing data. The scrum master wants more overview of the trends, coverage and number of test cases that were executed and how many of those that failed. They hope that by automating the collection of data and presenting it in the same way as the automatic testing data this should be possible. In terms of metrics, both interviewees expressed interest in utilising the requirements-to-test mapping and historic data in some kind of manual testing selection algorithm. Moreover, recording some type of causality data between groups of failing test cases that statistically tend to fail together was suggested as a potentially interesting metric to use.

4.2 Rules & Requirements

With the results of our expert interviews, two technological rules was formulated, which concluded our problem conceptualisation phase. Based on those rules, three design requirements was constructed highlighting technical aspects our solution must fulfil before our technological rules could be evaluated. In Section 4.2.1 we present our two rules, then in Section 4.2.2, we present our three requirements. Further, in Section 4.2.3 we present our initial thoughts on how to approach a solution to these requirements.

4.2.1 Technological Rules

As explained in Section 3.1, technological rules specify how to achieve a certain goal by applying a specific intervention in a context, and they should ideally be formulated as recommendation of usage [22]. Our rules are presented in Figure 4.1.

TR1 To improve cost-effectiveness of regression testing in the context of manual executions, apply a historic RTS technique.
TR2 To further improve regression test performance from regression test selection, apply a REQP technique in conjunction to the original RTS technique.

Figure 4.1: Technological rules from our problem conceptualisation that captures our intended use for techniques designed and presented in this thesis.

4.2.2 Design Requirements

Breaking down our technological rules into design requirements was done to represent a decomposition of our conceptualised problem as a whole. The order of our design requirements is of importance. They are semantically ordered in reversed necessary order of implementation. That it, in order to fulfil **DR1**, we should find a solution that satisfy **DR2** first. Moreover, fulfilling **DR2** should be trivial if **DR3** was achieved. We present our design requirements in this in Figure 4.2 and then promptly motivate their existence in this section.

DR1 Optimisation of manual regression testing costs has to be done using data sources already available to Sony.
DR2 Represent testing data in a coherent format, no matter if the data are from manual or automatic testing.
DR3 Represent manual test cases in a machine-readable format that is human-readable and without loosing information from the old test suite.

Figure 4.2: Design Requirements based on our technological rules from the problem conceptualisation phase.

From our interviews, it was made clear that Sony have the problem of manual regression testing not scaling well from a cost perspective. They expressed gaps in their manual regression test cases and a desire to add more manual regression test cases into their test suites. Despite having a select-all-test-all strategy, they expressed that some regression test cases are not as important as others. Implying that time can probably be saved by somehow prioritising the most important regression test cases. We see a motive for us to optimise Sony's time spent on manual regression testing efforts, which is why we present our first design requirement, **DR1**.

Before **DR1** can be solved, all data points of interest has to be systematically obtainable. In our interviews, dissatisfaction of manual regression test data was expressed, mostly because there was no way to represent manual test data programmatically. This was something they could only do with automatic regression test data. This problem motivated the formulation of our second design requirement, **DR2**.

Even the way manual regression test cases were represented makes them difficult to work with programmatically. There is no way to guarantee that each manual regression test case contains all the required components such as expected results, pre-conditions and what functional requirement that it verifies. Both interviewees expressed the necessity of some way to verify that manual regression test cases were well-defined, which leads us to our third design requirement, **DR3**.

4.2.3 Proposed Solution

With our design requirements established we will now present our initial thoughts we have on how to approach a potential solution for them. Literature research was done continuously during the whole problem conceptualisation phase. Based on our expert interviews and the concrete conceptualisation of the problem via the above presented design requirements, we did further literature research. This time with focus on finding existing technologies and potential solutions to our design requirements. We finished our conceptualisation by creating an initial idea of how we should start approaching each component of the problem.

We designed a history-based RTP technique with a cut-off criterion turning it into an RTS technique, inspired from existing solutions in literature [7, 14, 15] to address **DR1**. Ljung et al. show that the historic based RTP techniques tend to perform well in practice [5]. However, testers' confidence in the selected test plan proposed by automated tools can be low without

any case specific adaptations. In order to mitigate potential loss of confidence in our RTS technique as observed in [5], we complemented our history-based RTS with a second RTS technique. Because there exist a gap in research regarding RTS techniques based on requirements specifications [1, 9], we find motives for designing our second RTS technique such that it is based on requirement specifications. Sony already have traceability from requirement specification to automatic and manual test cases, giving us an exemplary opportunity to design an RTS with requirement specifications as input data.

Data representations have to be improved. We solidified this in both **DR2** and **DR3**. Our initial idea to solve **DR2** was to create an interactive testing tool that allow us to design how the output (i.e. manual test data) should be represented. However, before any RTS implementation or testing tool can be done, the representation of manual regression test cases needed to be machine-readable, thus satisfying **DR3**.

Chapter 5

Solution Design

With a finalised problem conceptualisation phase our solution design phase could begin. The agile work methodology presented in Section 3.2 was used throughout the design phase. Before starting on designing a solution to our design requirements, some initial preparatory work was made to the infrastructure at Sony, enabling us to access data as we pleased. This preparatory work is presented in Section 5.1.

Because of the semantic order of our design requirements explained in Section 4.2.2, we present their solutions in reverse order. Meaning, we present our solution design for **DR3** in Section 5.2, our solution that satisfies design requirement **DR2** in Section 5.3 and finally our solution design for **DR1** in Section 5.4.

5.1 Preparatory work

Before our thesis work, the testing process was such that all the testing data was saved manually by developers in tables that were later shared by the team. Our interviews showed the desire among team members to automate the collection and presentation of testing data to reduce human errors. After interviews were held, we realised that writing all the necessary software that would automate testing data collection and presentation from scratch was not necessary. Plenty of assorted testing related tools and platforms were found already written internally at Sony. Specifically, an old testing platform designed for regression testing of an older generation product stood out as especially interesting to us.

The old platform was originally designed exclusively for automatic regression testing, but we figured it could be extended with functionality for manual regression testing to fit our needs. This old testing platform was used to create raw data files containing passes and fails of test executions. These raw data files were saved in an open source build and automation server

(Jenkins) as build artefacts. In essence, the old testing platform contained tools for collecting test data from and reporting testing data to the Jenkins build server.

Our revision of the testing platform was done to maintain its previous functionality but with the addition of handling manual test data, albeit for the latest generation of products. We also introduced means of fetching additional metadata beyond the raw test data from the build server, which is essential for downloading historic trend data with their versions enabling trends to be used in a history-based RTS technique.

5.2 Representing manual test cases

In this section we present our approach to solving **DR3**:

> *Represent manual test cases in a machine-readable format that is human-readable and without loosing information from the old test suite.*

We started with examining how manual test cases were represented before our solution was introduced. They were represented in free text using Markdown's formatting capabilities. Markdown is a lightweight markup language, a type of markup language that uses a simple and readable syntax. A markup language differs from "plain text" in that various symbols can be inserted in the text to control its structure or formatting. For example asterisks ('*') is used for italic and bold words: "**this is italic**" while "****this is bold****". Markdown was created to be human-readable even without any compiling, this is why it's considered a lightweight language. The benefit of using Markdown is that it is highly portable, a Markdown file can be opened and edited with virtually any text editor, since it's just plain text with some symbols inserted. It is also platform independent and future-proof since the text you write can be easily read and used even if for some reason the language were to no longer be supported [25].

After examining all manual test cases, it became clear that there is a weak underlying structure to these manual test case representations. By weak we mean that a majority of manual test case representations followed what looked like a structured template, while others diverged from the structure, so not all test case representations conformed to the same rules. This was identified as the primary reason to why machine-reading of the old Markdown representation was hard to achieve.

Another issue with the old Markdown structure was that procedure lists for manual test cases were previously represented using Markdown's table syntax. Markdown tables have a tendency to produce long lines of text in their Markdown files, causing severe wrapping when code reviewing the plaintext in Sony's code reviewing tool. Since text wrapping showed to reduce readability, an active design decision was made to enforce a procedure's action and expected outcome to be displayed on separate rows. An example of the old manual test case representation is displayed in Listing 5.1.

```
<!-- This is a comment -->
# example test title

requirements: F1_1;F1_2

valid: TRUE

| action | expect |
|:-|:-|
| Do this. | Expect this. |
| Then do this more complex task, which require some more words to describe. | Expect this step to be readable despite having a long step description. |
| Finally do some cleanup. | Expect everything to go well. |
```

Listing 5.1: Example of how manual test cases were represented using Markdown. Fields are in free plain text, which is a threat since if there was a typo in the `valid` field, say `ture` instead of `true`, it would be hard to notice that typo. Additionally, wrapping of Markdown tables in plain text when there is a limitation to text display width was noted as hard to read.

We corrected all diversions from the semi-structured Markdown text files by modifying the files, making all Markdown represented manual test cases follow a common structure across the board. This structure contained rules and definitions on what aspects every manual test case representation have to contain and what type they should be in. For example, every test case has to have a unique name, which is a string. Every test case has a list of strings that is functional requirement identifiers and so on. With this new structure, we could re-define all human-readable Markdown test case representations to a data serialisation language, making them both human-readable and machine-readable.

YAML was proposed as the new language that could be used to define the manual test cases. YAML is a human-readable data serialisation language, basically a format for storing data structures or objects in a human-readable text format that can also be easily read by a computer [2]. In YAML there are three primitive data structures: lists, maps and scalars. The syntax it uses to represent these data structures are rather minimal in order to be human-readable. For example, python-like indentation can be used for structure, colons separate key/value pairs in maps and dashes are used to create lists. The simplicity combined with being written in plain text and being supported in many programming languages makes YAML highly portable, it can be edited in basically any text editor and reading or writing to YAML files is often very easily accomplished in code. An example of how a test case written in YAML might look is presented in Listing 5.2.

```yaml
---
# This is a comment
name: example test title
requirement:
    - F1_1
    - F1_2
valid: true
steps:
    - action: Do this.
      expect: Expect this.
    - action: Then do this more complex task, which require some
              more words to describe.
      expect: Expect this step to be readable despite having a long
              step description.
    - action: Finally do some cleanup.
      expect: Expect everything to go well.
```

Listing 5.2: Example of how a manual test case can be represented in YAML. It contains fields with different types since `name` is a string, `valid` is a boolean and `requirements` is a list of strings. Because types can be serialised in YAML, it is easier to detect the same typo as explained in Listing 5.1, since `ture` would not map to a boolean value. Procedures are broken down into discrete steps, where each step contains one action, which is an instruction and one expected outcome for that action.

It was pointed out during our interviews that any method for verifying the contents of a test case is of interest to the team at Sony. With the introduction of the above presented YAML format, it became possible to implement such a verification tool. We designed the verification tool so that it ensured no required field is missing, and no field contains the wrong type of data. If any error was found by the tool it would give out an error citing what went wrong, what test case raised the error and where in that test case's file content the error was raised, much like how a compiler would do when a syntax error is encountered. Our idea behind the design was that YAML should allow for an easier reviewing process and since YAML is a data-serialisation language, ensuring that all test cases conform to a structure should be trivial using any programming language able to parse text. Code reviews of manual test case changes should now focus on human aspects such as grammar and ambiguity of wordings instead of technical aspects.

5.3 Representing manual testing data

The next design goal we had to achieve was **DR2**:

> *Represent testing data in a coherent format, no matter if the data are from manual or automatic testing.*

First we looked into how the automatic testing data was being generated and represented. For automatic testing we have a host machine who sends an input vector to a device under test (DUT), which performs a testing procedure and then our host machine records expected

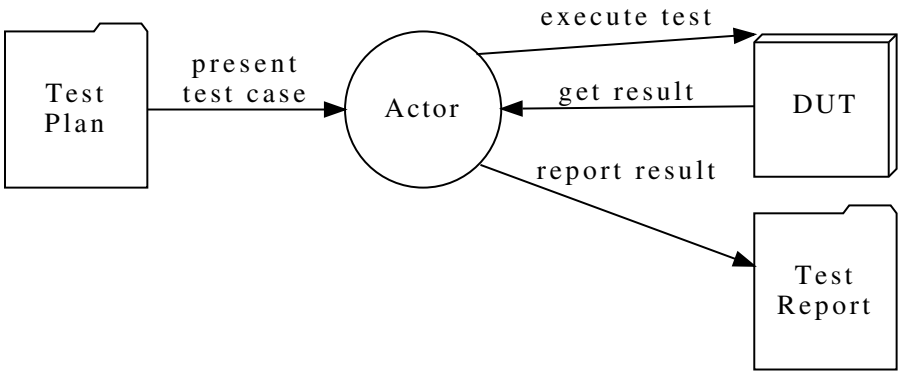


Figure 5.1: Context diagram displaying how an actor (human tester or host machine) interacts with a DUT during testing activity.

outcomes as either passes or fails based on the feedback from the DUT. This context is visualised in Figure 5.1.

With the old testing platform that we had decided to modernise, the automatic testing data was saved in raw data files that in turn were saved to Jenkins. We found that the automatic regression testing data was not generated by the testing platform itself. Rather, this generation was inherited from Android Open Source Project's (AOSP) built-in Gradle test job. That specific test job assumes that there is some DUT connected to the host machine.

If we consider our actor to be a person, a tester, instead of a host machine, and let them interact with a DUT while following their manual testing procedure. They will then report on expected outcomes based on the feedback from the DUT, just as in AOSP's testing context when the actor was a host machine. We can then see that representations of testing data is not affected by who or what executes a test case's procedure as long as each test case is well-defined and follows the structure of input vector, procedure and expected outcome.

Designing a solution that allows manual regression testing data to be represented in the *exact same* way as the automatic regression testing data was decided as a viable implementation strategy. We made the decision to create a testing tool with the purpose of guiding a tester during testing procedures. Two assumptions were made: First, the tool should take a list of test cases as an argument. Second, we assume that **DR3** from Section 5.2 is satisfied.

The tool was implemented to work in the following way. All existing test cases were loaded, then filtered by test cases mentioned in the input arguments. Test cases are then iterated and procedures, together with expected outcomes, are displayed on screen. A tester performs the displayed procedures and verifies expected outcomes. Verification of procedure steps was done with a prompt also displayed by the tool. The tester responds with pass or fail for each test case and this data was gathered by the tool, then compiled into an output file. When a test case was marked as failing the tester got a new prompt for adding a comment in plain text, explaining the reason for failure. Comments were preserved in the output together with the reported failure. Passes had no comments. See Figure 5.2 for an example of how a test case presented by our tool might look.

The tool is the only source that provides data to the output file, therefore it was designed in such way where we can guarantee that manual regression testing data follow the same format

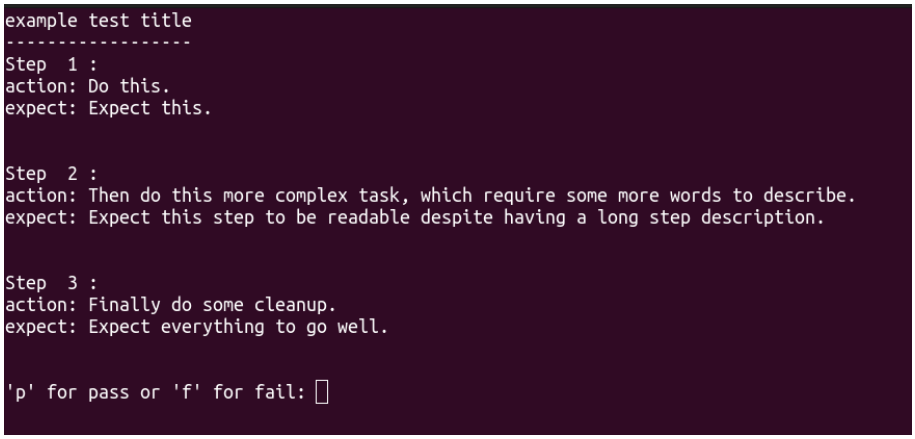


Figure 5.2: Example of how a test case is presented in the testing tool. The test case used is the same YAML example test case from Listing 5.2.

as automatic regression testing data. The final step was to upload the file to Jenkins where trend lines for each test case is promptly created.

5.4 Optimise costs of manual regression testing

For our final part of the solution design phase we present our design process for a solution satisfying **DR1**:

> *Optimisation of manual regression testing costs has to be done using data sources already available to Sony.*

For this section, it is important to think of a principle where RTP algorithms can be translated to RTS technique, if we introduce a selection criterion [5]. Any history-based RTP will order a list of test cases based on priority scores according to their past performance. Applying a selection criterion to this ordered list, we can create a test plan from that prioritised list, thus turning our RTP into an RTS. This relationship between RTP algorithms and RTS techniques is displayed in Figure 5.3.

By keeping our selection simple, i.e. in the style of selecting the top x percent of the prioritised list, our hypothesis is that we will satisfy **DR1**, if we can find a performant RTP algorithm.

RTP algorithms have already been explored and many solutions to RTP problems can be found in literature, as presented in Section 2.3. Our initial intentions were to find an empirically evaluated RTP algorithms and apply this to a real world case at Sony. An RTP algorithm proposed by Khalilian et al. in 2012 (Khalilian2012) was found [14]. This RTP algorithm fits our case well since it uses historic test data to prioritise each test case, which is something that we can access after **DR2** and **DR3** are achieved. Moreover, we found Khalilian2012 to be of interest because of its relative recency, leading us to believe this would work well in a

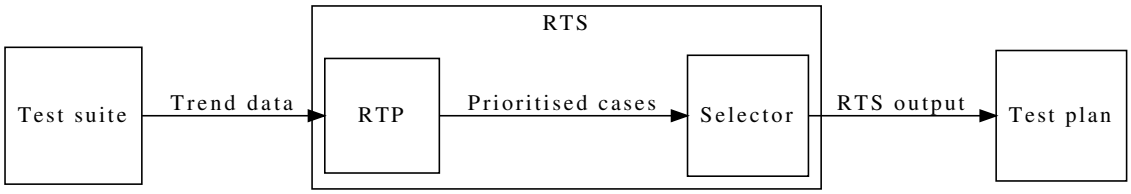


Figure 5.3: Block diagram displaying the principles behind RTP, RTS and how they relate.

modern software development project.

Khalilian2012 was implemented according to their paper [14]. Each test case is given a base priority. Then based on the test case's historical performance this priority value is modified to reflect the chance of finding a regression. In their paper, code coverage data is used to determine base priority score of test cases, which implies that Khalilian2012 is not a black-box technique. They address this issue by stating that alternative metrics can be used to determine the base priority score. A complete overview of this technique is presented in Appendix B. We decided to use a static priority score given to each test case, since we do not have access to code coverage metrics.

After implementation, exploratory testing was made where we manually fed inputs to the RTP algorithm and studied the plausibility of the results. Inputs to a history-based RTP algorithm consists of a vector of skips, passes or fails, representing the historic trend of a test case. We observed that when test cases are skipped in succession then prioritised with Khalilian2012, the slight increase in priority score does not compensate enough to make up for other factors in the algorithm detailed in B.1. Further, we observed that no matter the ordering of input (skips, passes or fails) priority scores approached zero but at differing speeds. Skips accelerated the slowest towards a zero score, thus retaining higher priority scores. Fails accelerated slightly slower towards a zero score than passes, yet faster than fails. Passes had the fastest acceleration towards a zero score. We found no natural or self-explanatory meaning behind priority scores in Khalilian2012. A test case's priority score was just a purely arbitrary decimal number intended to be put into relation to other test cases' score.

All these observed aspects from exploratory testing lead us to decide that Khalilian2012 was not a good solution to the problem at Sony. As such, we decided to implement two other RTP algorithms for comparison. Khalilian et al. presented a predecessor of our initially chosen implementation from 2009 (Khalilian2009). This earlier technique was implemented [7]. We also decided to implement Kim and Porter's RTP algorithm from 2002 (Kim2002), which is found to be one of the earliest history-based RTP algorithms [15]. The principles behind these two are essentially the same as for Khalilian2012. There are common equations present in both Khalilian2012 and Khalilian2009, which are displayed in Figure 5.4. In Khalilian2009 and Kim2002, priority score is dealt to all test cases, and this value fluctuates based on performance of the test case, just like Khalilian2012. We noticed while the principles among the three were common, all of their implementation details differ. Overviews of each technique can be found in Appendix B. Both of the additional RTP algorithms were implemented according to their respective papers.

$$fc_k = \sum_{i=1}^{k-1} f_i, \quad f_i = \begin{cases} 1 \text{ if test case revealed faults in session } i \\ 0 \text{ otherwise} \end{cases}$$

$$ec_k = \sum_{i=1}^{k-1} e_i, \quad e_i = \begin{cases} 1 \text{ if test case was executed in session } i \\ 0 \text{ otherwise} \end{cases}$$

$$h_k = \begin{cases} 0 & \text{if test case was executed in session } i \\ h_{k-1} + 1 & \text{otherwise} \end{cases}$$

$$fr_k = \frac{fc_k + 1}{ec_k + 1}, \quad 0 < fr_k \leq 1$$

Figure 5.4: Conventional equations for RTP techniques [14]. Fail count, fc_k is the number of fails for a test case up until test session k. Execution count, ec_k is the number of executions done for a test case up until test session k. Skip count h_k is how many consecutive test sessions a test case has been selected but not executed. Fail rate, fr_k, is the ratio between fails and executions, adjusted such that denominator is never zero.

Again, these RTP algorithms were exploratory tested using manually fed input, exactly as we did with Khalilian2012. We noted that Kim2002's and Khalilian2009's behaviour were more intuitive than Khalilian2012. Their priorities did not approach zero no matter the input, and they behaved more like we would expect a history-based RTP algorithm to behave. More precisely, their priority scores increased on failures and skips but decreased on passes. However, there were minor things in their implementations that we did not like. More precisely, they both relied on so-called *smoothing constants* determined by the developer to work, something which Khalilian2012 did not have. Our hypothesis is that their performance at any given time would be determined by how well the developer decides on the values for these smoothing constants.

Explorations with all three RTP algorithm led us to the decision of designing our own. We wanted a solution design that had no smoothing constants, similar to Khalilian2012, but with the intuitive behaviour of Khalilian2009 and Kim2002. In the paper by Khalilian et al. they mention that the Khalilian2012 technique was based on their earlier Khalilian2009 technique, but with the *smoothing constants* replaced by calculating the coefficients based on a feedback loop [14]. In our own implementation we started out from Khalilian2012, but modified it such that our final solution's behaviour were more similar to the intuitive behaviour of Khalilian2009, without ever re-introducing smoothing constants. The equations presented in Figure 5.4 were utilised in our modified version.

Modifications made to the algorithm were developed in small iterations over time, where we performed continuous exploratory testing and regularly received stakeholder feedback. A request we received from Sony was to have a normalised priority score from "0% important" to "100% important", if possible without largely affecting the performance. So it was decided to constrain the priority score as a decimal number between 0 and 1. Our proposed RTP

technique is presented with equations in Figure 5.5.

Two terms were defined α (alpha) and β (beta). Alpha determines the criticality of skipping a test case while beta determines how much a priority score should change based on performance history. When a test case is skipped its priority score will increase linearly at the rate of $\frac{fr_k}{2}$ for every successive skip. Skipped test cases with high fail-rates will increase faster than skipped test cases with lower fail-rates. Notice that for $h_k = 0$, i.e. test case *not* skipped, criticality of skipping will be 0. Beta is decided by scaling priority score of a test case from its previous session with the modifier m_k. Essentially, m_k has three outcomes, one for passes, one for fails and one for skips. We let m_k be defined using the fail rate, fr_k. Because of this test cases with low fail-rates will have priority scores drastically increase during a sudden failure (i.e. a regression). On the other hand, test cases with high fail-rates will not increase in priority as much during failures.

$$
\begin{aligned}
PR_k &= min((\alpha + \beta), 1), \quad PR_0 = \gamma, \quad 0 < \gamma \leq 1 \\
\alpha &= h_k \frac{fr_k}{2}, \quad \beta = m_k PR_{k-1} \\
m_k &= \begin{cases} max(1 - fr_k, 0.1) + 1 & \text{if test case failed in session } k-1 \\ fr_k & \text{if test case passed in session } k-1 \\ 1 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 5.5: Implementations for our proposed history-based RTP algorithm. During our thesis, $\gamma = 0.5$. The min function in PR_k definition ensures priority scores are bound to our target interval.

Exploratory testing was done on our RTP technique. We observed that it behaved similar to Khalilian2009 despite not utilising the smoothing constants. However, our RTP technique is still not empirically evaluated to be comparatively as performant as RTP techniques from related work. Therefore, a comparison between our RTP technique and the other three techniques mentioned in this section was conducted. The result of our evaluation are presented in Section 6.1. A cut-off selection criterion where the top percentage among the list of test cases ordered by priority score was put in place, translating the RTP techniques into RTS techniques.

As mentioned in Section 4.2.3 we implemented a historic RTS technique in conjunction with a requirement specification-based RTS technique. Böhme et al. present a testing strategy to divide feature domains into partitions [4]. In their paper, this partitioning was done using a white-box technique called static code analysis.

We propose a new black-box technique for dividing feature domains into partitions based on the structure and relations in a software's requirement specifications. Under the assumption that requirements that are close to each other in the FRS are somewhat connected even in their implementations, we can create partitions based on these relations.

In international standards regarding FRS documents [12, 13], they describe FRS documents to have underlying semantically meaningful structures, something we took notice of when

we presented our explanation of FRS in Section 2.2. Because of this underlying structure, it is possible to generate a tree graph containing all the requirements. Each node in the tree represents a requirement and their relations are represented by the parent-children relations. In our new test case representation presented in Section 5.2, we know that each test case has one or more requirement from the FRS that they validate. Utilising this information, both automatic regression test cases and manual regression test cases were connected to nodes in the tree. With information provided by the most recent automatic testing session we can flag the requirements in the tree that are failing. Then any manual test case that validates a node in proximity with one of these failing nodes can be selected for execution to test that partition of the FRS. An example can be seen in Figure 5.6.

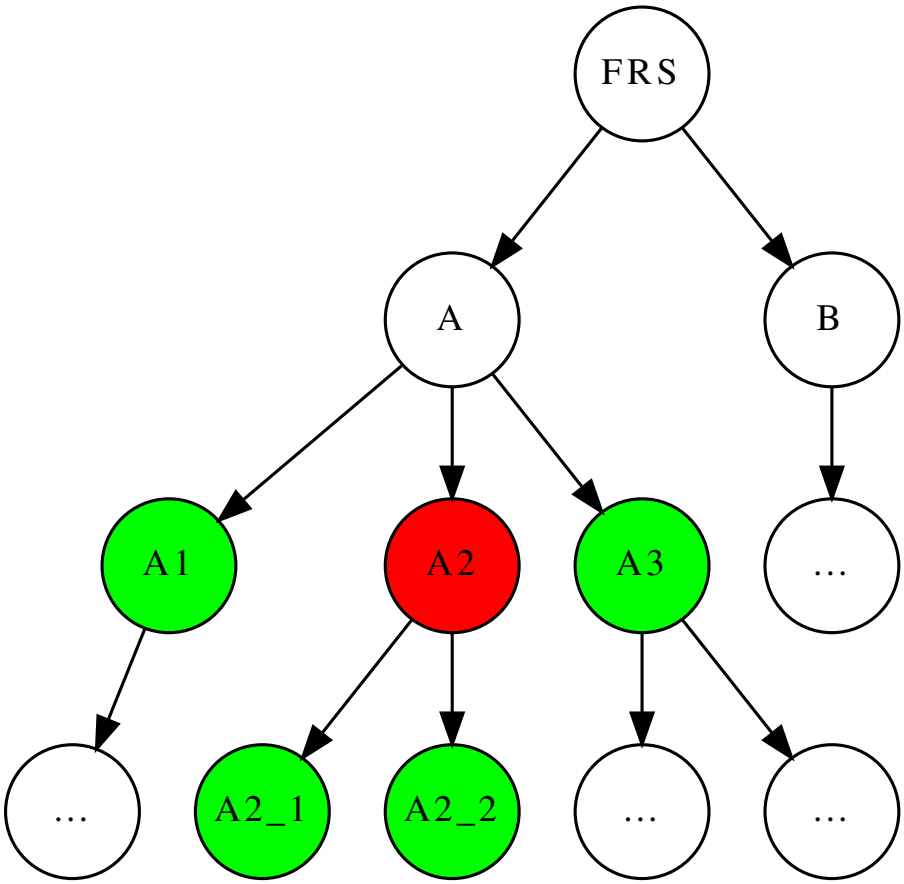


Figure 5.6: Visualisation of our definition on requirements that are in proximity to each other. A2, the red node, represents a requirement that is failing. While its siblings and children: A1, A3, A2_1 and A2_2 (green nodes), are considered to be in proximity of the failing node and are therefore selected.

As long as we can define our concept of proximity between two requirements we can select test cases of interest. We made a design decision to implement a naive criterion and select the siblings and children of the failing node to be part of the partition that we want to verify. Ultimately this technique aims to complement any other regression test selection technique by appending all high risk manual test cases to the pre-existing test plan.

In order to evaluate the impact of the requirement partition selection technique an evaluation was performed. The results of this evaluation is presented in Section 6.2.

Chapter 6

Evaluation

In this chapter we conclude the design science method by presenting the results of our evaluations made throughout the thesis work. Each section presents the evaluation method used and the results we got from applying these methods on our solution. Section 6.1 presents the comparative evaluation we performed to analyse the performance of various history-based RTS techniques in a manual testing context. Section 6.2 presents the value-cost comparison for using two complimentary RTS techniques together. Finally, Section 6.3 presents the field study we conducted at Sony during and after our solution design phase.

6.1 Comparing RTS results by simulation of failure trends

All history-based prioritisation techniques that were studied in related literature had one thing in common. They are specifically meant for automatic regression testing [7, 14, 15]. In our case, where we have a test suite of only manual regression test cases, we could not be certain that any of the studied techniques could be directly applied and still perform as expected. In order to validate our implementation, a method for comparing different history-based prioritisation techniques over random generated sets of trend data is presented in this section together with the results we got from applying this method to our proposed solution. A model operating over random generated data was chosen because currently, we do not have access to large enough sets of historic manual regression test data.

Principles of this evaluation method are presented in Section 6.1.1. Our model's specific design details are presented in Section 6.1.2. Lastly, we present our findings in Section 6.1.3.

6.1.1 Simulation model principles

The idea of our simulation model is to compare the general behaviour of history-based RTS techniques. If we allow RTS techniques to prioritise and select test cases based on each individual test case's past performance, we can compare the output, that is, the test plans that were generated by each RTS technique.

Assuming that a test case has a certain probability of failing, we can randomise the result of executing that test. This randomised result generates a *true trend* for that test case. The *true trend* is a trend consisting only of passes and fails, and is the trend we would get assuming the test case were to be executed during every testing session.

Any RTS technique is essentially a convoluted boolean function that determines if a test case should be selected or not. Naturally, it is impossible to know if a test case covers a faulty state *before* testing is done. Therefore, we must select the test cases based on information that is available to us, in this case the past performance (trend) of test cases. By using the trend as an input to an RTS technique it can decide whether the test case should be executed or not. A *selected* test case therefore gets a 'pass' or 'fail' added to its trend based on the result in the *true trend* for that testing session, while test cases *not selected* gets a 'skip' added to its trend. We call this trend consisting of passes, fails and skips the *actual trend*, the trend we would get if we decide to use that specific RTS technique for our testing. An overview of this can be seen in Figure 6.1.

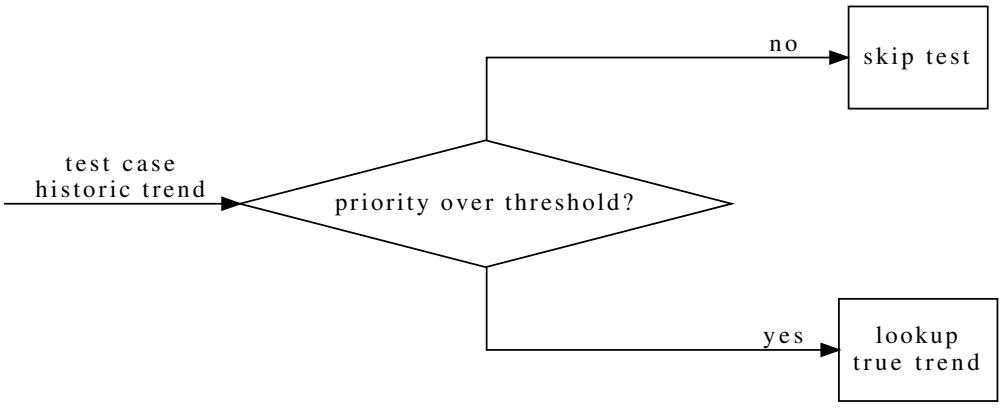


Figure 6.1: Essentially, every RTS technique implements the decision node in the displayed figure. I.e. via RTP techniques and selection criterion, some sort of boolean decision can be made regarding if a test case should be selected or not. Note that the *historic trend* is the test cases *actual trend*.

We imagine the worst case scenario for an RTS technique would be if it never selects test cases that had resulted in a fail, but always select the test cases that results in pass. This is bad because then we would never select any test case that is failing, we would never detect any regression error. Inversely, the perfect scenario would be if an RTS technique always skip test cases that would pass, and always select test cases that would fail. Because we have defined the best and worst case scenarios, we can compare the behaviour of different algorithms by measuring how similar each of them are in relation to these scenarios.

6.1.2 Simulation model design

In order to compare different RTS techniques we must have a common set of data to work on. A test suite was constructed with 36 test cases, each given a failure probability ranging from 0 to 100%. This failure probability reflects the probability for a certain test case to introduce a regression in our simulation. We introduced more test cases with low failure probability. This was a design decision that we made based on real test suites, where generally the majority of test cases would pass. See Appendix A.1 for the complete test suite.

With this test suite we could give each test case a random generated *true trend*, which is a list of passes and fails. The true trend of a test case is common among all the RTS techniques that we are currently comparing. Then, for each RTS technique in our comparison and each test case in our test suite we simulate the testing sessions as described in Section 6.1.1. This gives each test case a unique trend, the *actual trend*, for each RTS that we are comparing.

An example of what our data sets look like can be seen in Figure 6.2. Actual trends and true trends are the core of what data to analyse in order to draw conclusions on the algorithms decision-making behaviour, which is crucial for our **RQ1**.

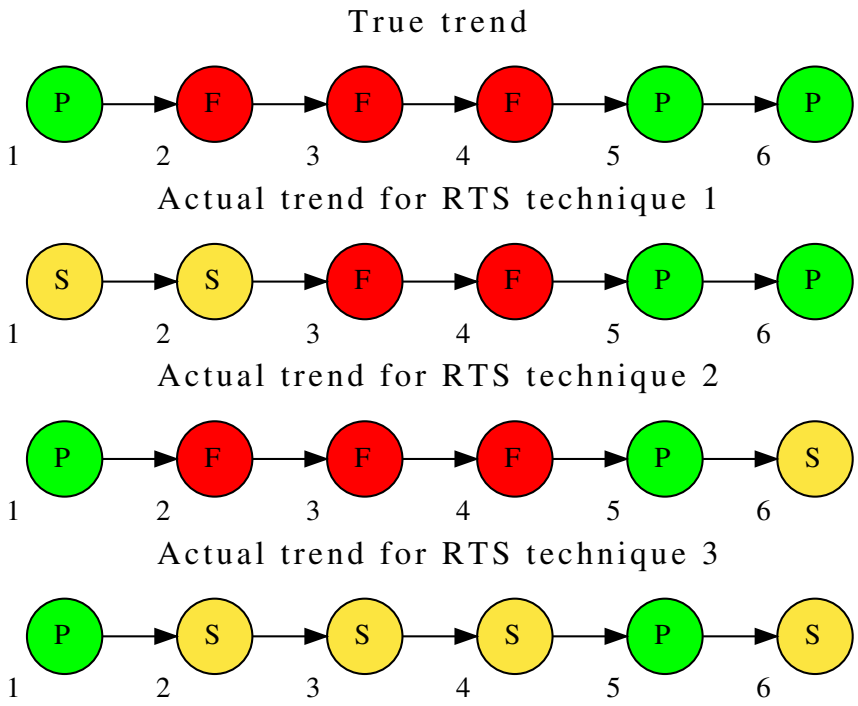


Figure 6.2: Example of how data can be represented when comparing three different selection techniques over the same test case. The true trend is common among them, which creates a baseline for comparison. Then, each RTS technique has their own actual trend that is used for evaluating the performance.

We had to come up with some way of modelling regressions in our simulated trends, rather than relying on the naive approach of homogeneous randomness. In the practical case we do not expect test cases to be flaky, i.e. any test case that have a probability of yielding different

outcomes each time they are re-executed given the same input and procedure. Therefore, we had to somehow guarantee that the stochastic model used was not homogeneously random. Considering this threat, we introduced constraints in the simulation model in an attempt to closer reflect reality. Realistic regressions in a test case's trend will be more reminiscent of a mathematical unit step function, where the moment a regression is introduced will be the point where consecutive passes transitions to consecutive failures. Then, the test case keeps giving consecutive failures until another change fixes the regression where the trend will transition back to consecutive passes.

We decided to set *regression lengths* to three, meaning each time a regression is introduced, three successive failures will be simulated. However, it is probable for high risk test cases to have multiple regressions of length three in a row. Think of the fixed regression length as the time it takes before a regression is fixed. These attributes gave us a stochastic model which achieved the unit step reminiscent pattern that we wanted.

In our simulation, we also introduced a memory variable X representing maximum size of the historic data to input for every testing session. Capping input this way creates a 'moving window' effect allowing us to simulate extremely long true trends, but our algorithms will work with local historic data instead of global. In a real world application there might be a limit to how many sessions of historic data that can be saved. Because we let this window size be a variable we will be able to test the same true trend data multiple times but with different memory sizes. We will be able to record data on how the size of the trend line affects the algorithm's decision-making behaviour.

Our evaluative simulation model comes with two metrics, both of the type less is better. First we count the number of misses which is defined as the number of sessions where an algorithm *skipped*, but the true trend for that session corresponds to *failed*. It is a crucial metric to keep track of since it means that we missed the start of a regression, something we want to avoid as much as possible. Remember, our regressions are 'fixed on its own' after a probable three sessions due to the regression length of three. Any technique that has a decision history where a test case is skipped over three times in a row might miss a regression as a whole. We call this a *full regression miss*, rather than just a miss, as this kind of miss is much more severe. A full regression miss is the worst case scenario, because the prioritisation technique will not get the chance to observe these failures, leading to worse estimations of a test case's performance.

Examples of both singular misses and full regression misses are presented in Figure 6.2. In technique 1, we see an example of a singular miss at the beginning of a regression (at session 2). In technique 2, we see an example of no misses whatsoever. Lastly, in technique 3, we see the worst case scenario, a full regression miss over the whole regression (at sessions 2, 3 and 4). Note how the observed (actual) trend for 'technique 3' in Figure 6.2 is two passes in a row, leading to a perfect performant test case. But in reality, we simply lost the three failures due to poor selections.

6.1.3 Evaluation results

We decided to compare our algorithm with three other prioritisation algorithms, namely the two algorithms proposed by Khalilian et al. from 2009 and 2012 [7, 14] and Kim and Porters prioritisation algorithm from 2002, which was one of the earliest historic based prioritisation algorithms [15]. The three algorithms are recited in Appendix B, while our algorithm is presented in Section 5.4.

The smoothing constant in Kim2002 was given $\alpha = 0.2$, they didn't present exactly what value they used in their testing except for that it was close to zero [15]. For Khalilian2009 we used $\alpha = 0.7, \beta = 0.7$ and $\gamma = 0.04$ with static priority score $PR_0 = 0.5$. This was the same values on the smoothing constants as were presented by Engström et al. in their paper where they implemented this algorithm [5]. The static priority score we used in Khalilian2012 and our proposed technique were the same as for Khalilian2009.

The averaged results from applying the previously described method on these four RTS techniques is presented in Table 6.1. For the complete results, see Appendix A.2. Three different seeds were used to randomise the true trends, with a length of 100000 each, where each data point in the data sets represents a testing session. The simulation was evaluated with four different window sizes; 5, 10, 50 and 100. The cut-off was a constant 60% for all runs. Meaning we select the top 60% from the list of ordered test cases after RTP.

Algorithm	Trend size = 5		Trend size = 10		Trend size = 50		Trend size = 100	
	M	**RM**	**M**	**RM**	**M**	**RM**	**M**	**RM**
Kim2002	3.779	1.155	3.429	1.053	3.178	0.983	3.034	0.938
Khalilian2009	1.580	0.200	1.619	0.324	1.455	0.222	1.433	0.169
Khalilian2012	2.467	0.031	6.271	1.555	5.656	1.197	5.059	0.804
Our design	1.571	0.036	1.335	0.025	1.026	0.032	0.967	0.022

Table 6.1: Average number of misses (**M**) and regression misses (**RM**) occurred per simulated testing session. Less is better, since best case scenario described in Section 6.1.1 is achieved when there are no misses.

Noteworthy from the table, for trend size = 5 we see that our design's performance is close to Khalilian2009 in average number of misses (**M**) and our design's performance is close to Khalilian2012 in average number of regression misses (**RM**). However, we can see that our design consistently have lower occurrences of both misses and regression misses for trend sizes greater than five. Kim2002 never has the lowest value for any metric, however at times its **M** or **RM** values are lower than Khalilian2012.

6.2 Combining RTS results

Our final RTS technique consists of two parts, a prioritisation technique and a requirement partitioning technique. They are presented in Section 5.4. Technically, both the prioritisation and partitioning techniques can each be seen their own RTS technique. However since we use them in conjunction with each other, we need a method to evaluate the cost and value

of combining RTS techniques. Therefore, we propose a method that we use to evaluate the costs and values of combining two RTS techniques.

Three metrics are introduced, *test plan increase* (***TPI***), *regression performance improvement* (***RPI***) and a metric to measure general *performance* (***P***). All three metrics will be calculated by analysing the set of test cases selected (i.e. the test plan) created from each RTS technique, and then evaluate their union. These two sets and their potential overlap are illustrated in Figure 6.3.

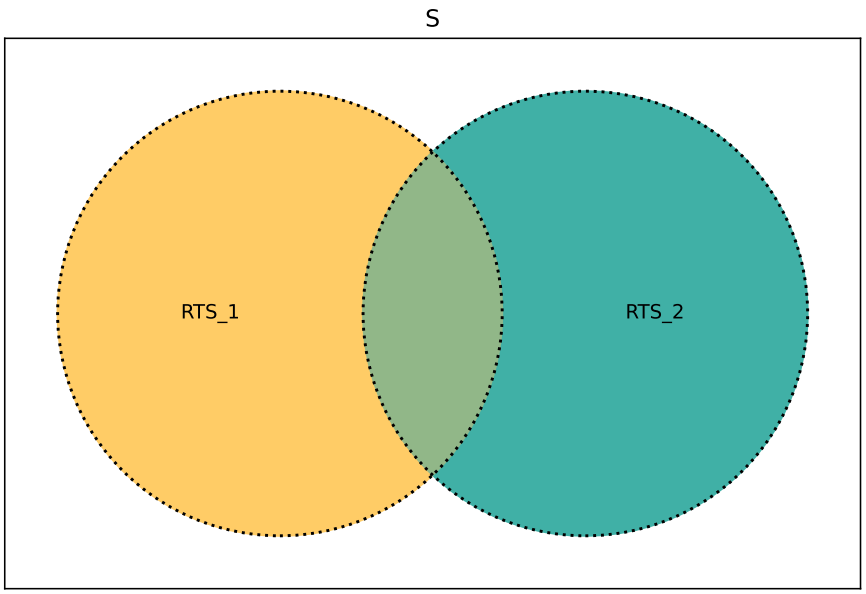


Figure 6.3: Diagram displaying two sets generated from RTS techniques and their potential union. S is the set containing all regression test cases. RTS_1 and RTS_2 are both sub-sets of S.

Test plan increase is defined as the relative amount of test cases added to a test plan after applying both RTS techniques as defined in Equation 6.1. This metric represents the additional *cost* (time) we incur when using the two RTS techniques together.

$$TPI = \frac{|RTS_1 \cup RTS_2|}{|RTS_1|} \tag{6.1}$$

Regression performance improvement is defined as the relative amount of found failures, that would not have been found, had we used only one of the RTS techniques. This metric represents added *value* we get by combining the two RTS techniques. Assuming that $reg(X)$ is a function that returns the number of failed test cases, where X is a selection of test cases, we define RPI in Equation 6.2.

$$RPI = \frac{reg(RTS_1 \cup RTS_2)}{reg(RTS_1)} \tag{6.2}$$

The performance of a selection, X, is evaluated by the metric presented in equation 6.3. P can only be measured if all test cases are executed, that is, you must select-all-test-all in order to measure P. Note that X can be the set of test cases selected by one of the RTS techniques or the union of both.

$$P(X) = \frac{reg(X)}{reg(S)} \tag{6.3}$$

In order to produce our results, we applied the method described from this section to real-world data. We looked into Sony's historic manual regression test data and could retroactively re-construct this test data into our new data representation presented in Section 5.3. Every internally released version had already been manually tested under a select-all-test-all technique. In our retroactive reconstruction we could apply our history-based RTS and requirement partitioning RTS for each version delta, creating the test plans illustrated by Figure 6.3. Some data was lost and could not be re-constructed, but we managed to fully re-construct six internally released versions, creating five version deltas. For every version delta, all metrics mentioned in this section was calculated, and they are presented in Table 6.2. Worth noting from Table 6.2, in version delta 3 the requirement partitioning technique gave added value to the final performance. In the other version deltas it did not add any extra failing test cases to the test plan. In version delta 1,2,4 and 5 it only gave additional costs without any real additional value.

Version delta	TPI	RPI	$P(prio)$	$P(req)$	$P(prio \cup req)$
1	1.0345	1.0	0.833	0.166	0.833
2	1.1176	1.0	1.0	0.429	1.0
3	1.2941	1.25	0.667	0.5	0.833
4	1.1818	1.0	1.0	0.143	1.0
5	1.1250	1.0	0.933	0.2	0.933

Table 6.2: Table displaying the metric values calculated for five version deltas. Each version delta represents newer versions.

6.3 Field study results

Here we present our observations made during the field study described in Section 3.3. The only structure to our field study is the fact that we take notes of any observations we make. We actively engaged in conversation with stakeholders of the project and daily attendance at the Sony office gave us a good understanding of how well adopted and how well received our interventions were.

By observing process adoption rates and perceived value in our work, it enabled evaluative discussions regarding our solution's qualitative performance as a whole. Adoption is observed as yes or no answer to *'did the team adapt to our process?'*. Perceived value is observed via received feedback, expressions from stakeholders after presenting our work and asking developers directly about their experiences with the new process.

The team provided code reviews during our migration from the old system (no tools and Markdown) to the new systems (rich tools and YAML). They were engaged in the changes, asked questions and discussed the design. Many developers approved the changes before the migration was merged to the main branch, indicating satisfaction over these changes.

The project owner ordered a high level presentation of our tools, highlighting the overall new opportunities the tools bring to product quality. All design requirements were presented and examples of how to use the tools in practice was explained. Project owner had nothing to complain about, was excited and saw big potential in how costs can be saved. Not only the fact that we run less tests per testing session, but he saw cost reductions in the testing tool presented in Section 5.3, because the tool removed some cognitive load in the workflow of a tester. Since previously manual regression testing data was compiled manually, doing this via a tool should make the task more efficient.

After migration from the old system to the new system was made to the main branch, the system was fully integrated and readily available to the team. They took over development from that point onwards. The team immediately started to prioritise tasks related to scaling our project to cover more test suites than just the one we covered in our thesis work. The team commented on matters such as YAML being at least as intuitive as Markdown when writing a new test case. We got active comments covering the ease of code review on YAML versus Markdown. The team expressed satisfaction in the improvement for reviewing manual test case representations, not only because of the readability of YAML, but also due to the usefulness of the verification tool.

The scrum master already had self-authored analytical tools to present automatic regression test data in charts and tables. He was one of the interviewees, and he expressed interest in the ability to represent manual regression testing data on an equivalent format as automatic regression testing data. It would then become possible to re-use the tools but input manual regression test data instead of automatic regression test data. During the course of our field study, i.e. our time doing our thesis work at the Sony office building, the scrum master did not explicitly mention that he was able to use the new manual regression test data meaningfully (nor did he express that any attempts had been made). However, he did express excitement over the capabilities provided given that there is now coherence between the data sets.

The lead architect was not heavily involved with our design decisions, but from time to time he gave us valuable inputs on how we can best integrate our tools into their internal ecosystem of tools. Moreover, he presented ideas of internal future work that can now be made using the new manual regression test data. Before our work, they manually compiled tables of manual regression testing data. One task he saw could be solved immediately after our interventions were to automate the compilation of the exact same table. Since the tables existed primarily so that any non-technical stakeholder, such as managers and project owners, could get an overview of regression testing progressions.

Adaptation of tester tool was rather slow in comparison to the manual test case representations in YAML, but once used initial reports were positive. Some small suggestions in quality of life improvement of the software were given by the developers, showing that they see a usefulness in the tool. They liked that this tool abstracts away the details, now the tester only have to worry about executing the test cases and reporting the results.

Chapter 7

Discussion

In our thesis we have conceptualised the regression testing problem of Sony and established two recommendations for solving it in the form of two technical rules. Using design requirements derived from our rules, a solution to the regression testing problem was designed and evaluated. This structure of problem conceptualisation, solution design and evaluation fits the paradigm of design science. In this chapter, we first discuss our design science method as a whole in Section 7.1. In Section 7.2 we progressively orient our discussion to a technical discussion surrounding our solution design by discussing any design decisions that were made. Section 7.2 is divided into smaller pieces, each discussing the results from our evaluations.

7.1 Method

Early on, in the planning of this thesis work, we struggled to settle on what research methods to use for solving problems in industry contexts. Our stakeholders at Sony had a problem that they would like to see solved, no matter what research findings could be made. For this to be a proper thesis, we are required to have an academic perspective leading to academic contributions. After initial research, we found DSM which made us realise that any research method we were to select and that qualifies as design science would likely help us cover both academic expectations and industry expectations. The three DSM phases can be used to help researchers in industry contexts to map industry activities to research activities [22].

Interactions with all stakeholders at Sony helped us create tangible intermediate constructs and artefacts useful for our report, such as the research questions and design requirements. Design requirements helped us implement solutions in a way where we always knew that we are developing towards the same end goal. They also helped us to structure our academic

work where we could reflect on design decisions in our thesis report. While design requirements are guidelines on what interventions to introduce in order to help Sony with their problem, research questions represent what we want to explore and contribute to academia. The research questions are based more in the literature study rather than the interviews, and answers to them were gained largely during the evaluation phase. For example, we decided to utilise FRS as a data source for a selection technique as a direct response to address the identified research gap in literature for that area. While the evaluation of combining RTS techniques provided us with insight for answering our research question.

The method we used when searching for relevant literature worked well overall. It was easy for us to find literature pertaining to regression testing and optimisation of regression testing. However, because of the large amount of scientific work that have gone into regression testing there is a lot to process when looking for specific answers. Literature reviews [1, 6] were key to navigate this vast domain of research. When we found relevant or interesting resources we noted a relevance score, short summary and their keywords for each of them in a shared document. Despite this, it was sometimes hard to remember what paper talked about what areas. We think that this is probably because pretty much all of our research material are within the subject regression testing with some specific twist. This made it harder to specifically remember what each paper said or did not say. Also, many of our summaries might have been too detailed, making them harder to overview.

We chose to conduct semi-structured interviews over structured interviews because we did not want to put constraints on ourselves during the elicitation process. Had we enforced a stricter structure, let us say a questionnaire with well-defined questions, then we feared that these answers would not contain enough nuance in them or interviews coming across as interrogative. The semi-structure worked really well for our purposes. Interview subjects were engaged during all sessions, which helped us to elicit information by asking follow-up questions that otherwise might have been missed in a stricter structure. All sessions turned out to be like an intellectual conversation over Sony's regression testing problem.

Finally, the agile way of working via Kanban with focus on implementation order disregarding time estimation was helpful during the solution design phase. There were always a clear prioritisation of what task to do next during our whole thesis work. The primary issue with prioritisation by implementation order rather than a time estimation was feature creeping. It was easy for us to add more features to our backlog without considering the amount of time that would be needed to implement all of them. We actively realised this risk of feature creeping, therefore we took time to go through the backlog and sort out all features that could be considered as 'extras'. Because of our extra caution, feature creeping did not hurt our thesis work negatively, but the fact of the matter that there was a risk for not finishing in time had we filled our backlog carelessly.

7.2 Execution

This section presents all discussion topics regarding the execution of our solution design and evaluation phases during our thesis work. Specifically, we will present all discussion behind our design decisions in 7.2.1. Then, we will present in-depth discussions regarding our evaluation results divided into our designed history-based RTS technique in Section 7.2.2, and our exploratory requirement partitioning technique in Section 7.2.3.

7.2.1 Design decisions

Our solution design phase involve making active design decisions. These decisions have to be made with underlying facts, which are presented in the form of design requirements and were decided after a proper problem conceptualisation phase. We present three design requirements in Section 4.2.2. Then we present our implementations of those design requirements in Chapter 5. In this section, we discuss our reasoning behind all design decisions that were made in Chapter 5. We also discuss whether our solution design actually satisfies our design requirements.

YAML was chosen as the data serialisation language to use for representation of manual regression test cases. We were convinced early on that whatever representation we end up with, it had to be properly machine-readable. However, since developers maintain and execute these test cases, they had to be human-readable as well. Within AOSP, XML is already used as a sort of default data serialisation language, implying Sony's team already had technical knowledge and experience with XML. It was not a coincidence that even though the team had experience with XML, they had previously opted for a human-readable format for their representations, namely Markdown. This indicates that human-readability was an important aspect to Sony prior to our thesis work. Because of this, we argue that YAML is a well motivated selection of a data serialisation language to use in our case. Moreover, the change request migrating from Markdown to YAML was met with positive reception as noted by our field study, indicating on work well done.

In the end, we decided that our easiest solution to represent regression testing data equivalently no matter if the data is from manual or automatic executions were to represent them identically. This design decision was found to be trivial in the context of black-box testing a DUT, which was presented in Figure 5.1. We quickly realised that the only differences in this context were the actor and the data collection. In both the manual and automatic case, we can abstract a testing activity to its fundamentals; an *input vector* with a *testing procedure* yielding pass or failed when compared to *expected results*. This is why we opted to create an interactive testing tool, that presents input vector and testing procedures to a manual tester, then prompts them whether the test case passed or failed. Enabling our testing tool to format testing data identically to AOSPs, creating equivalent results.

As mentioned in our result Section 5.4 three issues were identified in history-based prioritisation from related work. The only algorithm with no smoothing constants, Khalilian2012 had its priority scores approaching zero no matter the input, creating the first issue of zero-drifting scores. Secondly, smoothing constants was seen as a suboptimal implementation constraint, so any existing implementation with these constants are discarded. Thirdly, pri-

ority scores from the three algorithms were unintuitive and did not represent anything, other than a higher value equals a higher priority.

We noticed how zero-drifting scores could be mitigated for smaller trend sizes, however we cannot guarantee anything about trend sizes at Sony. We could set an upper limit for the trend size, but we do not see the need to limit ourselves to using fewer data points than what is potentially available. Another issue with zero-drifting scores would be that a newly introduced test case would get a comparatively high priority score for a long time after their introduction. Any test case that gets added will receive its static priority, which is going to take some time to catch up with the other test cases. Even if the new test case would consecutively pass over time, the rest of the test suite containing old test cases would have a comparatively small priority score because they already had time to drift towards zero. The zero-drifting also threatens priorities to get rounded off to zero, in which case the algorithm stops working entirely.

Two of the options that were studied, the algorithm from 2009 by Khalilian et al. and Kim and Porter's algorithm from 2002 [7, 15], did not experience the issue of zero-drifting scores. Instead, they suffer from the suboptimal design decision of including smoothing constants, which are parameter values set by developers. Software development projects progress over time and while they progress, different aspects becomes more important while others less important. For instance, it would be reasonable to assume that early in a project's life-cycle, feature implementations are of the essence, but after some time as the code base grows more complex, more regression testing activities has to be actively made. This raises the doubt that smoothing constants can be set early on to some values, then stay at those values during the whole project's life. They will most likely have to change just as any other software. Therefore, somebody has to be responsible for the fine-tuning of the smoothing constants based on their intuitive feeling of where the project is heading next. Because of all this, we argue that by including smoothing constants in the implementation makes these two algorithms less maintainable. Khalilian et al. solved this issue in their improved algorithm from 2012 with feedback loops, trading maintainability for some added design complexity.

The final issue that we identified was that the priority scores did not have any explicit meaning. For Khalilian2012 all test cases would start at their base priority score, then approach zero at different rates based on their performance. Implying that priority score of a test case should be lowered the older it gets, which is not necessarily true. Kim2002 and Khalilian2009 had priorities ranging between zero and infinity. A rather extreme example would be a failing test case that has a low criticality, it would get executed every session with fails until the developers introduce a fix for the regression. The priority of this test case would continuously increase towards infinity, making it significantly more important than any other test case. To combat this problem, our priority scores were constrained between zero and one. This lets us represent each test case's importance as a percentage value for the current session, which gives us the ability to select test cases in intuitive ways. One example could be to select all test cases with a priority of **0.5** or higher, meaning any test case with an importance of at least **50%** to be executed would get selected. Or, we could allow new test cases to be represented as **100%** important to be executed in the next session by forcing their priority score to be one.

Our chosen selection criterion where we selected the top x percentage of the test suite ordered

by priority score. We argue that such a cut-off selection gives an even spread of workload each session. If you select based on all test cases above **0.5** priority, some sessions there will have more work compared to others, while with a selection of the top **50%** you will always get half the test suite. This makes it easier to estimate how much time is needed to be spent on testing for each session.

In the second half of Section 5.4, we present an implementation of requirement partitioning as an RTS technique. Initially, we opted for a naive approach to the problem, focusing on providing any reasonable selection rather than a good selection. The idea was to then iterate on our solution design and improve our selection over the course of our thesis. During our planning phase, requirements specification partitioning were one of the main research points motivated by the research gap in related work. Because there were no related work to take inspiration from and because Khalilian2012 did not meet our quality expectations, it created two challenges for us.

First, we had no strategy on how to evaluate our requirements partitioning RTS technique. All metrics and evaluation methods presented in Section 6.2 are first defined by us. The method presented require two primary data sources: Test plans for each version generated from any other RTS technique and complete regression test data for each version generated by a select-all-test-all strategy. Test plans were impossible for us to retroactively generate before our history-based RTS technique was working well. So we could not even begin quantifying the quality of selections made in our requirements partitioning before our simulation analysis, presented in Section 6.1 was fully completed, leading to our second challenge.

Time-constraints were our second challenge for iterating our requirements partitioning design. We expected Khalilian2012, an empirically evaluated RTS technique, to work well enough to satisfy our quality expectations with minor to no adjustments from the implementation presented in their original paper [14]. It was not the case, during our initial exploratory testing Khalilian2012 did not do a good job in selecting manual regression test cases. Time was spent on designing a new history-based RTS technique, which we did not expect during planning. This meant we had less time than planned to improve on our requirement specification partitioning technique. The final design proposed in Section 5.4 was not the first design. We still managed to find time to execute more than one iterative cycles over the design. In our first design, we opted to select a failing requirement's node and all its children recursively, i.e. the whole subtree created by considering the failing requirement's node as root. Because this would lead to select-all-test-all test plan should the actual root node be a failing node, we improved from this first design to the one proposed.

7.2.2 History-based selection technique

In this section we discuss the results presented in Section 6.1. We present our interpretation of the results from Table 6.1. Moreover, we discuss how trend size affects the results and how this relates to manual regression testing, before we finish our discussion by presenting why we think our RTP algorithm performed the way it did.

Overall, our RTP algorithm performs well compared to other empirically evaluated RTP algorithms. It performs slightly better than Khalilian2009, which was the best performant among the three empirically evaluated. For smaller trend sizes, like the size five trend we

presented, Khalilian2012 performed well on regression misses. However, our RTP algorithm performed the best when measuring individual misses, although only slightly better than Khalilian2009. Khalilian2009 performed good on the regression miss metric as well, but not nearly as good as our RTP algorithm or Khalilian2012. Hence, it can be seen that our technique has utilised ideas and taken inspiration from both of these techniques, getting the strong points from each.

When trends grow larger (10, 50 or 100) our technique starts to outperform all others. Khalilian2012 quickly gets worse as the trend size increases, even getting outperformed by the oldest of them all, Kim2002. For individual misses, only Khalilian2009's performance is close to ours, while for full regression misses ours clearly outperform the best among the other RTP algorithms, Khalilian2009.

An important aspect when it comes to trend size is that the larger the trend size, the older data you get. It is important to remember that we are working with manual regression testing, so the time between each test session is going to be longer than for automatic regression testing. In the case of automatic regression testing it is not unusual to run tests every night. However, for manual regression testing, running them once a week would be considered a lot. Once or twice a month would be more realistic estimation on how often software development projects would conduct manual regression testing. In fact, in a survey study by Haas et al. [9] among 38 respondents suggest that the median number of test cycles that are executed were as low as 4.5 times per year. This means that a trend size of 10 for an RTS technique intended for automatic regression testing could use data from the last couple of weeks, while for an RTS technique intended for manual selection could mean data going from several months to years.

At some point historic data will also become outdated. Development projects are constantly changing, and the features that was worked on half a year ago might not see frequent changes today. So prioritising a test case based on if it failed a relatively long time ago can be highly inaccurate. On the flip side, using fewer data points for the trend and only using more recent data can be problematic, since these algorithms need at least a few data points in order to become accurate. Specifically in our case where we scale the β term depending on the fail rate. Over time, the fail rate should become more accurate because of the increase in sample size.

So why is it that our algorithm can outperform the others? We have a few hypotheses regarding this. Firstly, we believe that our utilisation of the fail rate as a variable when calculating priority scores allows us to control the changes in priority in a more intuitive way. We can make the priority increase quickly when a regression has been identified, especially for a test case that rarely fails. Secondly, we place a heavier emphasis on consecutive skips, since we designed our algorithm specifically for manual testing we could not allow for too many consecutive skips to happen. The empirically evaluated algorithms are all designed for automatic testing, so the criticality of skipping a test case, let's say five times in a row, is not that high. In an automatic regression test setting five consecutive skips means maybe one week of not executing the test case, while for manual testing it could mean maybe 5 months of no execution. We believe this emphasis where skips are critical is why our algorithm performed very well when it came to detecting full regression misses.

7.2.3 Requirement partitioning

We now discuss the results presented in Section 6.2. The discussion is centred around the value to cost comparison of using the FRS partitioning technique. The two metrics introduced, ***TPI*** and ***RPI***, respectively represent cost and value, allowing for a discussion to take place.

From Table 6.2, we swiftly note that for all sessions, costs were added. This is expected because our complementing RTS cannot remove any test cases from a test plan, it works additively. In four out of five sessions combining FRS partitioning did not provide any increase in value, although it did always discover some failing test cases. The one time when it generated value was in version delta 3, the same version delta when it also had the highest TPI (test plan increase). Combined performance of our techniques was generally good, at least 83% of all regressions were discovered. However, it is very clear that our historic prioritisation algorithm does the majority of the work. Conveniently, the one version delta when the primary RTS did not perform well was the same singular version delta where our requirements partitioning generated additional value. This time, increasing the overall performance by 25%. This is the exact sort of behaviour that we envisioned during our solution design phase. Requirement partitioning stemmed from the idea to explore if confidence in any RTS technique can be boosted by using FRS as a data source. The requirement partitioning technique is designed to act as sort of safety net for when the primary RTS have the occasional bad selection. Increase in confidence is rather difficult to quantify. But if this pattern is consistent, where our requirements partitioning will step up every time our primary RTS performs poorly, it could likely build confidence among developers over time.

There are multiple factors that affect the performance of requirement partitioning. It has a strong dependence on the structure of the test suite since it requires automatic test cases to fail to get an indication of what manual test cases to select. If some part of the system is only covered by manual test cases, these tests can never get selected since there is no automatic test case in proximity that could fail. Therefore, we argue that ideally every manual regression test case should have at least one automatic regression test case in its proximity. Another issue is that some test cases might verify multiple requirements. If an automatic test fails, all the requirements that it is verifying will be marked as failing. This can result in a rather large partition being selected. The requirements that each test case verifies is determined by the developers, so there is also a risk of human error when constructing the traceability between requirement and test cases. We argue that a better scenario would be if each test case only verify one requirement and each requirement is only verified by one test case, creating a one to one relationship.

Another factor that will affect the performance of requirement partitioning is how one defines the partitions in the FRS. A generous definition will result in more test cases being selected, increasing cost. A stricter definition will result in less cost but also a lower probability of it yielding extra value. Exactly how to balance this type of selection is something we believe can be looked into in future work on this area. We have established a foundation for the idea of requirement partitioning, but it requires more research and testing for it to become something generally applicable.

Chapter 8

Conclusion

In this chapter we answer all research questions presented in Chapter 1 by drawing conclusions based on discussions made in Chapter 7. For the sake of readability, we re-introduce our research questions before drawing our conclusions.

RQ1 What are the differences in prioritisation of automatic versus manual regression tests for the purpose of cost reduction?
RQ2 How does utilising the requirement specification in a selection algorithm for manual regression tests affect the efficiency?
RQ3 Is our implementation *perceived* as valuable and useful to the team at Sony?

RQ1

Empirically evaluated RTS techniques in published work have a tendency to be designed with automatic regression testing in mind. We argued that for prioritisation it does not matter if man or machine executes the testing, as long as the data collected can be used as an input to an RTS technique. Despite this, we experienced difficulties in successfully applying three empirically evaluated RTS techniques, all three to be known for automatic RTS. A major difference between automatic and manual regression testing is the frequency in which test cases are executed. Any historic test data collected for manual test cases risks becoming obsolete before the sample sizes grow large as a direct consequence of the low execution frequency. If those automatic RTS techniques were to be used as manual RTS techniques, they will therefore have to be re-adjusted in a way where they can perform well under small sample sizes. After we proposed a new RTS technique specifically designed to quickly amplify

priority score as soon as regressions were discovered, we managed to outperform all other RTS techniques evaluated during our thesis. Therefore, we conclude that the test execution frequency is a crucial difference when comparing between manual and automatic regression test prioritisation and selection.

RQ2

Partitioning FRS documents is demonstrated as a useful data source for complementary RTS techniques. Added value from requirements partitioning depends on completeness and traceability aspects of the underlying FRS, as well as the performance of the primary RTS technique. For our case we can conclude that the FRS partitioning technique always discovered some failing test cases. However, in four out of five data points these same test cases were also selected by the primary RTS technique. In one data point it provided an increase in value by selecting failing test cases that were not discovered by our RTS technique. Overall our efforts serve as a proof of concept that requirement partitioning could be used in order to boost efficiency of an RTS technique. However, during our thesis we did not have enough time to thoroughly explore and improve upon our initial proof of concept. Therefore, we conclude that while requirements partitioning can affect efficiency positively, our concepts demonstrated in this thesis has to be applied on future work before we can quantify the potential efficiency gain.

RQ3

Breaking down what interventions were made during our design science, we have three things. A new way for the team to represent, trace, validate and version control their manual regression testing, along with RTS tools to support these activities. A new standardised structure where test data can be collected and analysed, disregarding if their origin is from automatic regression test cases or manual regression test cases. Lastly, we incorporated a history-based RTS technique in conjunction with a partition-based RTS technique, which enables the team to orient their testing activities around cost-optimised test plans. Because manual regression testing has a low frequency of testing session, adaptation of our RTS tools will naturally be slow initially. Overall, the reception of our interventions has been positive and adaptation rates for interventions such as test case representation models were fast. Therefore, we conclude that overall our interventions in our design science are perceived as valuable to the team members.

Technological rule

Looking back on our design requirements formulated in Section 4.2.2, we can see that all of them have been fulfilled. Manual test cases are now represented in YAML making them machine-readable as well as human-readable. Manual testing data is now presented in the exact same structure as the automatic data. Finally, we can see an improvement in the cost-effectiveness of manual testing from using our intervention.

Our design requirements were created with the intent of helping us achieve our technological rules that act as our general goal for our thesis work. The technological rules are expressed as a recommendation from us to others in the field who are potentially trying to solve the

same problem that we have. Given the positive evaluative results discussed and concluded from our research questions, we conclude our thesis work by re-presenting our technological rules, but this time as a valid recommendation:

> To improve cost-effectiveness of regression testing in the context of manual executions, apply a historic RTS technique.
>
> *– and –*
>
> To further improve regression test performance from regression test selection, apply a REQP technique in conjunction to the original RTS technique.

Future work

For the requirement partitioning technique there is a lot of potential future work. It would be good to apply this technique on different systems with different FRS's and also to try and optimise the selection of partitions. Unfortunately we did not have enough time to perform a more comprehensive evaluation of this technique. However, we consider our work done here as a first step to bridge the research gap surrounding the utilisation of FRS in regression test selection and prioritisation.

We would also like to see our prioritisation algorithm more thoroughly evaluated on different systems, potentially even for automatic regression testing. Especially considering that most research is on automatic RTP, and we attempted to apply three of these techniques in a manual context with little success. How to apply a manual RTP technique, and how well it works, in an automatic setting could be an interesting research topic.

References

[1] Nauman Bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. On the search for industry-relevant regression testing research. *Empirical Software Engineering*, 24(4):2020–2055, 2019.

[2] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. YAML ain't markup language (yaml) version 1.2. `https://yaml.org/spec/1.2.2/`, 2021. Accessed: 2023-02-13.

[3] Georg Buchgeher, Christian Ernstbrunner, Rudolf Ramler, and Michael Lusser. Towards tool-support for test case selection in manual regression testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 74–79, 2013.

[4] Marcel Böhme, Bruno C. D. S. Oliveira, and Abhik Roychoudhury. Partition-based regression verification. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 302–311, 2013.

[5] Emelie Engström, Per Runeson, and Andreas Ljung. Improving regression testing transparency and efficiency with history-based prioritization – an industrial case study. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 367–376, 2011.

[6] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[7] Y Fazlalizadeh, A Khalilian, M Abdollahi Azgomi, and S Parsa. Prioritizing test cases for resource constraint environments using historical test case performance data. In *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pages 190–195. IEEE, 2009.

[8] Benny Godlin and Ofer Strichman. Regression verification. In *2009 46th ACM/IEEE Design Automation Conference*, pages 466–471, 2009.

[9] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. How can manual testing processes be optimized? Developer survey, optimization guidelines, and case studies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1281–1291, 2021.

[10] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Lars Heinemann, Rudolf Vaas, and Peter Braun. Hunting for smells in natural language tests. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1217–1220, 2013.

[11] Hadi Hemmati, Zhihan Fang, and Mika V Mäntylä. Prioritizing manual test cases in traditional and rapid release environments. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE, 2015.

[12] IEEE. Recommended practice for software requirements specifications. *IEEE Std 830-1998*, pages 1–40, 1998.

[13] ISO/IEC/IEEE. International standard - systems and software engineering – life cycle processes – requirements engineering. *ISO/IEC/IEEE 29148:2018(E)*, pages 1–104, 2018.

[14] Alireza Khalilian, Mohammad Abdollahi Azgomi, and Yalda Fazlalizadeh. An improved method for test case prioritization by incorporating historical test case data. *Science of Computer Programming*, 78(1):93–116, 2012. Special Section: Formal Aspects of Component Software (FACS'09).

[15] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 119–129, New York, NY, USA, 2002.

[16] R Krishnamoorthi and SA Sahaaya Arul Mary. Factor oriented requirement coverage based system test case prioritization of new and regression test cases. *Information and Software Technology*, 51(4):799–808, 2009.

[17] Remo Lachmann, Michael Felderer, Manuel Nieke, Sandro Schulze, Christoph Seidl, and Ina Schaefer. Multi-objective black-box test case selection for system testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1311–1318, 2017.

[18] Soren Lauesen. *Software Requirements Styles and Techniques*, chapter 9, pages 376–380. Pearson Education Ltd., thirteenth edition, 2002.

[19] Qi Li and Barry Boehm. Improving scenario testing process by adding value-based prioritization: an industrial case study. In *Proceedings of the 2013 International Conference on Software and System Process*, pages 78–87, 2013.

[20] Philipp Offermann, Olga Levina, Marten Schönherr, and Udo Bub. Outline of a design science research process. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, pages 1–11, 2009.

[21] Akira K Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, 1998.

[22] Per Runeson, Emelie Engström, and Margaret-Anne Storey. The design science paradigm as a frame for empirical software engineering. In *Contemporary empirical methods in software engineering*, pages 127–147. Springer, 2020.

[23] Softhouse. Kanban in five minutes. `https://www.softhouse.se/wp-content/uploads/In5_Kanban_en.pdf`. Accessed: 2023-02-13.

[24] Margaret-Anne Storey, Neil A Ernst, Courtney Williams, and Eirini Kalliamvakou. The who, what, how of software engineering research: a socio-technical framework. *Empirical Software Engineering*, 25(5):4097–4129, 2020.

[25] Markdown Team. Getting started – an overview of markdown, how it works, and what you can do with it. `https://www.markdownguide.org/getting-started/`. Accessed on: 2023-02-13.

[26] Patra Thitisathienkul and Nakornthip Prompoon. Quality assessment method for software requirements specifications based on document characteristics and its structure. In *2015 Second International Conference on Trustworthy Systems and Their Applications*, pages 51–60, 2015.

[27] Claes Wohlin and Per Runeson. Guiding the selection of research methodology in industry–academia collaboration in software engineering. *Information and Software Technology*, 140:106678, 2021.

[28] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

Appendices

Appendix A

Raw data for selection algorithm evaluation

This appendix contains the raw data used to produce the results in Section 5.4.

A.1 Simulated test suite

First, the list of all test cases that were used are presented in a condensed way. We only specify the unique traits of a given test and how many test cases share those traits. In our Discussion chapter, we motivate why some duplicates are used.

# of test cases	Risk for regression
1	100 %
1	95 %
1	85 %
1	75 %
1	65 %
1	55 %
1	45 %
1	35 %
1	25 %
1	15 %
1	14 %
1	13 %
1	12 %

# of test cases	Risk for regression
1	11 %
2	10 %
2	9 %
2	8 %
2	7 %
2	6 %
2	5 %
2	4 %
2	3 %
2	2 %
2	1 %
2	0 %

Table A.1: The raw representation of all test cases and their risk of introducing a regression used for simulating and evaluating the performance of selection algorithms.

A.2 Simulated test results

All the simulated test runs made in order to evaluate algorithm performance are presented in the following tables. Cut-off was at 60% for all runs.

algo	sum	reg. miss
KP	345175	105126
K09	158107	20034
K12	249639	3663
Our	156405	3522

algo	sum	reg. miss
KP	368471	113161
K09	157875	20064
K12	238721	2242
Our	156776	3586

algo	sum	reg. miss
KP	420108	128098
K09	158004	20023
K12	251796	3511
Our	158045	3696

(a) Three random seeds with trend size 5.

algo	sum	reg. miss
KP	307280	94038
K09	161524	32268
K12	630861	158588
Our	133083	2540

algo	sum	reg. miss
KP	366249	112555
K09	162116	32485
K12	627744	152310
Our	133301	2502

algo	sum	reg. miss
KP	355286	109425
K09	161938	32430
K12	622735	155455
Our	134033	2538

(b) Three random seeds with trend size 10.

algo	sum	reg. miss
KP	303878	92991
K09	145232	22003
K12	574564	123961
Our	102477	3153

algo	sum	reg. miss
KP	353134	108966
K09	146012	22259
K12	550163	112574
Our	102333	3166

algo	sum	reg. miss
KP	296292	93055
K09	145300	22374
K12	571955	122522
Our	103044	3213

(c) Same three random seeds but with trend size 50.

algo	sum	reg. miss
KP	290410	88849
K09	142720	16804
K12	511530	87275
Our	96444	2143

algo	sum	reg. miss
KP	353134	108966
K09	143231	16945
K12	490165	63734
Our	96613	2242

algo	sum	reg. miss
KP	266726	83730
K09	144067	17073
K12	515947	90072
Our	97149	2230

(d) Same three random seeds but with trend size 100.

Table A.2: We selected three different random seeds and four different trend sizes. This created 12 configurations to run.

Appendix B

Algorithms used in comparison

The following algorithms were used in our comparative evaluation. We include them in our appendix because our results presented in 6.1 are heavily dependent on these implementations. Everything presented here Appendix B are work done by others, not the authors for this master's thesis.

$$
\begin{aligned}
PR_0 &= \text{The percentage of code coverage of the test case with respect to n different coverage criteria / n} \\
PR_k &= \frac{\alpha h_k + \beta PR_{k-1}}{k}, \quad 0 \leq \alpha, \beta < 1, k \geq 1 \\
\alpha &= \left(1 - \left(\frac{fc_k + 1}{ec_k + 1}\right)^2\right)^{h_k} \\
\beta &= \left(\frac{fc_k + 1}{ec_k + 1}\right)^x, \quad x = \begin{cases} \mathbf{1} & \text{if the test case has revealed some faults} \\ \mathbf{2} & \text{if the test case has not revealed any fault} \end{cases} \\
fc_k &= \sum_{i=1}^{k-1} f_i, f_i = \begin{cases} \mathbf{1} & \text{if test case has revealed some fault in test session i} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
ec_k &= \sum_{i=1}^{k-1} e_i, e_i = \begin{cases} \mathbf{1} & \text{if test case has been executed in test session i} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
h_k &= \begin{cases} \mathbf{0} & \text{if test case has been executed in test session k-1} \\ h_{k-1} + 1 & \text{otherwise} \end{cases}
\end{aligned}
\tag{B.1}
$$

Figure B.1: Prioritisation algorithm as presented by Khalilian et al. [14]. Note that this algorithm does not have any smoothing constants, which the following algorithms will have.

$$
\begin{aligned}
PR_0 &= \text{The percentage of code coverage of the test case} \\
PR_k &= \alpha h_k + \beta PR_{k-1} + \gamma HFDE_k, \quad 0 \leq \alpha, \beta, \gamma < 1, k \geq 1 \\
HFDE_k &= \begin{cases} \mathbf{0} & \text{if the test case has not been executed yet} \\ fc_k/ec_k & \text{otherwise} \end{cases} \\
fc_k &= \sum_{i=1}^{k-1} f_i, f_i = \begin{cases} \mathbf{1} & \text{if test case has revealed some fault in test session i} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
ec_k &= \sum_{i=1}^{k-1} e_i, e_i = \begin{cases} \mathbf{1} & \text{if test case has been executed in test session i} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
h_k &= \begin{cases} \mathbf{0} & \text{if test case has been executed in test session k-1} \\ h_{k-1} + 1 & \text{otherwise} \end{cases}
\end{aligned}
\tag{B.2}
$$

Figure B.2: Prioritisation algorithm as presented by Khalilian et al. in [7]. α, β and γ are smoothing constants determined by the developer, where γ must be smaller than α and β.

$$
\begin{aligned}
P_0 &= h_1 \\
P_k &= \alpha h_k + (1-\alpha)P_{k-1}, \quad 0 \leq \alpha \leq 1, k \geq 1 \\
h1_k &= \begin{cases} \mathbf{1} & \text{if test case was executed in the previous test session} \\ \mathbf{0} & \text{otherwise} \end{cases} \\
h2_k &= \begin{cases} \mathbf{1} & \text{if test case failed in the previous test session} \\ \mathbf{0} & \text{otherwise} \end{cases}
\end{aligned}
\tag{B.3}
$$

Figure B.3: Prioritisation algorithm as presented by Kim and Porter in [15]. $h1_k$ and $h2_k$ are two possible methods for determining the value of h_k that were presented by Kim and Porter. α is a smoothing constant determined by the developer, smaller α emphasize older observations and a larger α emphasizes more recent observations.

EXAMENSARBETE Reducing costs of manual regression testing using prioritisation and partitioning techniques
STUDENTER Erik Nord, Robin Rasmussen Vinterbladh
HANDLEDARE Per Runeson (LTH), Peter Jansson (Sony)
EXAMINATOR Emelie Engström (LTH)

Spara tid genom att prioritera urvalet av tester – jobba smartare, inte hårdare

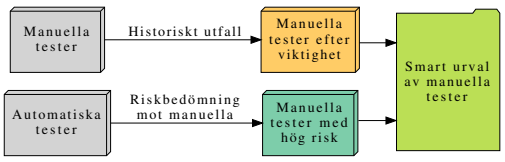
POPULÄRVETENSKAPLIG SAMMANFATTNING
Erik Nord, Robin Rasmussen Vinterbladh

En betydande del tid av mjukvaruutveckling ägnas åt kvalitetstestning. Ska man vara säker på att utfört arbete gick rätt till så kommer man aldrig undan denna tidsåtgången helt. Vi presenterar därför en smart urvalsprocess som föreslår de viktigaste kvalitetstesterna. Genom att endast testa detta smarta urvalet så sparar man tid.

Det finns ett vanligt problem inom mjukvarutestning som vi kallar *det stora regressionsproblemet*. Under tiden som mjukvara utvecklas sker många ändringar i programkoden. Varje introducerad ändring riskerar att orsaka nya fel i produkten. Felen behöver inte nödvändigtvis visa sig i den introducerade koden, utan det kan även dyka upp fel i gamla delar av produkten. Detta kallas för regressionsfel, alltså, ett fel som uppstår som konsekvens av en introducerad ändring. Man kan skydda sig mot regressionsfel genom att utföra regressionstester som går tillbaka och testar hela produkten, både de nyligen introducerade delarna och de gamla delarna. Man letar efter förändring i produktens beteende och det är de här förändringar som kallas regressionsfel. *Det stora regressionsproblemet* visar sig då varje kodändring som introduceras också måste introducera ett helt nytt regressionstest, vilket betyder att tidsåtgången för regressionstestning blir drastiskt större över tid.

De flesta tester är helautomatiska vilka snabbt kan utföras av en dator. Andra tester är manuella och måste utföras för hand av en utvecklare, vilket tar markant mycket mer tid än de helautomatiska testerna. Därför är det de manuella testerna som bidrar mest till det stora regressionsproblemet.

I vårt arbete har vi designat en urvalsprocess för manuella tester som väljer högrisktester och de viktigaste testerna. Först tittar vi på historiskt utfall för manuella tester och sorterar dem i en lista efter viktighet. Sedan tittar vi på helautomatiska testresultat och bedömer vilka snarlika manuella tester som har hög risk att upptäcka fel. Riskbedömningen blir som ett skyddsnät som fångar upp högrisktester, även om de av "misstag" aldrig klassades som viktiga i första steget.



Vi designade urvalslogiken efter att prioritera tester som haft tendensen att misslyckas. Resultatet visar att vår urvalsprocess är bättre än andra tidigare välkända urvalsprocesser från litteratur. Vi kom fram till resultatet genom att räkna antalet "misstag", alltså antalet gånger ett test *inte* valdes trots att testet skulle ha påvisat ett regressionsfel – ifall vi hade valt det.