Cost-Benefit Analysis of Using Dependency Knowledge at Integration Testing

Sahar Tahvili (✉)[1,2], Markus Bohlin (✉)[1], Mehrdad Saadatmand[1,2], Stig Larsson[1], Wasif Afzal[2] and Daniel Sundmark[2]

[1] SICS Swedish ICT, Västerås, Sweden
{sahart, markus.bohlin, mehrdad, stig.larsson}@sics.se
[2] Mälardalen University, Västerås, Sweden
{wasif.afzal, daniel.sundmark}@mdh.se

Abstract. In software system development, testing can take considerable time and resources, and there are numerous examples in the literature of how to improve the testing process. In particular, methods for selection and prioritization of test cases can play a critical role in efficient use of testing resources. This paper focuses on the problem of selection and ordering of integration-level test cases. Integration testing is performed to evaluate the correctness of several units in composition. Further, for reasons of both effectiveness and safety, many embedded systems are still tested manually. To this end, we propose a process, supported by an online decision support system, for ordering and selection of test cases based on the test result of previously executed test cases. To analyze the economic efficiency of such a system, a customized return on investment (ROI) metric tailored for system integration testing is introduced. Using data collected from the development process of a large-scale safety-critical embedded system, we perform Monte Carlo simulations to evaluate the expected ROI of three variants of the proposed new process. The results show that our proposed decision support system is beneficial in terms of ROI at system integration testing and thus qualifies as an important element in improving the integration testing process.

Keywords: Process improvement, Software testing, Decision support system, Integration testing, Test case selection, Prioritization, Optimization, Return on investment

1 Introduction

The software testing process is typically performed at various integration levels, such as unit, integration, system and acceptance level testing. At all levels, software testing suffers from time and budget limitations. Improving the testing process is thus essential from both product quality and economic perspectives. Towards this goal, application of more efficient testing techniques as well as automating different steps of the testing process (e.g., test case generation, test execution etc.) can be considered. For test execution, the decision of which test cases to select and the order in which they are executed can play an important

role in improving test efficiency. In our previous work [1], we introduced a technique based on dependencies between test cases and their execution results at runtime. The technique dynamically selects test cases to execute by avoiding redundant test cases. In our technique, identified dependencies among test cases give partial information on the verdict of a test case from the verdict of another one. In this paper, we present a cost-benefit analysis and a return on investment (ROI) evaluation of the dependency-based test selection proposed in [1]. The analysis is conducted by means of a case study of the integration testing process in a large organization developing embedded software for trains. In particular, we analyze various costs that are required to introduce our decision support system (DSS) and compare these costs to the achieved cost reductions enabled by its application. To improve the robustness of the analysis, stochastic simulation of tens of thousands possible outcomes have been performed. In summary, the paper makes the following contributions:

- A high-level cost estimation model, based on Monte-Carlo simulation, for the evaluation of integration test-case prioritization with test-case dependencies. The model is generic and can be used to analyze integration testing for a wide range of systems that exhibit test case dependencies.
- An application of the cost estimation model in an industrial case study at Bombardier Transportation (BT) where three alternatives for process improvement are compared to the baseline test execution order.
- A sensitivity analysis for the model parameter values in the case study. Through the analysis, various scenarios have been identified where the application of the proposed DSS can be deemed as either cost beneficial or not.

The remainder of this paper is structured as follows. Section 2 presents the background while Section 3 provides a description of the DSS for test case prioritization. Section 4 describes a generic economic model. Section 5 provides a case study of a safety-critical train control management subsystem, and gives a comparison with the currently used test case execution order. In Section 6, the results and limitations are discussed and finally Section 7 concludes the paper.

2 Background

Numerous techniques for test case selection and prioritization have been proposed in the last decade [2], [3], [4]. Most of the proposed techniques for ordering test cases are offline, meaning that the order is decided before execution while the current execution results do not play a part in prioritizing or selecting test cases to execute. Furthermore, only few of these techniques are multi-objective whereby a reasonable trade-off is reached among multiple, potentially competing, criteria. The number of test cases that are required for testing a system depends on several factors, including the size of the system under test and its complexity. Executing a large number of test cases can be expensive in terms of effort and wall-clock time. Moreover, selecting too few test cases for execution

might leave a large number of faults undiscovered. The mentioned limiting factors (allocated budget and time constraints) emphasize the importance of test case prioritization in order to identify test cases that enable earlier detection of faults while respecting such constraints. While this has been the target of test selection and prioritization research for a long time, it is surprising how only few approaches actually take into account the specifics of integration testing, such as dependency information between test cases.

Exploiting dependencies in test cases have recently received much attention (See e.g., [5, 6]) but not for test cases written in natural language, which is the only available format of test cases in our context. Furthermore, little research has been done in the context of embedded system development in real, industrial context, where integration of subsystems is one of the most difficult and fault-prone task. Lastly, managing the complexity of integration testing requires *online* decision support for test professionals as well as trading between multiple criteria; incorporating such aspects in a tool or a framework is lacking in current research.

The cost of quality is typically broken down into two components: conformance and nonconformance costs [7]. The conformance costs are prevention and appraisal costs. Prevention costs include money invested in activities such as training, requirements and code reviews. Appraisal costs include money spent on testing such as test planning, test case development and test case execution. The non-conformance costs include internal and external failures. The cost of internal failure include cost of test case failure and the cost of bug fixing. The cost of external failure include cost incurred when a customer finds a failure [8]. This division of cost of quality is also a basis for some well-known quality cost models such as Prevention-Appraisal-Failure (PAF) model [9] and Crosby's model [10]. While general in nature, such quality cost models have been used for finding cost of software quality too, see e.g., [11–13]. Software testing is one important determinant of software quality and smart software managers consider the cost incurred in test related activities (i.e., appraisal cost) as an investment in quality [8]. However, very few economic cost models of software testing exist, especially metrics for calculating the return on testing investment are not well-researched. It is also not clear how the existing software test process improvement approaches [14] cater for software testing economics. One reason for this lack of attention of economics in software quality in general is given by Wagner [15]. According to him, empirical knowledge in the area is hampered by difficulties in cost data gathering from companies since it is considered as sensitive. Nikolik [16] proposes a set of test case based economic metrics such as test case cost, test case value and return on testing investment. A test cost model to compare regression test strategies is presented by Leung and White [17]. They distinguish between two cost types: direct and indirect costs. Direct costs include time for all those activities that a tester performs. This includes system analysis cost, test selection cost, test execution cost and result analysis cost. Indirect costs include test tool development cost, test management cost and cost of storing test-related information. A test cost model inspired by PAF model is also presented by Black [18] while several cost factors for ROI calculation for automated

test tools are given in other studies [19–21]. Some other related work is done by Felderer et al. [22], [23] where they develop a generic decision support procedure for model-based testing in an industrial project and compare estimated costs and benefits throughout all phases of the test process.

3 Decision Support System for Test Case Prioritization

In this section we outline our proposed DSS, which prioritizes and selects integration test cases based on analysis of test case dependencies. Although not the focus of this paper, the DSS is also capable of performing multi-criteria decision analysis. The details of the approach can be found in [1]. In essence, the DSS provides an optimized order for execution of test cases by taking into account the execution result of a test case, its dependency relations and various test case properties. The steps performed in the DSS can be categorized into an offline and online phase: The offline phase produces an order for execution of test cases based on different test case properties (e.g., fault detection probability, execution time, cost, requirement coverage, etc.) while in the online phase, the pass or fail verdict of executed test cases is taken into account in order to identify and exclude upcoming test cases based on knowledge of dependencies between executed and scheduled test cases. The following definition of result dependency for integration test cases, first introduced in [1], constitute the basis of the dependency-based prioritization considered in this paper:

Definition. For two test cases A and B, B is *dependent* on A if, from the failure of A, it can be inferred that B will also fail.

In industrial practice, such dependencies may exist e.g., whenever a subsystem uses the result of another subsystem. During testing, the dependency may manifest whenever a test case B, dependent on test case A, is scheduled for execution before a component, tested by A, has been fully and adequately implemented and tested. By delaying the execution of B until A has passed, we ensure that the prerequisites for testing B are met. For instance, if the power system in a train fails to work, the lighting and air conditioning systems will not function either.

3.1 Architecture and Process of DSS

In this section, we give the basic architecture and process for the decision support system [1]. We use the term 'decision support system' to emphasize that it can be instantiated in contexts similar to ours, i.e., test cases written in natural language, meant for testing of integration of subsystems in embedded system development.

In Fig. 1, we describe the steps of the semi-automated decision support system for optimizing integration test selection. New test cases are continuously collected in a *test pool* (1) as they are developed. The test cases are initially not ordered and are candidates for prioritization. As a preparation for the prioritization of the test cases, the values for a selected set of *criteria need to be*

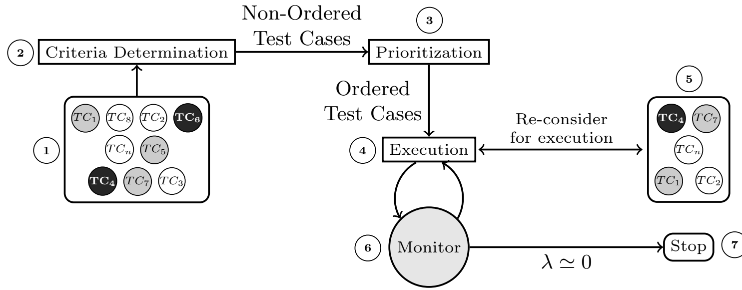


Fig. 1: Architecture of the proposed online DSS.

determined (2) for the new test cases. To *prioritize* among the test cases, the DSS expects the use of a multi-criteria decision making technique (3, prioritization)[24]. Once prioritized, the test cases are in this step *executed* (preferably) according to the recommended order (4). The result of executing each test case could either be *Pass* or *Fail*. We have previously in [1] shown that by detecting the dependency between test cases, we are able to avoid the redundant executions. When a test case fails during the execution, all its dependent test cases should be disabled for execution. The failed test cases from the previous step enter a queue for troubleshooting. The failed test cases will be *reconsidered for execution* once the reason of their failure is resolved (5). The results of each test are *monitored* (6) to enable (re-)evaluation of the *executability condition* (see [1]) of the test cases that are dependent on it. This will determine if the dependent test case should be selected for execution or not. Furthermore, the completeness of the testing process will be monitored through the metric *fault failure rate* (see [25]), denoted by λ in Fig. 1. This metric is the proportion of the failed test cases to the total number of executed test cases. The goal is to reach successful execution of maximum test cases to be able to finish the current test execution cycle. The current test execution cycle will *stop* (7) once the fault failure rate becomes 0. The steps in the DSS are performed in the integration phase for each iteration/release of the system. This will ensure that the relevant test are executed in a suitable sequence, and that the tests resources are used in an optimal way.

4 Economic Model

In this section we describe an economic model for the cost-benefit analysis of a software integration testing process where test cases are delayed until their execution requirements are fulfilled. The model is independent on the specific multi-criteria decision making technique used and of the specific system under test.

The purpose of the economic model is to adequately capture the costs and benefits which are directly related to the adoption of a DSS-supported testing process aiding in the prioritization of test cases. In industry, it is common that

time and cost estimates for software development processes only exist as estimates or averages, if at all. Finally, for an analysis to be useful in practice, the analysis model should be reasonably lightweight and contain only the absolutely necessary parameters. Using the proposed economic model, a stochastic ROI analysis can then be obtained by Monte Carlo simulation. A stochastic analysis avoids the sensitivity of traditional ROI analyses by considering a large number of possible parameter values, thereby offsetting some of the disadvantages in being forced to use less reliable parameter value estimates.

In this paper, we use a cost model with the following parameters:

1. A one-time fixed DSS implementation cost, corresponding to a fixed-price contract negotiated beforehand,
2. Variable costs for DSS training, on a per person-days of DSS usage basis,
3. Variable costs for DSS maintenance, on a per person-days of DSS usage basis,
4. Variable costs for (a) DSS data collection, (b) test planning, on a per test-case basis, and
5. Variable costs for (a) test-case execution (per executed test-case) and (b) troubleshooting (per failed test case).

We make the following simplifying assumptions on the testing process to be analyzed:

(a) A test case is executed at most once in each release cycle,
(b) If a test case fails, it is delayed until the next release cycle,
(c) Reliability of an already implemented and properly maintained system grows according to a simplified Goel-Okumoto model [26], and
(d) Test execution and troubleshooting effort is independent of each other and between test cases.

In the model, we only include the costs and benefits which are affected by using the DSS, and hence, do not need to consider other efforts such as the effort of testing for test cases that pass or that fail for other reasons than dependency. The following cost model is used for the approach in each release cycle t:

$$C_t = C_t^{\mathrm{I}} + d_t \cdot \left(C_t^{\mathrm{T}} + C_t^{\mathrm{M}}\right) + n_t \cdot \left(C_t^{\mathrm{D}} + C_t^{\mathrm{P}}\right) + \gamma_t \cdot \lambda_t \cdot n_t \cdot \left(C_t^{\mathrm{E}} + C_t^{\mathrm{B}}\right), \quad (1)$$

where C^{I} is the implementation cost, d_t is the number of person-days in t, C^{T} is the training cost, C^{M} is the maintenance cost, n_t is the number of test cases, C^{D} is the data collection cost, C^{P} is the test order planning cost, including possible preprocessing, test case data input, DSS runtime, post-processing and test case prioritization output, λ_t is the fraction of failed test cases in the release cycle, γ_t is the fraction of test cases that failed due to a fail-based-on-fail dependency (out of the failed test cases), C^{E} is the average test execution cost, and C^{B} is the average troubleshooting cost. The last term, $\gamma_t \cdot \lambda_t \cdot n_t \cdot \left(C_t^{\mathrm{E}} + C_t^{\mathrm{B}}\right)$, calculates the cost for unnecessarily running test cases which will surely fail due to dependencies. This is the only runtime cost we need to consider when comparing a DSS-supported process and the baseline process without DSS support, as the other costs for running test cases will remain the same.

Over the course of time, the maintenance cost for a deployed system with a small number of adaptations will be approximately proportional to the failure intensity of the system. In this paper, we therefore assume that the DSS software reliability grows according to a simplified Goel-Okumoto model (see [26]), i.e., that the failure intensity decrease exponentially as $\lambda(d) = \lambda_0 \mathrm{e}^{-\sigma d}$, where $\lambda(d)$ is the failure intensity at time d, λ_0 is the initial failure intensity, and σ is the rate of reduction. Further, for any release cycle t, each test case belonging to t is assumed to be tested exactly once within t. It is therefore logical to assume that there is no decrease in the test case failure rate during any single release cycle. Under these assumptions, the expected maintenance cost in release cycle t can be calculated as follows.

$$C_t^{\mathrm{M}} = C_0^{\mathrm{M}} \cdot D_t \cdot \mathrm{e}^{-\sigma D_t}, \tag{2}$$

where C_0^{M} is the initial maintenance cost and $D_t = \sum_{i=1}^{t} d_i$ is the total project duration at release cycle t, where d_i is the duration of a single cycle i.

Apart from the implementation cost C^{I}, there are other unrelated infrastructure costs which are not affected by the process change and can therefore be disregarded from. Likewise, full staff costs are not included, as the team size remains constant, and we instead focus on measuring the savings in work effort cost from a change in process. In the model in Eq. (1), the savings of a process change taking test case dependencies into account can be measured as a difference in costs, under the assumption that all other costs are equal. As each integration test case is normally executed once in each release cycle, after each cycle there is a set of failed test cases that needs to be retested in the next cycle. In this paper, we are interested in estimating the economic benefits of delaying test cases whose testability depend on the correctness of other parts of the system. In other words, we are interested in estimating the number of test cases which fail solely due to dependencies. For these test cases, we can save executing and troubleshooting efforts. If $\gamma_t \cdot \lambda_t$ number of the test cases fail due to dependencies, then, from Eq. (1), we have that by delaying the execution of such test cases, the saving (i.e. benefit) of changing the process can be at most:

$$B_t = \gamma_t \cdot \lambda_t \cdot n_t \left(C_t^{\mathrm{E}} + C_t^{\mathrm{B}} \right). \tag{3}$$

The estimate is an upper bound as in reality we may not be able to capture all dependencies in our analysis. Further, there is a possibility that the analysis falsely identifies dependencies which do not exist. The effect of delaying the corresponding test cases to the next phase is to delay the project slightly; however, this effect is likely small and we therefore disregard from it in this paper.

4.1 Return on Investment Analysis

A Return on Investment (ROI) analysis represents a widely used approach for measuring and evaluating the value of a new process and product technology [27]. In this study we consider all costs directly related to the process change to be part of the investment cost. If we also assume that the sets of test cases

to execute for all release cycles are disjoint, we can calculate the total costs and benefits by adding the costs and benefits for each release cycle. We use the following ROI model based on the net present value of future cash flows until time T and an interest rate r.

$$R_t = \frac{\sum_{t=0}^{T} B_t/(1+r)^t}{\sum_{t=0}^{T} C_t/(1+r)^t} - 1 \tag{4}$$

We assume that the implementation cost is paid upfront, so that $C_t^{\mathrm{I}} = 0$ when $t \geq 1$, and that there are no other benefits or costs at time $t = 0$. In other words, $B_0 = 0, C_0 = C_0^{\mathrm{I}}$ and, consequently, $R_0 = -1$. The interest rate r is used to discount future cash flows, and is typicallt the weighted average cost of capital for the firm, i.e., the minimum rate of return that investors expect to provide the needed capital.

5 Case Study

In order to analyze the economic feasibility of our approach, we carried out a case study at Bombardier Transportation (BT) in Sweden, inspired by the guidelines of Runeson and Höst [28] and specifically the way guidelines are followed in the paper by Engström et al. [4]. We investigated the software/hardware integration testing process for the train control management subsystem (TCMS) in the Trenitalia Frecciarossa 1000, a non-articulated high-speed trainset. The process aims to identify faults at the interface of software and hardware. The case study spanned six releases of 13 major and 46 minor function groups of the TCMS during a time period of 2.5 years, which involved in total 12 developers and testers for a total testing time of 4440 hours. The testing process is divided into different levels of integration, following a variation of the conventional V-model. The integration tests are performed manually in both a simulated environment and in a lab in the presence of different equipment such as motors, gearboxes and related electronics. The testing for each release have a specific focus, and therefore, there are only minor overlaps between the test cases in different releases. Each test case has a specification in free-text form, and contain information (managed using IBM Rational DOORS) on the (1) test result, (2) execution date and time, (3) tester ID, and (4) testing level.

The test result is one of the following: (a) Failed, (i.e., all steps in the test case failed), (b) Not Run, (i.e., the test case could be not executed), (c) Partly Pass, (i.e., some of the steps in the test case passed, but not all), and (d) Pass (i.e., all steps in the test case passed).

According to the test policy in effect, all failed test cases (including "Not Run" and "Partly Pass" test cases) should be retested in the next release. Furthermore, each of these initiates a troubleshooting process that incurs cost and effort. In the rest of this paper, we therefore use the term *failed* to mean any test verdict except "Pass". The objective of the case study is to analyze the improvement potential for the integration testing process at BT from decreasing

the number of unsuccessful test executions using knowledge of test-case dependencies. The chosen method is to estimate the economic effect on BT in the form of earned ROI using Monte-Carlo simulation. We answer the following research question in this case study:

RQ: What is the economic effect of introducing a DSS for reducing the number of unsuccessful integration test executions based on dependency knowledge?

The data collection for the case study was done through both expert judgment, inspection of documentation and a series of semi-structured interviews. The initial parameter value estimates for $C^{\mathrm{I}}, C^{\mathrm{T}}$ and C^{M} were made by the author team, as it was judged that the members of the team, having deployed several decision support systems in the past (see e.g. [29, 30]), possessed the necessary experience for this evaluation. Likewise, C^{P} was estimated by the research team through multiple meetings and re-evaluations. The documentation consists of the test case specification and verdict records in DOORS. In particular, the fault failure rate (λ) was calculated directly by counting the corresponding test case reports, and the fraction of dependent test cases (γ) was estimated through manual inspection of the comments in the same set of reports. Finally, a series of semi-structured interviews were conducted to both estimate the parameter values for the testing process itself, and to cross-validate the full set of parameter values already identified. The interview series were made with two testers (T1 & T2), a developer (D), a technical project leader (PL), a department manager (DM) and an independent researcher (R) in verification and validation. The composition and main purpose of the interviews are shown in Table 1. The final parameter values can be found later in this paper in Table 2 and Table 3.

Table 1: Series of interviews to establish parameter values

#	T1	T2	D	PL	DM	R	Main purpose
1	×						Estimate C^{D} from dependency questionnaire.
2, 3	×	×	×				Identify criteria for dependencies. Validate C^{D}.
4		×					Validate dependencies.
5, 6	×	×	×	×			Validate number of dependencies (γ). Estimate C^{E}.
7			×				Estimate C^{B}.
8				×	×		Validate $C^{\mathrm{E}}, C^{\mathrm{B}}, C^{\mathrm{I}}, C^{\mathrm{T}}, C^{\mathrm{M}}$ and C^{P}.
9				×	×	×	Validate C^{I}, C^{T} and C^{M}.

5.1 Test Case Execution Results

To estimate the number of result dependencies between test cases, we performed a preliminary analysis of the results for 4578 test cases. The analysis was based on an early discussion with testing experts, in which test result patterns that were likely to indicate a dependency were identified. The patterns have previously been independently cross-validated on a smaller set of 12 test cases by other test experts at BT (see [1]). We classified the test results using the same

patterns, resulting in 823 possible dependency failures out of 1734 failed test cases, resulting in a total estimate of $\gamma \approx 0.476$. In the semi-structured interviews, two testers independently estimated that approximately 45% of the failed test cases were caused by dependencies, which is close to our estimate. Table 2 shows the full results for the six release cycles.

Table 2: Quantitative numbers on various collected parameters per release. Note that the γ rate is reported as a fraction of the fault failure rate (λ).

	Release number						
Parameter	1	2	3	4	5	6	Total
Working Days (d)	62	89	168	65	127	44	555
Test cases (n)	321	1465	630	419	1458	285	4578
Fault failure rate (λ)	0.545	0.327	0.460	0.513	0.346	0.246	0.379
Fail based on fail. rate (γ)	0.411	0.267	0.393	0.753	0.630	0.457	0.475

5.2 DSS Alternatives under Study

We analyzed three different DSS variants, which all prioritize the test cases by aligning them with the identified dependencies but vary in the amount of automation they offer. The goal was to identify the tool-supported process change which is most cost-effective (as measured by the ROI metric) within a reasonable time horizon. The following DSS variants were considered:

- **Manual version:** prioritization and selection of test cases in the level of integration testing manually. In this version, a questionnaire on the time for test execution, troubleshooting and set of dependencies to other test cases, is sent to the testing experts. To be manageable for an individual, the questionnaire is partitioned into smaller parts according to the subsystem breakdown. To increase precision and decrease recall, it is preferable that several experts answers the same questionnaire; however, the exact number should be decided based on the experience level of the testers. One of the commercially and publicly available toolboxes for multi-criteria decision analysis (such as FAHP or TOPSIS) are then used for prioritization of test cases. Data is fed manually into and out of the DSS, and a spreadsheet is used to filter, prioritize and keep track of the runtime test case pool.
- **Prototype version:** Dependencies are collected as in the manual version. However, the DSS is custom-made to read the input in a suitable format, automatically prioritize, filter and keep track of the runtime test case pool, and can output a testing protocol outline, suitable for the manual integration testing process.
- **Automated version:** in addition to the prototype version, the DSS detects the dependencies automatically by utilizing a publically-available toolbox (such as Parser [31]). The criteria determination step (in Figure 1) would be applied on the test cases by utilizing some learning algorithms (for example

a counting algorithm for calculating the number of test steps in a test case for estimating the execution time for a test case).

As explained earlier in Section 4, we divide the total cost needed for software testing into fixed (one-time cost) and variable cost. The fixed cost includes the DSS cost for three versions which includes implementation, maintenance and training costs.

The variable cost contains execution cost and also troubleshooting cost for the failed test cases. The variable cost changes in proportion to the number of executed test cases and the number of failed test cases per project.

5.3 ROI Analysis Using Monte-Carlo Simulation

The three version of the DSS were evaluated on the six release cycles described before. As many other mathematical methods, ROI analyses are sensitive to small changes in the input parameter values. As an effect, the calculated ROI can fluctuate depending on the varying time estimates. For this reason we chose to both evaluate the ROI model above using Monte Carlo simulation, as detailed below, and to perform sensitivity analysis by varying the expected value of some of the time estimates, as detailed in the results section. The parameters for the three versions are shown in Table 3.

Table 3: DSS-specific model parameters and distributions

			Distribution param.		
Param.	Comment	Distr.	Manual	Semi	Auto
γ_t	Failed TC rate	Constant	*See Table 2.*		
λ_t	Failed dep. TC rate	Poisson	*See Table 2.*		
C^{E}	TC execution time, per TC	Rayleigh	2	2	2
C^{B}	TC troubleshooting time, per TC	Rayleigh	4	4	4
C^{I}	Total implementation time	Rayleigh	120	825	1650
C^{T}	Training time, per year	Rayleigh	540	360	360
C^{M}	Maintenance time, per year	Rayleigh	40	165	330
C^{D}	DSS data collection time, per TC	Rayleigh	69.2	69.2	0.00
C^{P}	DSS run time, per TC	Rayleigh	32.3	5.37	0.00

The focus on initial analysis means that estimation efforts should be kept low. For this reason, a single-parameter Rayleigh distribution, which is the basis in the Putnam model (see [32], [33]), was chosen for the distribution of effort for software testing and implementation tasks. Test-case failures were sampled from a Poisson distribution.

The three different DSS versions were simulated by sampling 100 000 values for each data point using the parameters in Table 2 and 3. The mean cumulative results are shown in Fig. 2 for the studied project at BT. In the experiments, utilizing all three versions of DSS resulted in a positive ROI at the end of the six release cycles. Moreover, the maximum value of ROI was found for the manual DSS version, where the maintenance and implementation costs are low as

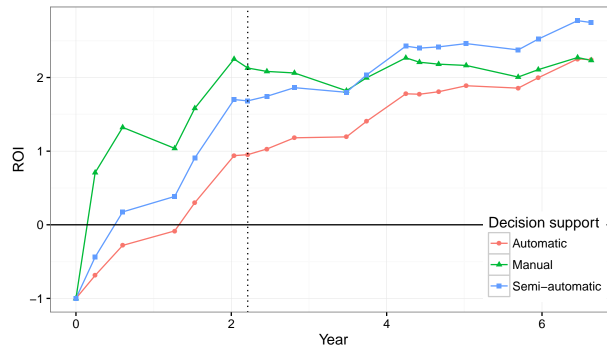


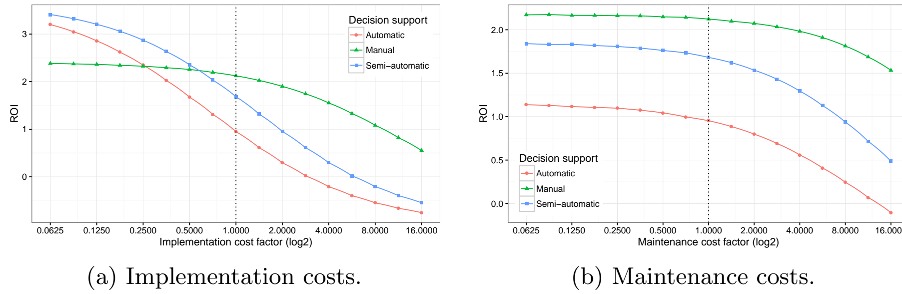
Fig. 2: Expected ROI for the three DSS versions. The vertical dotted line indicate the end of the six release cycles; later cycles are simulated using repeated data.

compared to the other versions. By evaluating in total three identical projects in sequence, thereby simulating one team of developers working on three larger projects over the course of almost seven years, it can be noted that the prototype tool is the most promising from an expected ROI perspective. The initial implementation effort and the continued maintenance costs of the automatic version makes it less promising even after seven years of deployment.

5.4 Sensitivity Analysis

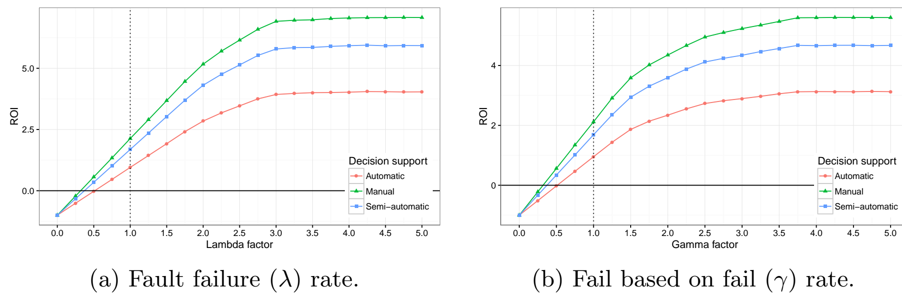
To increase the validity of the study, we also performed a sensitivity analysis by varying key parameters. In particular, the ROI analysis is sensitive to changes in the fixed and variable costs for the DSS. For this purpose, we varied the implementation costs, i.e., the fixed up-front cost for the DSS variant, and the maintenance costs, i.e., a critical part of the variable costs. The evaluation was performed using the same parameters as in the previous section, with the exception that the time horizon was fixed to the six development cycles in the case study project, with a duration of approximately 2.2 calendar years. The results of the experiments are shown in Figure 3. Each data point shows the mean result of 100 000 Monte-carol simulations, each consisting of six release cycles.

As can be seen, when increasing the implementation cost, the relationship between the three variants are the same. The manual variant is profitable even up to an implementation cost factor of 16. If the implementation factor is below 0.5, the prototype tool is the most profitable. It is worth noting that going to the left from the normative case, the ROI of the manual variant changes little, which is due to the fact that the implementation costs for the manual variant is small compared to the other variants. There is no change in the relationship between the variants when varying the maintenance costs by equal factors (Figure 3.b). Further, changing the cost factor of any single variant by less than 8 still maintains this relationship.



(a) Implementation costs. (b) Maintenance costs.

Fig. 3: Sensitivity analysis results for DSS costs.



(a) Fault failure (λ) rate. (b) Fail based on fail (γ) rate.

Fig. 4: Sensitivity analysis results when varying test case failure rates (λ and γ).

Figure 4 shows the results from varying λ, the fault failure rate, and γ, the fail-based-on-fail rate, using a coefficient between 0.0 and 5.0 individually for each release cycle. As can be seen, for both λ and γ the average ROI is above zero when the coefficient is above 0.5, corresponding to average rates of 0.189 and 0.237, respectively. Further, it can be observed that, for the 6 release cycles, the relationship between the three DSS variants are the same for all coefficients. The tailing-off effect which can be observed for higher coefficients occurs because for each individual release cycle, there is a natural upper limit of 1.0 for both rates, which is enforced after sampling.

6 Discussion and Threats to Validity

In this paper, we have analysed the cost-benefit of utilizing a dependency-based DSS in integration-level testing. It is clear that our results are based on simulation and it will be more convincing to do a similar investigation on real data from an actual project. This is one of our future goals with this direction of research. Our simulation results indicate that a DSS that considers dependencies in prioritizing test cases for integration level testing may be economically beneficial, but that the extent to which the method should be automated highly depends on the characteristics of the context in which the DSS should be introduced. It should be noted that prioritization based on test dependencies is

just one way to reduce effort of integration testing, and the DSS is not inherently limited to dependency-based prioritization. There are a number of other techniques to reduce effort such as system design planning and scheduling implementation. In fact, prioritization of the systems and sub-systems under test can reduce the degree of dependency between test cases. Another point to consider is earlier fault detection [34] which can be enabled by prioritization based on fault detection probability. It should further be noted that the costs in our model, as well as the calculated values of ROI are based on estimates. Consequently, the experimental uncertainty is inherent in the study design. We have tried to mitigate this uncertainty by dividing the costs into smaller units. For example, design cost and implementation cost for a test case are considered as two separated costs. By further defining the execution cost as a separate cost in our cost model, we account for the fact that a test case can be designed and created one time, but can be executed more than one time. We also separate the staff (testers and developers) cost between test case implementation and execution cost. In addition, the gathered data from BT is another source of uncertainty. As discussed in Section 5.4, by performing various sensitivity analyses we have identified and evaluated how different factors can impact the cost-benefit and ROI of the application of the DSS. In terms of validity threats, in the economic model, we assume that if test case A fails then running B (which depends on A) after A yields no additional information. This assumption could be invalid in other systems, where execution and failure result of test case B could still provide additional information, which in turn may affect the results of the analysis. Similarly, there are other assumptions related to our work (see Section 4), which can be invalid in the real world. Further, in our study we detect dependencies between test cases from test result reports to measure the percentage of fail-based-fail in the analyzed project. Considering the fact that the identification of dependencies by testers to some extent is subjective, other studies using other subjects may result in differences in dependency identification, and estimated gains from the DSS could thus be affected.

7 Conclusion and Future Work

In this paper, we introduced and assessed cost and benefits of applying a decision support system (DSS) for reducing the efforts at integration level testing. We identified the cost factors relevant in integration testing and provided a cost estimation model for calculation of return on investment based on which we evaluated where the use of our proposed DSS will be economically beneficial and result in cost reductions. The proposed decision support system has been applied to an industrial case study of a safety-critical train control subsystem and a brief analysis of the result was given. The results of the BT case study indicate that utilizing the proposed DSS can reduce test execution efforts and achieve a positive value for ROI for a system of that size and complexity, where higher test execution efficiency was enabled by identifying and avoiding test redundancies based on their dependencies. Moreover, by applying the proposed DSS, we can detect the hidden faults in the system under test earlier and fault

failure rate increases with time, as we have demonstrated in [24]. The ROI of the proposed DSS increases in situations where the number of test cases is large, the system under test is complex consisting of dependent modules and subsystems, and there exist additional limitations, such as budget, deadline, etc. The level of ROI depend on various cost factors, such as the cost for implementing and maintaining the tool, personnel training, execution time of the decision making algorithms, total number, size and execution time of test cases. This leads also to the conclusion that for a system which is small in the sense that the number of test cases is very small and there are no extra limitations for performing testing activities, a high ROI value may not be achieved by using the DSS.

Acknowledgements

This work was supported by VINNOVA grant 2014-03397 through the IMPRINT project and the Swedish Knowledge Foundation (KKS) grant 20130085 through the TOCSYC project and the ITS-EASY industrial research school. Special thanks to Johan Zetterqvist, Ola Sellin and Mahdi Sarabi at Bombardier Transportation, Västerås-Sweden.

References

1. Tahvili. S, Saadatmand. M, Larsson. S, Afzal. W, Bohlin. M, and Sudmark. D. Dynamic integration test selection based on test case dependencies. In *The 11th Work. on Testing: Academia-Industry Collaboration, Practice and Research Techniques (TAIC PART)*. 2016.
2. Yoo. S and Harman. M. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120.
3. Catal. C and Mishra. D. Test case prioritization: A systematic mapping study. *Soft. Qual. Journal*, 2013.
4. Emelie Engström, Per Runeson, and Andreas Ljung. Improving regression testing transparency and efficiency with history-based prioritization–an industrial case study. pages 367–376, 2011.
5. Bell. J. Detecting, isolating, and enforcing dependencies among and within test cases. In *22nd Inte. Symp. on Foundations of Software Engineering*, 2014.
6. Zhang. S, Jalali. D, Wuttke. J, Mucslu. K, Lam. W, Ernst. M, and Notkin. D. Empirically revisiting the test independence assumption. In *Int. Symp. on Software Testing and Analysis*, 2014.
7. Campanella. J. *Principles of quality costs: Principles, implementation and use.* ASQ Quality Press, 1999.
8. Black. R. What it managers should know about testing: How to analyze the return on the testing investment, 2004.
9. British Standards Institution. *Guide to the economics of quality. Proc. cost model.* B.S. (Series). BSI, 1992.
10. Crosby. P. *Quality is free: The art of making quality certain.* Penguin, 1980.
11. Slaughter. S, Harter. D, and Krishnan. M. Evaluating the cost of software quality. *Communications of the ACM*, 41(8):67–73, 1998.

12. Krasner. H. Using the cost of quality approach for software. *Crosstalk: The Jour. of Defense Software Engineering*, 11:6–11, 1998.
13. Boehm. B, Huang. L, Jain. A, and Madachy. R. The ROI of software dependability: The iDAVE model. *IEEE Software*, 21(3):54–61, 2004.
14. Afzal. W, Alone. S, Glocksien. K, and Torkar. R. Software test process improvement approaches: A systematic literature review and an industrial case study. *Jour. of Systems and Software*, 111:1 – 33, 2016.
15. Wagner. S. *Software product quality control*. Springer, 2013.
16. Nikolik. B. Software quality assurance economics. *Info. and Software Technology*.
17. Leung. H and White. L. A cost model to compare regression test strategies. In *Proc. of the 1991 Conf. on Software Maintenance*, 1991.
18. Black. R. *Managing the testing process: Practical tools and techniques for managing hardware and software testing*. Wiley Publishing, 3rd edition, 2009.
19. Münch. S, Brandstetter. P, Clevermann. K, Kieckhoefel. O, and Reiner Schäfer. E. The return on investment of test automation. *Pharmaceutical Engineering*.
20. Hayduk. B. Maximizing ROI and avoiding the pitfalls of test automation, 2009. Real-Time Technology Solutions, Inc.
21. Hoffman. D. Cost benefits analysis of test automation, 1999. Software Quality Methods, LLC.
22. Mohacsi. S, Felderer. M, and Beer. A. Estimating the cost and benefit of model-based testing: A decision support procedure for the application of model-based testing in industry. In *Proc. of the 2015 41st Euromicro Conf. on Software Engineering and Advanced Applications*, SEAA '15.
23. Felderer. M and Beer. A. Estimating the return on investment of defect taxonomy supported system testing in industrial projects. In *Proc. of the 2012 38th Euromicro Conf. on Software Engineering and Advanced Applications*, SEAA '12.
24. Tahvili. S, Afzal. W, Saadatmand. M, Bohlin. M, Sundmark.D, and Larsson. S. Towards earlier fault detection by value-driven prioritization of test cases using ftopsis. In *Proc. of the 13th Int. Conf. on Inform. Tech.: New Generations*, 2016.
25. Debroy. V and Wong. W. On the estimation of adequate test set size using fault failure rates. *The Jour. of Systems and Software*, page 587–602, 2011.
26. Musa. J and Okumoto. K. A logarithmic poisson execution time model for software reliability measurement. In *Proc. of the 7th Int. Conf. on Software engineering*.
27. Rico. D. *ROI of Software Process Improvement*. J Ross Publishing, 2004.
28. Runeson. P, Höst. M, Rainer. A, and Regnell. R. *Case Study Research in Software Engineering*. WILEY, 2012.
29. Bohlin. M and Wärja. M. Maintenance optimization with duration-dependent costs. *Annals of Operations Research*, 224(1):1–23, 2015.
30. Bohlin. M, Holst. A, Ekman. J, Sellin. O, Lindström. B, and Larsen. S. Statistical anomaly detection for train fleets. In *Proc. of the 21st Innovative Applications of Artificial Intelligence Conf.*, 2012.
31. Marneffe. M and Manning. C. Stanford typed dependencies manual. *Tech. report, Stanford University*, 2008.
32. Putnam. L. A general empirical solution to the macro software sizing and estimating problem. *IEEE transactions on Software Engineering*, 4(4):345, 1978.
33. Putnam. L. A macro estimating methodology for software development. 1976.
34. Hunt. B, Abolfotouh.T, Carpenter. J, and Gioia.R. Software test costs and roi issues. University Lecture, 2014.