



# Software test results exploration and visualization with continuous integration and nightly testing

Per Erik Strandberg<sup>1,2</sup> · Wasif Afzal<sup>2</sup> · Daniel Sundmark<sup>2</sup>

Accepted: 13 January 2022  
© The Author(s) 2022

## Abstract

Software testing is key for quality assurance of embedded systems. However, with increased development pace, the amount of test results data risks growing to a level where exploration and visualization of the results are unmanageable. This paper covers a tool, Tim, implemented at a company developing embedded systems, where software development occurs in parallel branches and nightly testing is partitioned over software branches, test systems and test cases. Tim aims to replace a previous solution with problems of scalability, requirements and technological flora. Tim was implemented with a reference group over several months. For validation, data were collected both from reference group meetings and logs from the usage of the tool. Data were analyzed quantitatively and qualitatively. The main contributions from the study include the implementation of eight views for test results exploration and visualization, the identification of four solutions patterns for these views (filtering, aggregation, previews and comparisons), as well as six challenges frequently discussed at reference group meetings (expectations, anomalies, navigation, integrations, hardware details and plots). Results are put in perspective with related work and future work is proposed, e.g., enhanced anomaly detection and integrations with more systems such as risk management, source code and requirements repositories.

**Keywords** Software testing · Test results · Visual analytics

## 1 Introduction

Modern embedded systems tend to come with increased complexity, e.g., in the automotive industry the number of computers per car has increased from 0, to a handful in the 1970s, to hundreds in the 2010s [45]. Also, the companies developing embedded systems tend to strive more and more for agile and continuous practices in the software development, which could involve compilation and testing of each code change in isolation, nightly testing, etc. With continuous practices, there is an increased amount of information generated from the software process as the development and

test cycles become more frequent [44]. Making sense of test results data can be challenging, and if data are not presented well, time and attention may be wasted while making critical decisions [1].

Previous work seems to have: (i) been conducted for unit testing or a less complex testing process, e.g., Wang et al. [55] that visualize test artifacts on the state machine model of a system; (ii) aimed at distilling the potentially rich and diverse test data into a quality indicator such as Ram et al. [38] that explore several metrics and focus on these as opposed to focusing on the test data itself; or (iii) striven for creating a dashboard, mash-up or 3D-visualization on top of data [4,5,9,16]. The *goal* of the study at hand is to investigate a test results exploration and visualization (TREV) tool to support the development of embedded software in parallel code branches with continuous integration and nightly testing. We implement and evaluate a tool, Tim,<sup>1</sup> for TREV, to allow developers, testers, project managers and others,

✉ Per Erik Strandberg  
per.strandberg@westermo.com; per.erik.strandberg@mdh.se

Wasif Afzal  
wasif.afzal@mdh.se

Daniel Sundmark  
daniel.sundmark@mdh.se

<sup>1</sup> Westermo Network Technologies AB, Västerås, Sweden

<sup>2</sup> Mälardalen University, Västerås, Sweden

<sup>1</sup> Tim is named after Tim the Enchanter, portrayed by the British actor John Cleese, in *Monty Python and the Holy Grail*.

to make sense of the massive but sparse and heterogeneous data that come from nightly testing at Westermo Network Technologies AB (Westermo), a company that develops networked embedded systems. Tim replaces an existing TREV system [48] with three *problems*:

1. *Requirements* The old system was developed ad hoc, and it is not clear if the implementation matches what users need.
2. *Technological Flora* The building blocks of the old system were not developed in accordance with current company values such as modularity, testability and maintainability.
3. *Scalability* With the increased test intensity and parallelism, graphical elements no longer fit on the screen, and some database queries have grown slow, making the system appear unresponsive.

This paper builds on our previous work [48] where the old TREV system was presented and several areas of future enhancements were elicited to solve the above mentioned problems. Based on those elicited requirements, we present a new and complete tool (Tim), its underlying architecture and database layout, as well as its continuous implementation and evaluation with an industrial reference group. With respect to the problems of the old system, we describe how Tim was implemented in a development process involving an industrial reference group of users that assisted in prioritizing tasks and evaluating the implementation; that technological choices were made based on preferences of the test framework team at Westermo, and finally, that the performance of the database queries was monitored; the visualizations now fit on screens and better allow filtering on the views of Tim.

In addition to the solution of problems with the existing TREV system, the *main contributions* of this paper are as follows: First, the identification of four solution patterns: filtering, aggregation, previews and comparisons (in Sect. 4.2). Second, eight views for TREV, as well as an empirical evaluation of their use (in Sects. 4.3 and 4.4). Third, the identification of six challenges for TREV: expectations, anomalies, navigation, integrations, hardware details and plots (in Sect. 4.5).

The rest of this paper is organized as follows: In Sect. 2, we present some background to the study such as the industrial context. The research process is described in Sect. 3. The functionality and evaluation of the Tim tool are described in Sect. 4. In Sects. 5 and 6, we discuss and conclude the paper.

## 2 Industrial motivation and context

This section motivates the study and introduces the industrial context and some factors that lead to TREV being an impor-

tant but sometimes time-consuming and non-trivial activity for industry practitioners.

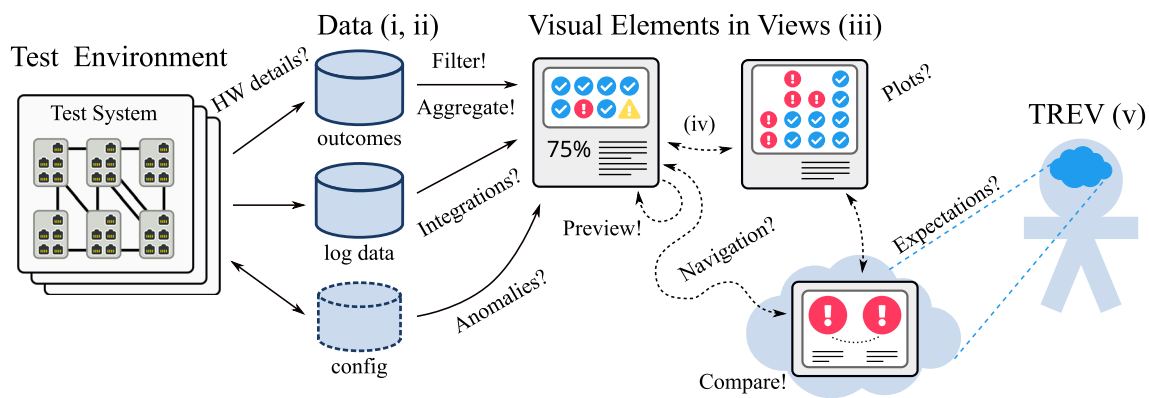
### 2.1 Visualization

Visualization can be seen as a process of “representing data, information, and knowledge in a visual form to support the tasks of exploration ... and understanding” [56]. This is a mental process that occurs when interacting with *visual elements*. The visual elements are, in turn, representations of data that have been collected, stored, fetched, filtered, transported and transformed to and from some data storage. Figure 1 illustrates an overview of Tim’s visualizations based on simplification of previous work [28,31,43,53,56]: (i) Data are collected from the test systems in the test environment: verdicts, log files, etc. (ii) The configuration of how test resources are partitioned is a valuable reference for the user. (iii) Visual elements such as icons, plots and tables are generated either beforehand or, e.g., inside a web browser with some JavaScript library. (iv) The user will navigate between the views, and (v) the TREV process occurs, while the user interacts with the visual elements. In the figure, we indicated challenges with question marks (e.g., Anomalies?) and solution patterns with exclamation marks (e.g., Filter!)

In a software testing context, the visualizations support hypothesis generation, decision-making, and can map to information needs [1]; e.g.: Is this the same bug as I saw yesterday? Has the bug already been fixed? Is this bug only present on some types of devices? What version of the test case did we run? Why was my new test case not selected for testing tonight? Is the level of quality sufficient for this feature to be considered completed? Can we release the software?, etc. We use the term *test results exploration and visualization* (TREV) for when users interact with a system that generates visual elements to support answering these types of questions.

### 2.2 Context

This study was conducted as an industry-academia coproduction involving the Software Testing Laboratory research group at Mälardalen University and Westermo Network Technologies AB (Westermo), both in Västerås, Sweden. Westermo develops embedded systems for communication networks in industrial applications. A typical product is a router with hardware suitable for a harsh industry environment, and software that enables standard communication protocols providing communication for other systems. Typical customer domains are onboard rail, track-side rail, power distribution and industrial automation. Westermo has more than 300 employees, does R&D in several European countries, has offices globally and targets an international market. The test framework team in Västerås develops the Fawity



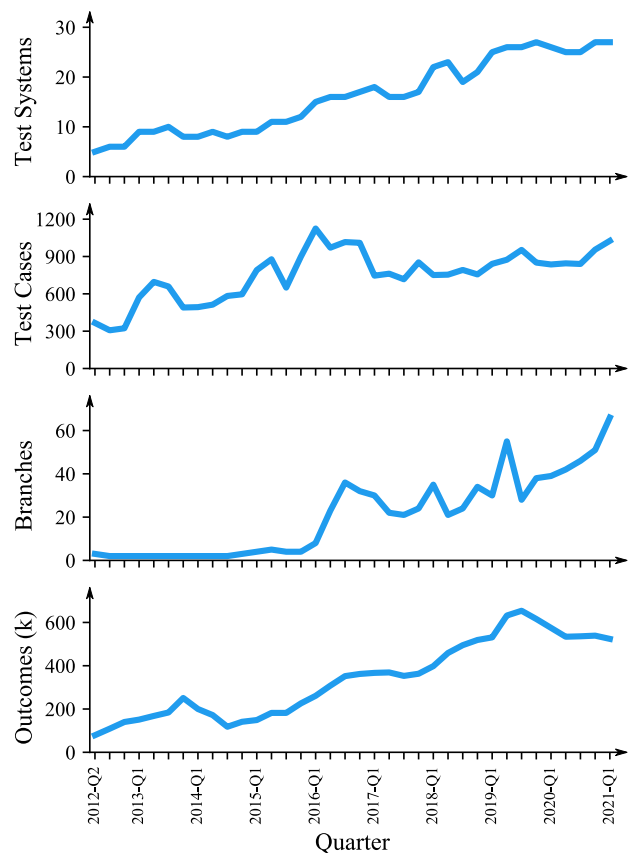
**Fig. 1** Overview of software test results exploration and visualization. Challenges are indicated with question marks (e.g., Anomalies?), and solution patterns with exclamation marks (e.g., Filter!). Steps (i) to (v) are described in Sect. 2.1

test framework<sup>2</sup> for automated testing of the software components in the Westermo Operating System (WeOS), an embedded OS based on GNU/Linux running in many products and developed by several software teams. This team also develops, builds and maintains physical and virtual test systems to allow testing in realistic conditions. A typical physical test system has between 6 and 20 WeOS devices connected in a network topology, controlled by a PC running the test framework which communicates with the devices over a serial console connection. In addition to the devices under test, many test systems also have devices used to simulate loss of connection, traffic generators, as well as I/O for powering units on and off.

Software development at Westermo is conducted in an agile process based on Kanban, where isolation of feature development in separate feature branches is central. By branching and merging the WeOS code base, the development of one feature is isolated from the development of another feature. The different WeOS branches are also tested separately and resources for nightly testing are distributed between branches. Once a feature branch is implemented, tested, passed risk analysis and several other quality assurance activities, the branch is merged into a main branch. After a number of such merges, a release branch is branched off from the main branch, and in this release branch, a software release is created.

### 2.3 Complicating factors

In the past decade, the testing at Westermo has scaled up significantly, see Fig. 2. Since the introduction of the test results database, the number of test systems in use in nightly testing has increased from 5 to 27, the number of parallel code branches changed drastically in 2016 when the approach to



**Fig. 2** Growth in complexity over time: number of test systems, test cases (with parameter combinations) and code branches used, as well as produced verdicts (in thousands) broken down per quarter since the introduction of the test results database

test feature branches was introduced and went from about two to more than 65 in Q1 2021. The number of test cases (distinct combinations of test script and parameter settings) increased from about 350 to about 1025, and the number of started test cases increased from about 80k to about 525k per quarter.

<sup>2</sup> Fawly is named after Basil Fawly, portrayed by the British actor John Cleese, in *Fawlty Towers*.

At Westermo, the complexity of testing involves, not only the creation and execution of test cases, but also regression test selection [47,51], where test cases are prioritized in each test suite for nightly testing. This means that certain risk factors, such as previous history of failures and code changes, play a role in the selection. In addition, each test system has a number of devices (six in the test system illustrated in Fig. 1) that each test case may or may not use. A firewall test might require three devices—the first will act as the device to be protected behind the firewall, the second will try to reach the inner device through the firewall, and the third would fill the role of the firewall. Now, if a test case requires three devices and a test system has six, then there are many ways of selecting these three [50]. If the devices in the test system are somehow different, then the actual hardware selection may be very relevant to a user of a TREV system. Furthermore, if a test case cannot be mapped onto a test system, perhaps because the test case requires hardware resources or a network topology that is not supported by the test system, then that test case cannot start and thus cannot go pass or fail, instead this result is called *unmappable*. In addition, results from test cases that do not run to completion (because of an unexpected exception, etc.) are called *invalid*, test cases that cannot be found when they are to be loaded before the suite are *unloadable*, and test cases that are part of a suite but not executed due to constraints in time produce the verdict *skipped*.

Finally, a challenge described in electrical systems in the 1940s, in the aerospace domain in the 1960s and for agile test automation in blogs four decades later is intermittent faults and intermittently failing tests [2,8,15]. In the software engineering domain, the term flaky tests is often used and this topic is heavily researched for unit-level testing on open source systems, and more recently also on system-level testing [49]. Typical root causes of the intermittence may be shortcomings in the software under test, the hardware of the embedded systems or in the test environment including the test cases themselves.

If a user of a TREV tool investigates test cases from one night at a time, and nightly testing only contained a portion of available tests, then one might not realize if tests are flaky or not—one needs to aggregate test results data over time in order to determine the level of intermittence. Furthermore, if a user only investigates failing tests and a certain test case is not shown, then there could be several reasons for this: perhaps the test case passed, perhaps it was not selected for nightly testing, or perhaps it could not be executed on a certain test system (it was unmappable), etc.

### 3 Research process

As previously mentioned, the *goal* of the study is to investigate a test results exploration and visualization (TREV) tool, in order to support the development of embedded software. This work was conducted as a case study [40] that also drew inspiration from action research [36] and Munzner's nested model for visual design [28]. The case in the case study was the new TREV tool Tim, as it was being defined, implemented and adopted in the context at Westermo. The study can be seen as a holistic single-case study since the case is also the unit of analysis. We focus on two data sources: (i) a reference group that represents users of the system and give input on what the problems are and what decisions the system should support and (ii) an open<sup>3</sup> alpha version of the system and logs from its usage.

This study is to some extent also *action research* because it covers “how an intervention, like the introduction of a method or tool, affects a real-life context” [36]. This is relevant because we want the intervention (implementation of Tim) to have an effect (users improve their capacity to explore test results). Action research is also applicable when “the change and its observation are an integral part of addressing the research question and contribute to research.”

#### 3.1 Data collection

*Reference group* In order to provide the most utility to the company, and also to validate the implementation, we recruited a reference group of twelve individuals. We aimed at diversity and the group contained line managers, project managers and software developers both in the test framework and WeOS teams. The differences in roles could be considered a sort of triangulation, thereby addressing some threats to validity. The test framework team manager, the product owner of Tim, was also part of the group. Reference group meetings were led by the first author of this paper (who is employed at Westermo as well as a member of the test framework team). At every meeting, there was at least one WeOS developer, test framework developer and manager represented, with the exception of one meeting that specifically targeted project and line managers. However, many meetings had a majority of WeOS developers, which might have biased the work in their favor. In discussions with the reference group, we were particularly interested in three critical decision points in the development process: in daily work, at merge time (when a feature is completed and its code branch is to be merged into the main code branch) and at release time (when a new software version is about to be released). The reference group gave feedback on mockups (implementation

<sup>3</sup> Open in the sense that every employee could use it.

proposals), evaluated implementation and helped prioritize continued work.

The first reference group meeting was held in November 2020, and the seventh in April 2021. Each meeting had about seven participants and lasted for about one hour. The meetings were held in a video chat tool due to the pandemic, and they were recorded and transcribed into a total of 36 A4 pages of text which was later analyzed with qualitative coding [6]. Toward the end of the study, member checking was conducted during one meeting. In each meeting, we tried to capture not only the desires of the reference group, but also the motives or problems behind their preferences. In parallel with the meetings, the implementation was ongoing (illustrated in Fig. 3).

In addition to the reference group meetings, several education efforts were conducted: one session on how to use the tool, a second targeting familiarization with the implementation of the frontend and a third session on all of Tim's code including an introduction to the continuous integration process set up with the dev.ops. team. All three of these meetings were also recorded as a sort of "user guide" and "developer guide."

*Web server logs* Initially, the plan was to develop Tim off-line and only open it more broadly to Westermo staff after some time. However, early on we were asked to make it available despite not having much functionality in place (only early versions of the first three views). Therefore, we made an "open alpha" available that ran in parallel with the system to be replaced. In addition to providing utility to Westermo staff, we also enabled logging in the open alpha which gave us clues on slow database questions, navigational profile, etc. The open alpha also helped the reference group champion the adoption of Tim among other colleagues, and opened up for ad hoc feature requests (e.g., during meetings that were not related to Tim).

From the alpha, we stored the logs from the nginx web server. In addition to what nginx writes to log by default (e.g., each request for data), we also stored the request time, which is the duration needed by the backend to generate contents for the frontend. In the case where non-trivial database queries are needed to generate contents, the request time can be expected to be noticeable, which is a risk identified by Munzner [28].

### 3.2 Ethical considerations

According to our previous work in research ethics when involving both an academic and an industrial partner [46], ethical principles include: consent, beneficence, confidentiality, researcher skill, etc. The main ethical threat we identified for this study was for researchers to expose sensitive data or opinions from the company or the participants. This was mitigated by anonymizing transcripts from the reference group

meetings while transcribing and never letting raw data out of company computers. Furthermore, we anonymized all mock-ups, as well as the screenshot of Tim, and have requested and received permission to conduct and publish this research. To avoid harm, such as making incorrect decisions based on Tim, we also suggested that Westermo staff use the old system in parallel, at least when making important decisions or when results in Tim seem suspicious.

## 4 The Tim tool

This section covers technical details of Tim (in Sect. 4.1), as well as the main findings of this study: recurring patterns in the views we implemented (Sect. 4.2), the views and how they were evaluated (Sects. 4.3 and 4.4), as well as challenges with TREV that were frequently discussed at reference group meetings (Sect. 4.5).

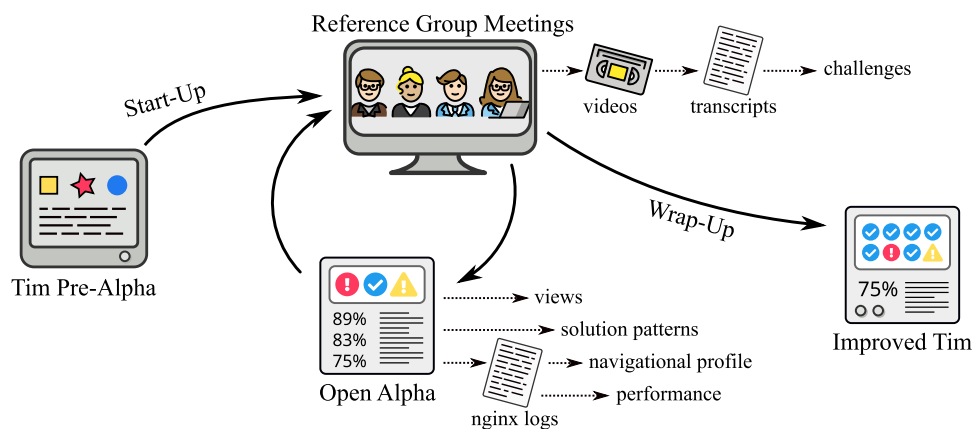
A screenshot of one of the actual views in Tim is shown in Fig. 4. However, for reasons of readability and confidentiality, mockups are used in the remainder of the paper. (The mockup showing the same view is illustrated in Fig. 8.)

### 4.1 Technical details

Before work with the reference group started, the old TREV system had been up and running since 2012. Based on the old system and interviews conducted at the company, we had a set of user stories describing desirable functions for Tim. Parts of the old system and the user stories are described in our previous research [48]. In addition, we had a skeleton-implementation and definition of technical constraints for the continued implementation of Tim. This had been a significant undertaking and involved refactoring of the database schema, introduction of containerization, as well as technological trials with tools and frameworks (programming languages, web frameworks, etc.). In addition, a prototype in Python 3 and Django had been implemented in 2019, which was rejected based on technologies used. Much of the motivation for decisions on technical flora was based on business decisions (such as focus on scalability and maintainability), tool preferences in the test framework team and partially on a master's thesis on potential advantages of statically typed languages [41].

In 2012, when the testing at Westermo was done on two parallel code branches on four physical test systems, it was observed that exploring test results in PDF-files was no longer scalable (each test session generated a separate PDF, so the number of documents to read increased linearly with both number of test systems and code branches). At that time, the first author of this paper suggested and later implemented a test results database, and an overview of the database schema is illustrated in Fig. 5. The main motivation for implementing the database was to make it easier to store information

**Fig. 3** Overview of the research process. Tim was updated iteratively with repeated reference group meetings. Based on data from transcribed meetings and logs from the open alpha, we identified solution patterns (Sect. 4.2), views (Sect. 4.3), challenges (Sect. 4.5), as well as information on the performance (Sect. 4.4.4) and navigational profile (Sect. 4.4.5) of Tim



related to the state of the test systems and also to aggregate results, i.e., test verdicts. During the implementation of Tim, the database schema was slightly simplified and modified to better fit the naming conventions used in the database communication library (SQLBoiler). An example of a raw sql-query to the database in order to generate the output shown in Table 1 and Fig. 8 is:

```
select
  system, branch, result,
  path as test, outcomes.id as oid
from sessions
  join outcomes on session_id = sessions.id
  join tests on test_id = tests.id
where
  sessiontype = 'nightly' and
  branch = 'brA' and
  result in (1, 2) and
  sessions.started > '2021-04-03 14:30:00' and
  sessions.started <= '2021-04-04 14:30:00'
order by system, branch, test;
```

In order to support development of Tim and the testing of database queries, we implemented functionality to populate the database with random data, and also to sync data from the legacy database as an alternative. Generating random data makes Tim independent of the environment in which nightly testing is conducted, thereby allowing rapid development and test cycles of the Tim tool itself. Using real data in the development of Tim enables manual high-level acceptance testing.

The overall architecture of Tim is illustrated in Fig. 6. Significant technologies included are: (i) containerization with docker, docker-compose and kubernetes; (ii) a test results database in MariaDB (a fork of MySQL); (iii) a backend in the Go language using SQLBoiler for object relational mapping (ORM); (iv) a web-based API that exports or imports test results data in json format (part of the backend); (v) a command-line interface for showing test results—at first implemented before the frontend was in place, but found to be very useful, fast and lightweight; (vi) a web frontend implemented in JavaScript with Vue/Vuex; (vii) a server storing log files from historic test executions; and (viii) the current

configuration of how to distribute test resources in nightly testing. In the future, more integrations are planned (e.g., with the source code repositories to more easily navigate to documentation showing the purpose of a certain test case, etc.).

Important technical differences between Tim's code base and that of the system to be replaced include: (i) Containerization: e.g., the database resides in its own docker container, making it possible for different projects to have different instances of a test results database, and to experiment with changes in database schemas, etc. (ii) Testability: the Go libraries in the backend can be tested with dummy test data that can be generated and re-generated rapidly—a complete cycle that creates a database, populates it with data, and runs all unit tests and database integration tests, requires less than 20 seconds. (iii) Continuous integration: each code change in any of Tim's components triggers linting of Go code with golangci-lint (a lint aggregator configured to use 33 linters) and Vue code with prettier. The purpose of linting on each code change is to capture potential shortcomings in the code (e.g., unused variables, "magic numbers," etc.), and to enforce a common coding style (which is not trivial for a language such as JavaScript [52]). Running the tests on each code change is desirable and will likely be added in the future. (iv) Technology choices such as programming language, JavaScript framework, web style, etc. have been based on team consensus to build a feeling of team ownership.

## 4.2 Solution patterns

During the implementation of Tim, four concepts were repeated in different ways in the views. Despite the risk of researcher bias and biases coming from what is possible or easy in the technological platform (e.g., using Go or Vue), these patterns may be valuable for other researchers or practitioners since they augmented TREV in Tim. Furthermore, some of the patterns were discussed with the reference group,

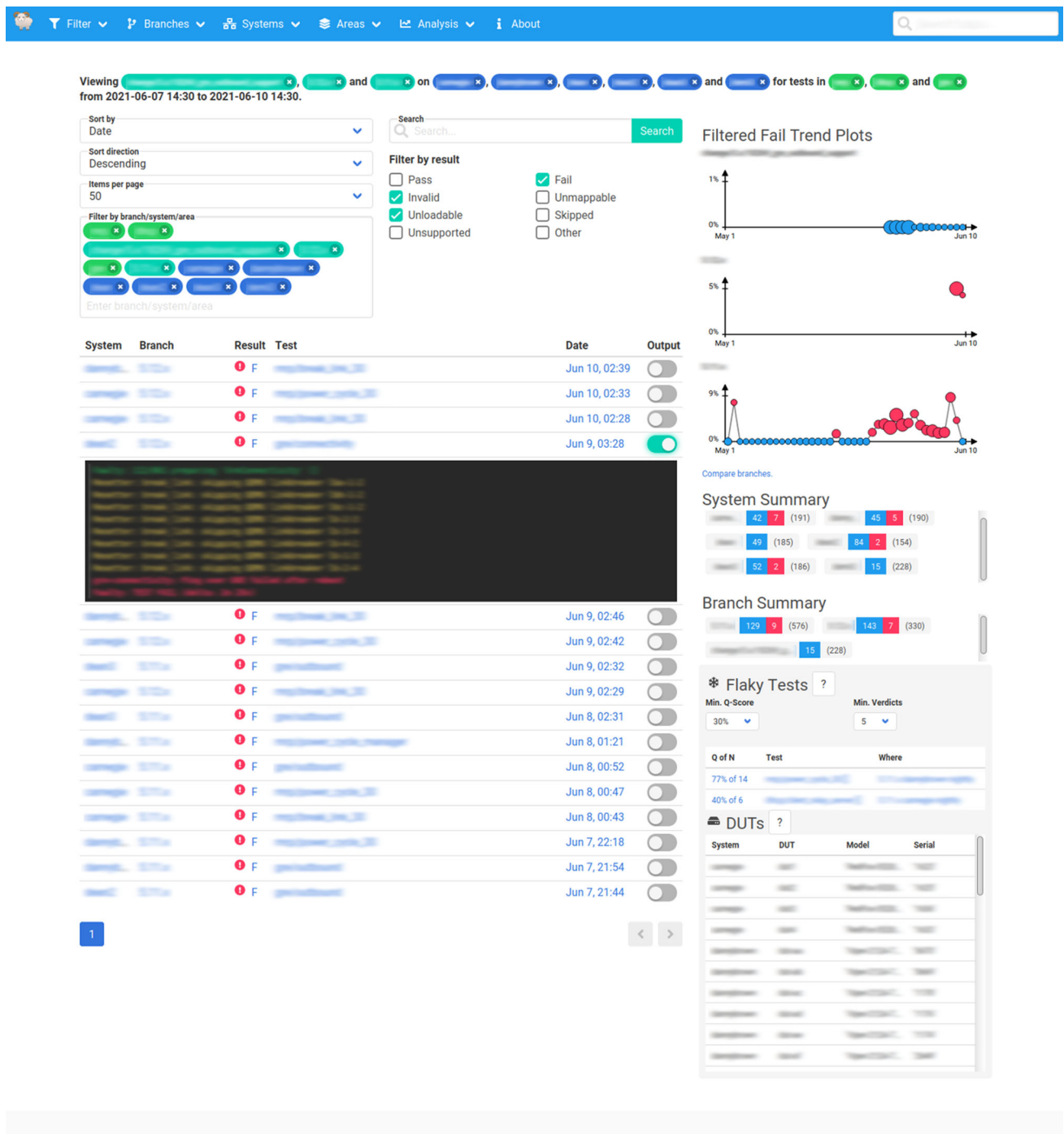
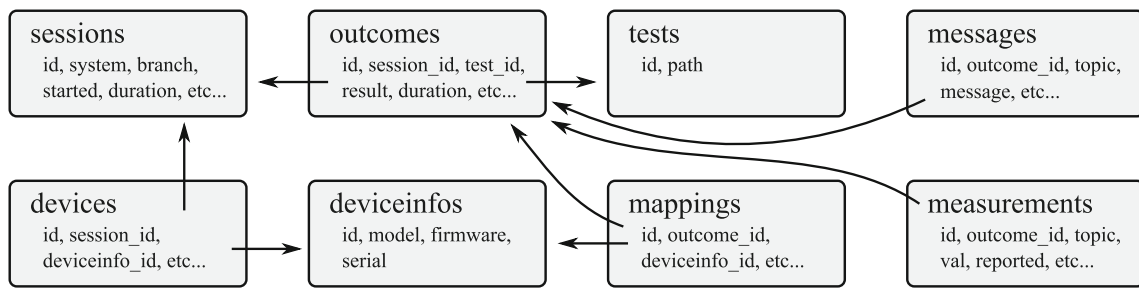


Fig. 4 Screenshot of the outcomes view (mockup in Fig. 8)

which encouraged their use. The patterns are illustrated in Fig. 1 and highlighted with exclamation marks. Examples of usages of the patterns in the Tim views are also provided in the below text.

#### 4.2.1 Filtering

Test results data from nightly regression testing come in many dimensions. From one night, data are generated over test cases, over test systems and over code branches. Each outcome also represents a verdict: In addition to pass or fail, a verdict at Westermo may also be invalid (the test case could



**Fig. 5** Overview of Tim’s test results database: tables as boxes, with fields in text and relations (i.e., foreign keys) as arrows. Example: an outcome (e.g., “a fail”) involves one test case and is part of one session.

Each session typically involves several unique devices that are described as a deviceinfo with details on device model, firmware version and serial number

not run to completion), unmappable (the resources required of a test system were not fulfilled), unloadable (the test case did not exist or had syntax errors) or skipped (when time did not permit continued testing). While looking at test results, one has typically applied a filter along some of these dimensions, sometimes without even realizing it. Answering the question “which tests failed in my branch tonight?” implies filtering along the dimensions: verdict, time and code branch. Exploring test results without filtering is inconceivable—there is just too much data. However, there are limitations in which filters a user interface can support, and which filters can be implemented. Implementing support for all possible filters might be wasteful, but implementing too few filters could frustrate the user as desirable filters might be absent. It is also difficult to fit all possible filters in a user interface in such a way that the user can make sense of them (such as when navigating to the “next” test case in a test session, discussed in Sect. 4.3.3).

Different from filtering is selecting—as opposed to filtering out a set of verdicts or test sessions, a user may want to select one verdict or one session. Based on the selection, one may want to filter within that set. For example, in the mockup of the session view (Fig. 10), we illustrate how a user filters on type of verdict from a selected test session. In the outcome view (Fig. 9), we show one verdict where we

**Table 1** Example output from a database query to generate the data needed for what is shown in Fig. 8

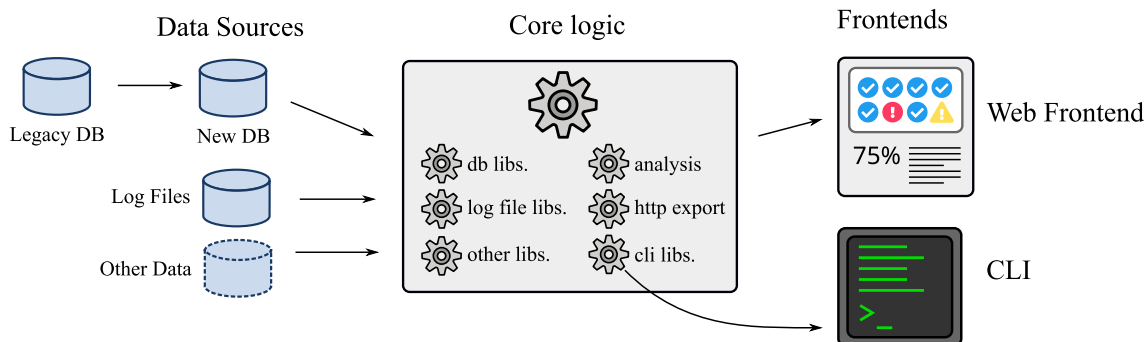
System	Branch	Result	Test	oid
sys1	brA	1	src/test/protocol-a/test1	36822
sys1	brA	2	src/test/protocol-b/test2	36827
sys2	brA	1	src/test/protocol-b/test2	36901
sys3	brA	1	src/test/protocol-b/test2	36976

For the query that generates the table, see main text. At the company, a result of 0 means pass, 1 means fail, 2 means invalid, etc.

filter on which log files to show, and also the filter on time span for log messages to show within a log file.

### 4.2.2 Aggregation

Based on a filtered data set, say all verdicts from a certain night, one may want to aggregate data into certain categories. In the mockup of the start view (Fig. 7), test results from one night are aggregated based on their branch and verdict. In the trend plot, the data are instead aggregated based on date and shown as a time series (Fig. 17). The aggregation may be very natural, like number of fails per night in the last week, or number of executed tests shown branch by branch or system by system. The aggregation may also be more artificial, like



**Fig. 6** Overview of Tim’s architecture



counting verdicts for test cases covering one functional area when compared to test cases for another area—what about test cases that cover both areas?

### 4.2.3 Previews

One of the main tasks when exploring test results is to look at test execution logs, e.g., by reading the test framework log message “a test failed because the firewall did not block packets as expected.” If a user has reached one view, say the list of outcomes, and the system allows the user to peek at the most important log messages from another view, without navigating to that view, then many clicks and context switches would be saved. (Fig. 8 illustrates this.) The old system has no preview function, but in Tim this was so appreciated that we sometimes asked the reference group what a preview would mean in a view that did not have it (such as in the start view), and then we implemented a preview based on their suggestion (for the start view: results per system). In some views, the preview was instead implemented as expand or collapse of details, e.g., in the compare branches and the analyze branch views.

### 4.2.4 Compare

The fourth recurring theme is to compare. The simplest example is to compare logs from two failing tests in the same test session—this way one may determine if the tests failed for the same reason. Comparisons could of course, at least in theory, be done along almost any test results dimension—if more tests fail on one test system and not another, then perhaps there is some hardware dependency in the software? If tests failed more today than yesterday, then perhaps a bug was introduced? If my development branch has other failing tests than the main branch I want to merge it into, then perhaps I would decrease the software quality of the main branch (an example of this is illustrated in Fig. 13)?

## 4.3 Implemented views

Eight views were implemented in Tim. Mockups are shown in Figs. 7, 8, 9, 10, 11, 12, 13, and 14. Each view was either first implemented in Tim and then evaluated at a reference group meeting, or first drawn as a mockup and discussed at a reference group meeting before being implemented. The mockups in the figures have been anonymized and simplified to show principles and to increase readability.

### 4.3.1 The start view

The start view shows an overview of each branch active in nightly testing. The results are aggregated over all systems and color-coded—blue for pass, red for fail, yellow for

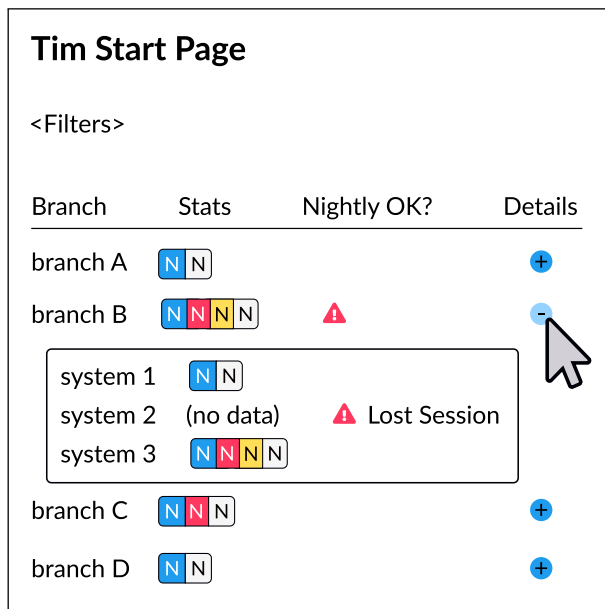


Fig. 7 Mockup of the start view

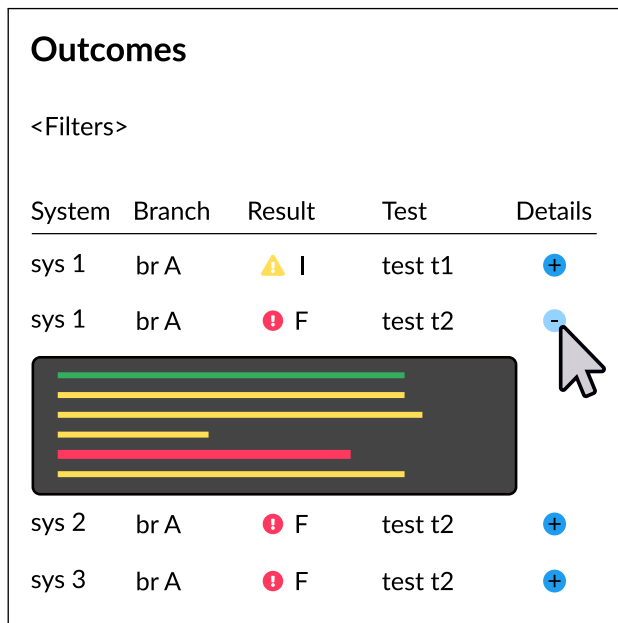


Fig. 8 Mockup of the outcomes view. Log text is illustrated with lines

invalid, and test cases that could have been tested given more resources on time or devices are gray. In order to have redundant information coding, we have strived to never use color alone as carrier of information—it has always been combined with size, shape or position. When colors alone is used to carry information, there is a risk that users cannot receive the information in particular when shown on a projector, when printed in grayscale, or when the person viewing the graphic

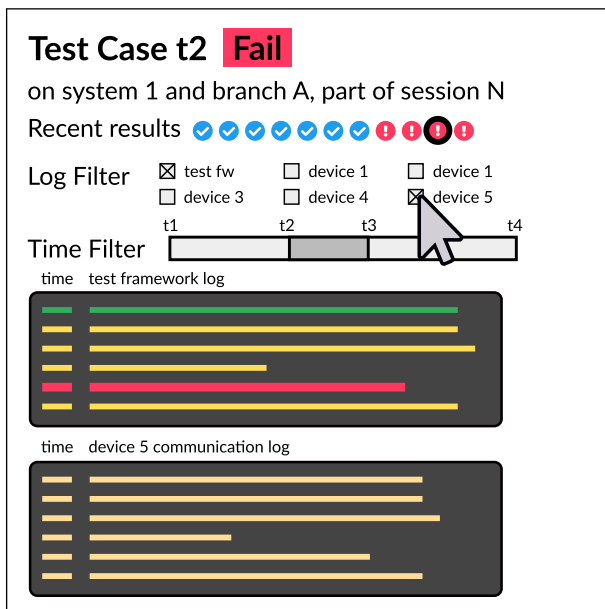


Fig. 9 Mockup of the outcome view. Log text is illustrated with lines

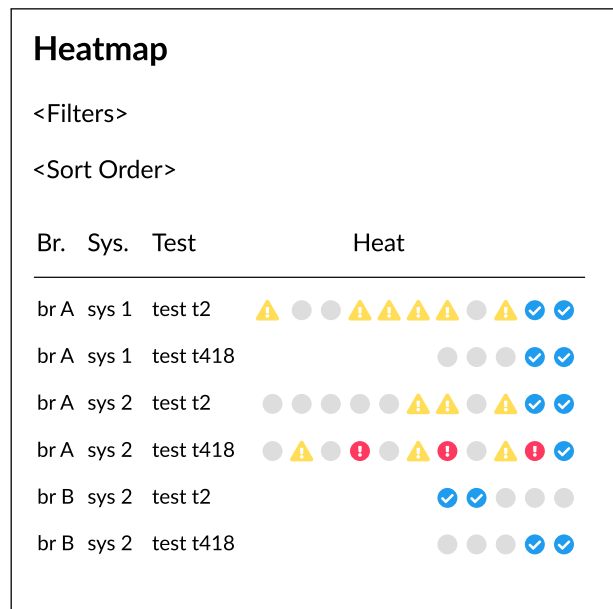


Fig. 11 Mockup of the heatmap view

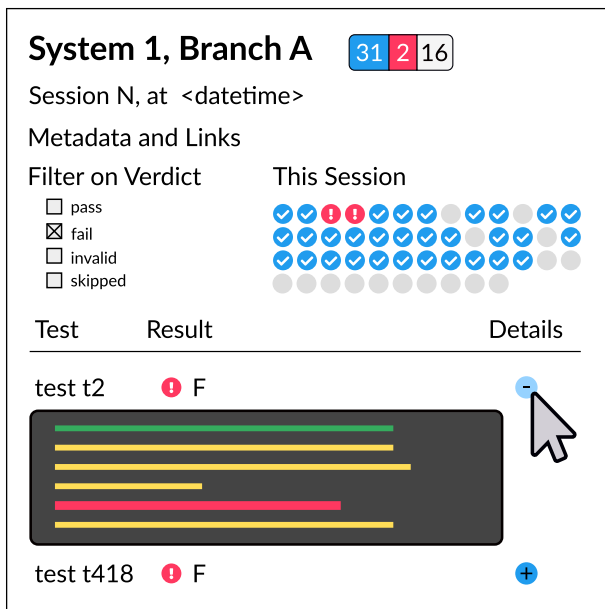


Fig. 10 Mockup of the session view. Log text is illustrated with lines

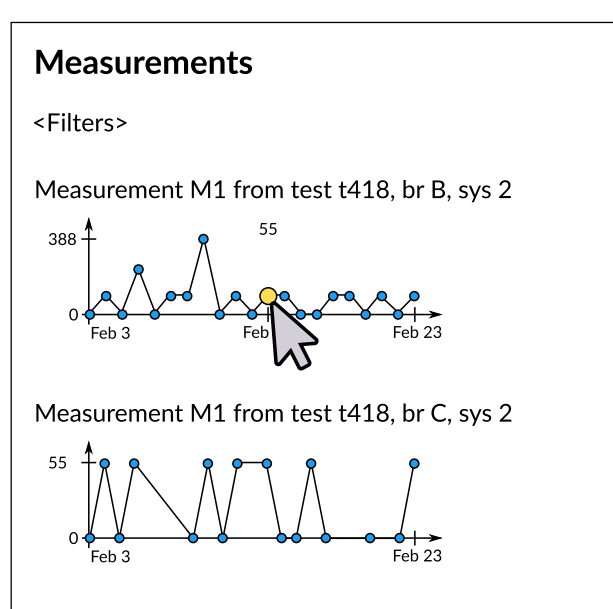


Fig. 12 Mockup of the measurements view

is color-blind [57].<sup>4</sup> Anomalies from nightly testing—such as when a test session should have started but did not, are highlighted with a warning sign. A button can be pressed to expand the results from one branch, thereby showing results from the branch split on test systems. In the example illustrated in Fig. 7, the expand feature shows the user that all of the non-passing tests in this code branch came from one particular test system. This would narrow down the search

<sup>4</sup> Roughly 8% of all men and 0.5% of all women have some form of color blindness [24]; the first author of this paper is one of them.

for the failing tests, and if the user knows that this system also had issues yesterday and that replacement parts have been ordered but not installed, then perhaps no further click is needed. We hypothesize that a non-negligible amount of clicks and effort will be saved.

At first, this view was not organized in a table and it had no details button for a preview. In order to save space in the view, and to fit the potentially many branches on one screen, they were placed in alphabetical order as compact as possible. Some rows would fit many branches and their results, but

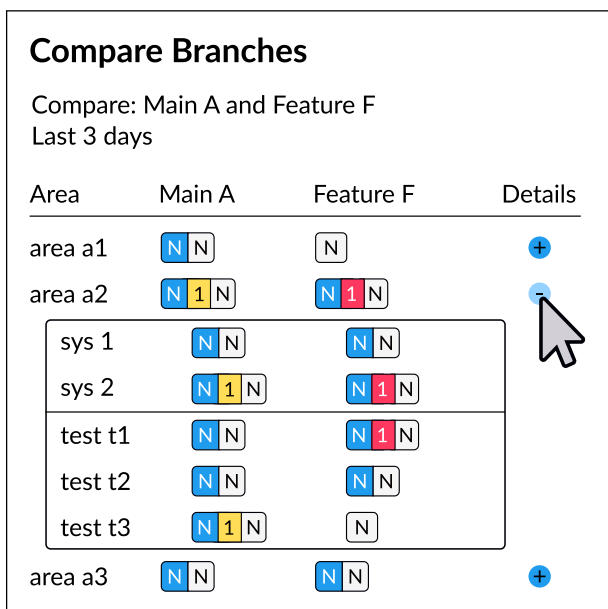


Fig. 13 Mockup of the compare branch view

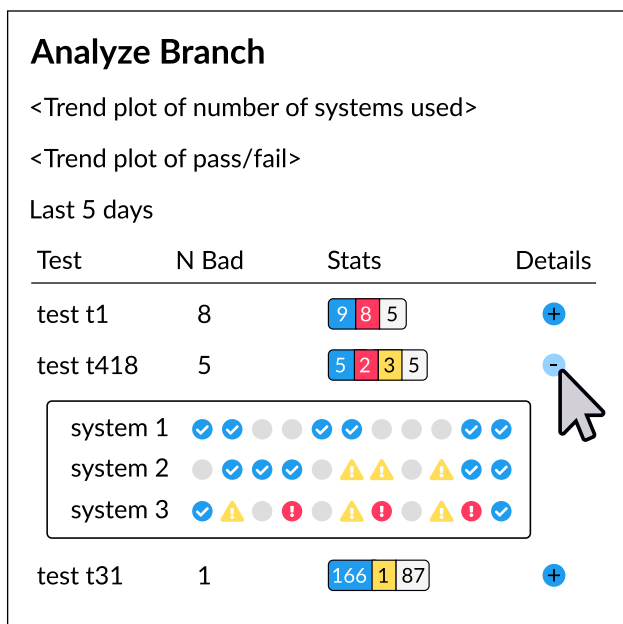


Fig. 14 Mockup of the analyze branch view

others would only fit a few (if the branch names were long). The reference group expressed irritation of the lack of straight lines and they perceived it as hard to use this view. When shown a mockup which led to the implementation shown in Fig. 7, a WeOS developer said “It’s much easier to read...I like when there are columns <waving hands to show>.” One of the members of the test framework team mentioned that the expand feature “would save me clicks...I’d almost never have to click on to the next view before the daily meeting.”

The poor usability of Tim’s initial start page was one of the reasons individuals would keep using the old system. Since Tim is expected to be able to handle a large number of branches, annoying scrolling to find one’s branch is to be expected. We therefore implemented support for filtering on branch name, which might reduce the need for scrolling.

### 4.3.2 The outcomes view

The outcomes view shows filtered test results without aggregating it along any dimension. Each verdict occupies one row in a table that can split over several pages if needed. Typically, a user would enter this view from the start view and the active filters would by default show results from one night, from one branch and only include failing, invalid and unloadable tests. Also in this view, is a list of flaky tests, a list of devices in the test systems used for testing, as well as trend plots for the branches selected (shown in the screenshot in Fig. 4, but not in the mockup in Fig. 8).

The outcomes view also includes a type of preview to see details from the test framework log messages (error messages are shown but not debug information). If several test cases have failed due to one root cause, a user could rapidly figure this out by expanding the preview for these tests, without the need to navigate to another view. Again, this is expected to save clicks and allow the user to keep situational awareness.

The outcomes view was one of the first to be implemented, even before Tim reached alpha-phase, and before the reference group meetings started. Therefore, the reference group discussions were centered around the secondary information in the side bar of this view. Some expressed frustration over the design of the plots (further discussed in Sect. 4.5.6), and others that too much secondary information (such as flaky tests) was shown. As opposed to implementing an “advanced mode,” or some role-based views, where, e.g., a project manager would see fewer details than a software developer, the additional information will remain as is, but stay in the side-bar (see Fig. 4).

### 4.3.3 The outcome view

The outcome view shows the details of one executed test case, on one test system, using one code branch, at one particular point in time, see Fig. 9. Recent results of the same test case, code branch and system are shown in a dotted pattern where each icon represents one verdict. This facilitates understanding when a failure first appeared, if the test case is flaky, and also navigating over verdicts in time. The mockup illustrates a user looking at the third failure in a streak of four failures.

The most important part of the outcome view is for a user to be able to read logs coming from the testing. In addition to the test framework log, there are also communication logs for each device that had a role in the test. The view

allows for zero, one or more log files to be visible at the same time, by clicking check-boxes next to names of the log files. The previous system only allowed viewing one file at a time and users would frequently open multiple browser windows, which risks leading to lost focus when exploring results. Showing multiple logs simplifies comparisons, such as when two devices are expected to behave the same way. By default, the test framework log is shown, and as discussed in Sect. 4.4.5, not all users request any device log.

Log files typically have hundreds or thousands of lines of log information, and seeing all at once can be cumbersome. Two ways of filtering log contents are in place. First, a time filter that acts on the time stamps on each log entry. This is useful, e.g., when the test framework reports of an issue on a particular device at a particular point in time—now a user can filter away all log information except what happened from a few moments before that point in time up until a few moments after. Second, a filter on log level is also present. This acts in the framework log, where each log entry has a severity attached to it, e.g., warning, info, debug, etc. (this filter is not shown in the mockup). The reference group was positive to the time filter, which is new when compared to the old system.

The outcome view also has buttons for navigating to the next or previous verdict in the same test session (not shown in mockup). Implementing this navigation between test cases is not necessarily straightforward—what do we mean when we wish to see the next test case: The next in the session? But if that represents an unmappable test case, would the user like to see the next executed test case? But if that test case passed, would the user like to see the next executed but non-passing verdict? Should these navigation options have a button each? If so, how do we fit so many buttons in the user interface without making the tool too complicated to use?

In the end, the simplest form of navigation was implemented in Tim, one that shows the next test case regardless of verdict.

#### 4.3.4 The session view

Test results are produced by running a test suite in a test session on a test system while having software from a certain WeOS code branch on the devices under test. The results from one such test session can be seen in the session view, see Fig. 10. The view shows, by construction, result from one night, on one test system and one code branch; it has therefore been placed late in the exploration—a user would have to navigate from the start page, through the outcomes and outcome views before reaching it. Instead, the outcomes view is a more central view, and the outcomes view is also more flexible in that it can easily adapt to show several test systems, code branches and also show results from more than one night.

In the session view, the verdicts from each test case in this particular session are illustrated with icons, as in the outcome view. This can be very valuable at times, e.g., when one test case pollutes the system state for consecutive test cases. When this happens, a user would see a continuous line of red or yellow (for fail, invalid or unloadable), instead of scattered red or yellow dots, among the blue and gray tests (for pass, skipped or unmappable). The same type of test framework log file preview as in the outcomes view is used.

Not shown in the mockup is a trend plot of the results from the same branch and system in question, over time. The plot is clickable and allows the user to navigate between test sessions over time. Furthermore, the view supports filtering on test cases in case many are shown, such as when viewing passing tests (not shown in mockup). Again, the session view was one of the first to be implemented (before work on the reference group started)—most discussions on it involved the trend plot and how the filters on test cases works.

#### 4.3.5 The heatmap view

One of the most appreciated views is the heatmap view, see Fig. 11. The main idea of it is to show results over time in a dotted pattern. Each row represents a test case executed over time on one test system in one code branch, where each verdict is illustrated with an icon based on the type of verdict. The view has filters on number of days to show results from as well as systems, branches and test cases to include (not shown in the mockup). The table can also be resorted (sort on test system, code branch or test case). Furthermore, the view shows the separation of the test cases' test script and parameter settings (not shown in the mockup).

The main use case for the heatmap view is to gain an understanding of when and where a test case has been failing. If a test case passed in one code branch, but not in another, then perhaps a problem is present in one of the code branches, but if the test case fails on one particular test system regardless of code branch, then perhaps the hardware of that test system has some shortcoming, and if the test case fails on all branches and systems, then perhaps an error in the test code has been introduced. Furthermore, if tests are intermittently failing or flaky, a user would rapidly see if the intermittence is isolated to a certain test system or code branch, which can be very helpful when root causing the flakiness. By also allowing a user to see results from more than one test case, a number of hypotheses can rapidly be confirmed or rejected. Clicking the icons would lead the user to the outcome view of that verdict.

When shown a mockup of the heatmap view, a member in the reference group said “this is a dream when compared to the old system.” For reference, the old system shows pre-generated images rendered to file. These images are rendered a few times each day, so when looking at them recent results

might not be included. Each of the images in the old TREV tool shows a heatmap for all available test systems for one test case and one branch (without separation of parameter settings). The view then shows all images in branch order in a very long web page. This would make it difficult for a user to compare results from a particular test system as one would have to scroll up and down without being able to see the lines of heat of interest at the same time in the view. Furthermore, in order to compare over test cases, the user would have to open separate windows and look at only one test case per window. The clickability of the old system is also poorer since the user would first have to click one image to open it and then click the verdict to navigate to the outcome view of it. We have been unable to overcome the technical thresholds of removing the middle step in the old system.

In the example illustrated in the mockup (Fig. 11), we see that test cases t2 and t418 had an issue leading to invalid results on branch A on test systems 1 and 2. Two days ago, the issue was repaired and these test cases no longer go invalid on branch A. These tests seem to go pass on branch B. However, t418 on test system 2 seems to also sometimes fail, so perhaps there were two issues present at the same time for this test case on this branch and system, and this issue might not yet have been corrected.

Also note that the dotted pattern of the heatmap view (Fig. 11) and the session view (Fig. 10) is different. In the session view, each verdict in the session is illustrated, as if it was a long word, with no dimension other than time (in the order of seconds or minutes) between each icon. In the heatmap view, typically about a month of time has passed from the leftmost to the rightmost verdicts (the mockup shows eleven nights), and each line represents one combination of test case, parameter settings, test system and WeOS code branch.

#### 4.3.6 The measurements view

In addition to producing passing or failing verdicts, some test cases also generate measurements as a side effect. One such measurement could be the number of milliseconds for a redundancy protocol to react to changes in a network topology. In the measurements view, these measurements are visualized (Fig. 12). Similar to the heatmap view, this allows for comparisons between test systems, test cases and code branches. As illustrated in the example in the mockup, test case 418 has logged values between 0 and 388 on branch B, but no value on branch C go above 55, so something is slower in branch B than in branch C. Not shown in the mockup is the possibility to also see the raw data in a table, in case one would like to copy and paste to a spreadsheet, an email, etc.

The reference group expressed that this view is very helpful for detecting unexpected degradation in performance that has to be corrected. Without the recording of measurements and the visualizations, one might not be aware that code

changes in a feature branch actually had an impact on performance, and only realize it close to a release. It is easier and preferable to detect performance issues in a feature branch and correct before the code is merged to a main branch. (In addition to metrics collected by the Fawltly test framework, Westermo also measures performance with third party measurement tools from Ixia/Keysight, etc.). In discussions with the test manager, a task for future work was created to enhance the measurements data with anomaly detection, so that warnings would be emitted if a performance degradation is detected.

#### 4.3.7 The compare branch view

The view for comparing branches was implemented to support decision making at merge time, typically when a feature branch is to be merged into a main branch. To start with, the user selects one, two, three or four branches to compare. (In theory more branches could be compared, but that would be hard to fit in a usable interface.) The view then shows results aggregated over time and test systems but split by branch and functional area (such as communication protocol or other feature). A concern expressed by several members in the reference group is to add poor quality from a feature branch to the main branch. (The software development process also aims at mitigating this.) They desired a view showing that relevant tests have been running as expected. Also, if there are failing tests in a feature branch, then one must easily be able to tell if the same tests fail in the main branch or not. It would of course be preferable to have no failing tests, but if tests do fail and also fail in the main branch, then one would have less reason to suspect poorer quality in the feature branch than in the main branch.

The compare branch view was based on a common pattern in many shopping websites, where a user can compare, e.g., two different types of mobile phones side by side, and see details category by category (the screen size, screen resolution, memory, etc.). As can be seen in the mockup in Fig. 13, a user can also expand the aggregated data to see the same results split on test system or split on test case. As illustrated in the mockup, area a2 has had some non-pass results, and the details show that these results come from the same test system, but different test cases. The test case t3 is shown to go both pass and invalid in the main branch, but has not been executed in the feature branch. The mockup also shows that area a3 is passing on both branches, but area a1 has no test that has been executed on feature branch F. The reference group was positive to the view: “That’s nice, I like it, like when [the development team] comes and what to run a merge and go through the [quality assurance] checklist, to get an overview. As a project manager I like that.”

#### 4.3.8 The analyze branch view

At release time, the project managers in the reference group expressed an interest in being able to get an overview of any and all failing tests in the last days. To support this, the analyze branch view was implemented and discussed at a reference group meeting with only project managers. Not illustrated in the mockup in Fig. 14 are two plots showing trends of number of passing, failing, invalid and unloadable tests over time, as well as number of test systems used in nightly testing. The main part of the view shows a table of all failures in the last days, aggregated over test case. In the mockup, three tests have at least one non-pass verdict, first t1 with eight bad results, t418 with five and t31 with one. Expanding a row would show the same verdicts split over systems and time, similar to what is shown in the heatmap view (Fig. 11). In this example, we see that t418 produced a number of bad verdicts in the last two weeks, but that it was mostly passing yesterday, and only passing tonight—so perhaps the issue has already been corrected.

Below the table of tests with bad verdicts is also a second table of the test cases that passed, as well as a third of test cases that have not been executed (not shown in mockup).

#### 4.4 Risk-based evaluation of the implemented views

Papers on software visualization often have poor or no evaluation according to Merino et al. [27], who recommend to evaluate in situ with a case study, or to conduct experiments. In addition to a lack of evaluation, the different levels of the visualization come with separate threats and approaches, according to Munzner [28]. In short, we have relied heavily on the reference group for the first three risks mentioned by Munzner. (Solve the wrong problem, use bad data abstractions, or use bad visualizations.) In addition, we have collected web server log data, primarily to evaluate the performance of the back-end for the fourth risk (bad or slow algorithms). The logs were also used to identify a navigational profile. Finally, we have used standard software engineering code quality tools such as lint (static code analysis), and automated tests on the unit and integration levels for the fourth risk.

##### 4.4.1 Risk 1: solved the wrong problem?

The first challenge is to consider if a visualization targets the wrong problem, in Munzner's words, the risk is that "they don't do that." With respect to our study, a risk on this level would, e.g., be to implement visualizations for code complexity of test scripts—it would be interesting to know, but is not at the core of test *results* exploration and visualization. Having an author from the case company is of course one mitigation to this threat, another mitigation is the fre-

quent meetings with the reference group. Munzner suggests to measure the adoption rate at the company, but warns that this is not always a reliable metric: "many well-designed tools fail to be adopted, and some poorly-designed tools win."

##### 4.4.2 Risk 2: bad data abstraction?

In short, one risks to visualize "the wrong thing," e.g., by using incorrect data types. From a test results database perspective, the data types stem from the data collection: log files, verdicts, test sessions, etc. As can be seen in the top of Fig. 2, the number of test systems in use each quarter has grown over time, and there has been a shift at Westermo, from investigating test results in one test session at a time, to aggregating results from several test systems, and sometimes several nights. Without this aggregation, a user would have to investigate "what failed on my branch on test system A tonight?", then "...on test system B?" and so on. One could argue that the PDF test reports from each test session that Westermo used prior to implementing a test results database represents an instance of this risk: visualization focused on sessions, and not results aggregated over systems and time.

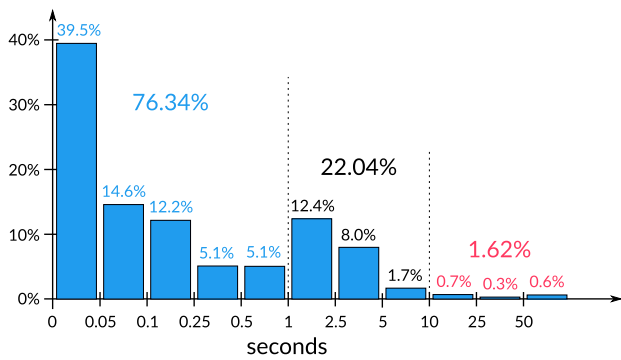
Two approaches suggested by Munzner are to let members of the target audience use the tool and collect data on the usefulness of the tool, which is in line with how we worked with the reference group, or to observe users when they use the tool. We have not conducted long-term observation studies, but consider it for future research.

##### 4.4.3 Risk 3: bad visualizations?

On an interaction (or encoding) level, the risk is that "the way you show it doesn't work." An example of this risk is how the reference group reacted to the trend plots we implemented. Based on comments from the reference group, we allowed them to vote on alternatives and it was obvious that the way we showed it did not work. We discuss the challenge of plots more in Sect. 4.5.6. Munzner suggests to conduct formal user studies on laboratory experiments where individuals perform tasks using visualizations. Another proposed mitigation technique is qualitative discussion using images or videos, which again is in line with how we worked with the reference group.

##### 4.4.4 Risk 4: bad or slow algorithms?

With poor algorithms, one risk is that "your code is too slow," or that the code is wrong and generates incorrect visual elements. Munzner suggests complexity analysis as mitigation. We have, however, only conducted manual code reviews supported with linting and unit tests as part of the development of Tim (i.e., no formal analysis of complexity was conducted). Another suggested mitigation is to measure duration needed to generate visualizations. When we did this, we analyzed



**Fig. 15** Distribution of speed of backend calls. Most bars cover a time interval roughly twice as large as the previous one

logs from 201 days of use and found that about 98.4% of backend calls were faster than 10 seconds (see Fig. 15). One could also consider time required by the frontend to render plots from data collected from the backend, which we have not done. Again, this could be done as part of future work as part of a long-term user study. Regarding the functional correctness, we have implemented automated unit level and database-integration level testing of the backend code. The frontend has been manually tested. We have also used Tim in parallel with the old system to be replaced in order to identify incorrect behaviors.

#### 4.4.5 Navigational profile

Based on 201 days of log data from the web server, we identified the dominating navigational profile in Tim, illustrated in Fig. 16. The data were collected in parallel with developing the system. The typical user first navigates from the start view to the filtered outcomes view, and then to an outcome. Once at a particular outcome, the user would sometimes navigate to the session the outcome was part of, or back. From an outcome, users would sometimes go to other outcomes (e.g., earlier or later in the suite), and sometimes also to the heatmap. Surprisingly, the movement from a heatmap to an outcome was stronger than the opposite direction, which would indicate that a user viewing a heatmap would open several outcomes, or that the user found the heatmap from the navigation bar (not shown in mockups) instead of going to it from the outcome view.

As expected, most log files from nightly testing were viewed in the outcome view, almost 70% of all requests to load a log file came from that view (52.4% for the test framework log and an additional 17.4% for device logs). Also, about 29% came from the outcomes view and an additional 1.4% from the session view. This indicates that quite many users use the preview function in the outcomes view and possibly that some clicks from the outcomes view to the outcome view are “saved.” About 17.4% of all requested log

files were for device communication logs (in the outcome view). We speculate that this typically happens when a user needs more details on what happened, or perhaps in what order. Another way to interpret the data is that in about a third<sup>5</sup> of the cases where user wishes to explore an outcome, he or she also wishes to see a device communication log (the test framework log is shown by default in the outcome view, whereas the device logs have to be added by selecting them).

### 4.5 Challenges identified by reference group

Thematic data analysis from the transcribed reference group meetings indicated the existence of six challenges. These are listed in Table 2, highlighted with question marks in Fig. 1 and discussed in the below subsections.

In order to validate if the reference group agreed that these were indeed challenges, we conducted member checking by presenting the challenges, one by one, and allowing members to vote in a manner similar to poker planning, the level of agreement of the reference group is shown in Table 2. In addition to ranking the challenges, the members were also asked to prioritize future work on Tim. Some tasks were closely related to some challenges. To our surprise, the link between what was a challenge and what was important for future implementation sometimes seemed contradictory. For example, the challenge of hardware details had a relatively low importance, but all three tasks related to it had a very high priority for future implementation. Similarly, the reference group did not agree that the implemented plots represented a challenge, but still they ranked all other possible ways to plot trend data higher and gave the existing plots the incredibly low rank of 1.5 of 5 in terms of how useful it was.

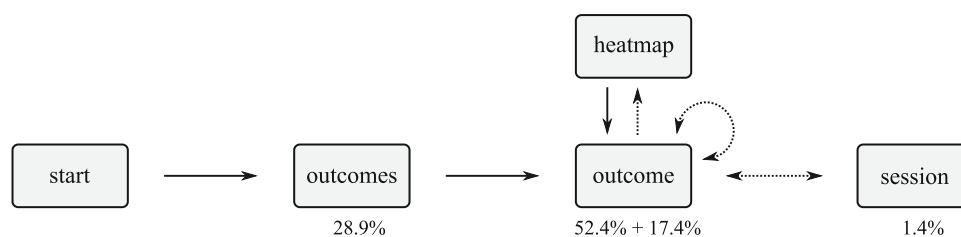
#### 4.5.1 Expectations

The challenge of expectations was phrased as: “I’m not really sure that the right tests are selected over time. Sometimes my branch runs on three systems, and I’m not sure if these are the right ones. It’s hard to know if the testing I get is enough. I don’t have ground truth to compare with.” The reference group voted this as one of the two most important challenges.

The reference group mentioned that this was related to test coverage and test selection, and that, by not having a baseline of what an expected level of testing is, there is a risk of over-testing. It was requested that if the regression test selection tool is trying to prioritize a functional area, but that this feature cannot be tested on the test systems allocated for this branch, then Tim ought to visualize that somehow.

This challenge has some overlap with the next challenge, on anomaly detection—if we could expect something we

<sup>5</sup> 52.4% / 17.4%  $\approx$  3



**Fig. 16** Illustration of the dominating navigational profile in Tim. Portion of log views and previews indicated in percent below each view, for outcomes and session only a preview of the test framework log is available, and for the outcome view the first number indicates the

test framework log and the second number the device communication logs. Solid arrows indicate the main profile and dotted lines a weaker navigational profile

**Table 2** Results after member checking: Challenges are ranked on average score given by participants, from strongly agree (5) to strongly disagree (1)

Challenge	Avg.	Summary
Expectations	4.11	In a large-scale test automation scenario, where very many testing opportunities exist, it's hard to know if a certain level of testing is suitable or inadequate
Anomalies	4.11	In a large-scale test automation scenario, anomalies such as interrupted test suites or flaky tests may be overlooked
Navigation	3.44	Navigating from a high-level view of test results down to a detailed level of test results offers many opportunities for getting lost
Integrations	3.44	A test results exploration system needs to be integrated with other systems, in order to provide a richer exploration of test results, and not only involve test results
HW Details	3.22	The recipient of the test results may not understand enough details on the hardware of the testing context in which the results were produced
Plots	2.44	Plots in Tim ought to be improved in order for users to make more sense of the results

Top four challenges were interpreted as agree, and the bottom two as neutral and disagree

did not get, then that represents an anomaly whose detection could perhaps be automated. The most important task related to any of the challenges (with a score of 4.67 of 5) was “Someone should warn me if a test system only ran 3 tests when we expected 300.”

#### 4.5.2 Anomalies

The challenge of anomalies could be phrased as: “To avoid anomalies in testing (e.g., flaky tests), apply measurements, supervision and/or warnings.” During the reference group meetings, four main anomalies were discussed: test sessions that did not start, sessions that stopped prematurely, tests cases that were flaky and test case dependencies (e.g., where an incorrect cleanup of resources from one test case had an impact on the following test cases). The reference group strongly agreed that anomaly detection is a challenge. However, implementing anomaly detection can be difficult: a member of the test framework team asked whether “we have the right data” for detecting anomalies. Another member of the test framework team also mentioned that without

anomaly detection, some test patterns might be interpreted as anomalies when in fact they are not, e.g., if a test case cannot run on a certain hardware, one might get worried and ask “why has this stopped working” when perhaps it never worked.

Regarding the four anomalies identified, the start view targets lost test sessions (see Fig. 7) and in the outcomes view flaky tests are listed (shown in screenshot in Fig. 4). This leaves the two challenges of sessions that ended prematurely and test case dependencies unaddressed in Tim. Both of these were given tasks in future work that the reference group ranked very high (priority of 4.0 or above).

#### 4.5.3 Navigation

The challenge of navigation was phrased as: “To navigate from a high-level view of results down to a detailed view is hard. It's easy to get lost, break filters, forget what I was doing, etc. Sometimes I just have to start over.” The reference group agreed somewhat with an average score of 3.44 for this challenge. A developer mentioned that this was more of a



problem in the old system; another developer said that “I open new tabs all the time...and then I have tabs on everything I want to see. So I still have the start position...If you only click in one window, then you might more easily get lost.” Other topics discussed was missing navigational paths and the need to sometimes take detours to get from one view to another, e.g., to reach the heatmap of a test case—this was mitigated by adding a path in the navigation bar (not shown in mockups) to a blank heatmap that could then be populated by the user.

Navigation tasks deemed important were related to how the filter on time span worked (e.g., should clicking in a plot result in filters being altered), and error handling in the frontend (e.g., when the backend fails to reply). Initially, the default filter on time span was three nights such that one would not miss results if exploring on a Monday morning, but the reference group members requested it changed to one night. How well the users were aware of which filters they had activated (either the “global” date filter in a dropdown in the navigation bar, or “local” filters within each view) was not explored in this study.

#### 4.5.4 Integrations

A rule of thumb related to the challenge of integrations with other systems could be: “To achieve a richer experience in test results exploration, implement integrations with other systems such as log servers, issue trackers, requirements trackers, etc.” As seen in the overview of Tim’s architecture (Fig. 6), a number of data sources are central to showing test results or log files in Tim. When the reference group was asked whether or not Tim needed integrations with additional systems such as the issue tracker, risk analysis artifacts, the requirements management tool or code changes, their response indicated a weak agreement that this was a challenge. One developer that agreed that this was a challenge mentioned “that the number of tools is increasing...In the long run, it’s better with many things in one tool.” A member of the test framework team that was also of the opinion that this is a challenge mentioned that it is sometimes difficult to go from a change in test results to the code change that triggered the error. The reference group members that disagreed mentioned that the challenge of integrations has mostly been solved now that Tim already has integrations with the test results database and the log file server.

Several tasks for future work were related to integrations, and the ones that the reference group ranked highest were related to getting information on code changes, test system status (i.e., if a test system is deactivated for maintenance) and issue tracker links. After discussions with Tim’s product owner, an integration with the requirements tracking system was deemed to be the most important future integration.

#### 4.5.5 Hardware details

The challenge of hardware details was phrased as “When a test fails on a system, it’s hard for me to understand what hardware was involved. I have to invest a long time to figure out which architecture or port type, that it failed on.” With an average rank of 3.22 we interpret this as a neutral answer—this was not seen as a challenge. A developer argued that “...we have some support in Tim, at least you see what type of [device under test] there is, but then you must know what that means...” They also mentioned that this is mostly a challenge for new colleagues.

Despite this not being seen as a challenge, three tasks for future work with a clear link to hardware details received votes between 4.16 and 4.33. These tasks relate to presentations of what devices a certain test system has, how a test used those resources (i.e., the “mapping,” as discussed in detail in [50]) and the possibility to filter verdicts based on hardware details (perhaps allowing filters like “show me results from firewall tests on test systems where the devices have architecture XYZ”).

#### 4.5.6 Plots

We asked the participants if improved plots would improve their understanding of the test results. The reference group mentioned that improved graphical elements might be more important at merge time, than during development: “for me as a developer, I’m more interested in looking at tests that fail, and see those details, rather than seeing an overview...[when you are] about to merge, and want an overview, then I guess it’s more important.” The reference group ranked this challenge the lowest with an average below 3 which could be interpreted as a weak disagreement.

Several views show a trend plot. However, the plot is rarely the central information of a view, and it is thought of as a support to the core of the view. A trend plot for test results data could potentially show many different things: number of test systems involved in testing, number of passing tests, number of failing tests, percentage of passing or failing tests, etc. In the old system, the pass ratio per test session is plotted, with one line per test system. This led to the problematic scenario where there are dozens of overlapping lines, where most fluctuate between 95 and 100%—in practice, it was difficult to see how a code branch performed with the many overlapping lines. For Tim, we implemented the fail ratio over time, with a circle for each night of testing. The size of the circle was a function of the number of executed tests in the session. If, during one night, all tests passed, the circle was filled with blue, otherwise with a red color. If a circle was hovered with the mouse pointer, it would change color and data labels would be shown (see left in Fig. 17). During the reference group meetings, it became obvious that this plot

was not very well liked. A member of the test framework team said “I get confused about the size and colors,” and at a later meeting “I have to look at it for so long that I give up and look at the details instead.” From the transcripts of the reference group meetings, it also became clear that time that could have been used discussing what we as researchers had planned to discuss was wasted on discussing the plots. We interpret this as a frustration with a plot we, as researchers liked, but that the users wasted time on. To address this frustration, we prepared twelve different types of mockup plots based on data from two real branches: a stable main branch with much testing and only few failures and an unstable feature branch with less testing and a large portion of failures. The members were given the chance to vote for each plot using a similar scale as the challenges and tasks—a scale from 1 (not helpful at all) to 5 (very helpful). The alternatives we suggested included: using a linear or log scale on the  $y$ -axis; splitting pass from non-pass in separate series or aggregate them into one; if the series are split, should they have the same scale or separate scales on the  $y$ -axis; should there be a bar chart or a line plot, and if so, should the values be stacked, etc. The implemented plot received the very poor average of 1.5, which was the lowest result of all the plots. Instead the reference group preferred a plot in log scale where passing verdicts were separated from fail and invalid and on the same scale for both series. As can be seen in the mockup illustrated in the right in Fig. 17, the log scale is somewhat violated to allow zero fails to end up on the  $x$ -axis (where 0.1 should have been had the scale been correct).

## 5 Discussion

In this case study, we have followed the implementation of Tim, a tool for test results exploration and visualization (TREV) at Westermo, a company that develops embedded systems for industry networking applications. The main findings of this paper are (i) four patterns for TREV: filtering, aggregation, previews and comparisons; (ii) the eight views implemented for TREV in Tim; as well as (iii) the identification of six challenges with respect to TREV: expectations, anomalies, navigation, integrations, hardware details and plots. These findings can serve as a starting point, or be of relevance more generally for other researchers or practitioners that strive to implement a TREV tool in a similar or slightly different context. For the case company, the Tim tool is already in use and continued improvements are planned.

Similar studies, such as those stemming from Q-Rapids<sup>6</sup> (discussed on more detail in Sect. 5.3.2), have found similar challenges as we have, which implies that these are not unique to the case company. When it comes to the choices

of views, many tools and visual elements are possible. In their theory of distances, Bjarnason et al. [3] argue that software development practices increase, decrease or bridge distances between actors. Using their terminology one could argue that TREV strives to decrease or bridge cognitive and navigational distances between, on the one hand, actors in the software development process (software developers, test framework developers, project managers, etc.) and on the other hand artifacts such as aggregated, split and/or plain test results and log files.

### 5.1 Revisiting the industry problems and process

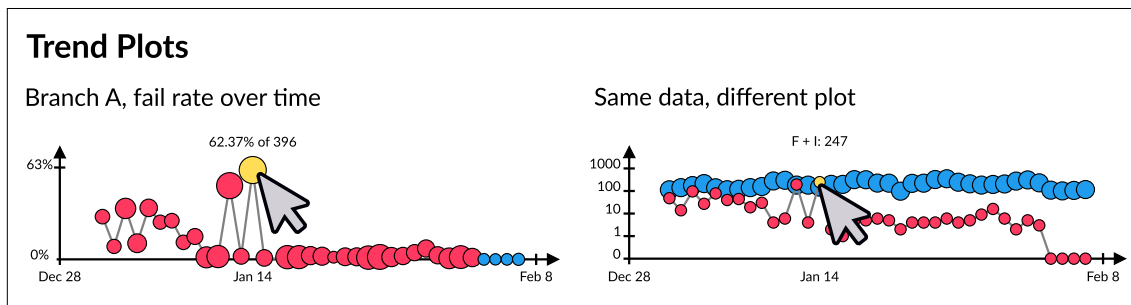
The old system had three main problems. First, to counter the problem of an unknown match between implementation and *user needs*, the same kanban process that is already in use in the test framework team was used also for Tim (including, e.g., requirements, development and testing of functional increments). Requirements were elicited in the form of user stories or tasks, both in our previous work [48] and during the implementation. Prioritization and scoping of tasks was done with the reference group and the test framework team manager.

The second problem was related to *scalability* in that visual elements would not fit on screen and that the performance had degraded. The database performance issues were addressed by refactoring the database layout (see Fig. 5), improving the architecture (Fig. 6) and separating the test environment for the tool from the environment in which it is used. This separation facilitated the implementations of test data generators to allow rapid testing of database calls in the backend, i.e., testing of both the functional correctness and the performance of the backend and its database queries. Furthermore, we logged actual usage of Tim and conclude that 98.4% of backend calls were faster than 10 seconds (Fig. 15), which is acceptable for the needs of the company. Second, to address poor scalability of the user interface, we implemented Tim incrementally with a reference group that also evaluated the views of the tool (Figs. 7, 8, 9, 10, 11, 12, 13, and 14).

The challenge of the too wild *technological flora* was addressed with technological trials, in particular in the pre-alpha period, which also meant that several person months per person were spent on learning the new tools and languages alone. (Some more details on the contents of the technical flora are discussed in Sect. 4.1.)

The views in Tim are enablers for TREV in different phases in the development process. During the implementation, and in the user documentation and training material for using Tim, we used the three phases of daily work, merge time and release time, when explaining the views. In short, the first views are mainly targeting daily work, when a developer desires to dive deep into data and explore individual

<sup>6</sup> <https://www.q-rapids.eu/>



**Fig. 17** Mockups of two types of trend plots based on the same raw data. Left: the plot proposed and implemented by researchers. Right: plot preferred by the reference group

**Table 3** Mapping of the steps in the software development process—daily work (D), branch merge (M) and release (R)—with implemented views in Tim

View	D?	M?	R?	Figures
Start	Y	–	–	7
Outcomes	Y	–	–	8
Outcome	Y	–	–	9
Session	Y	–	–	10
Heatmap	Y	(Y)	–	11
Measurements	(Y)	Y	(Y)	12
Compare branch	–	Y	(Y)	13
Analyze branch	–	(Y)	Y	14

Y indicates that a view is primarily targeting a step in the process and (Y) indicates a partial match

verdicts, debug messages, or perhaps timing within individual log files. In the case of debugging, a user might want to know if an issue is present also in other code branches, on other systems, if it has been present over time and if it is intermittently failing, which motivates the heatmap view. As features have been implemented, a new perspective might be needed: Do we have the desired test coverage? Have the non-functional aspects degraded? Is the main branch we want to merge into as stable as our branch? These questions are addressed by the measurements and compare branch views. Finally, when a new WeOS version is about to be released, the analyze branch view can answer if any test case has failed on this branch in the last few days, and a user can drill down into where (on which test system) and when (in time) the failures occurred. This view also shows the test intensity the branch has received in terms of number of test systems used and test cases executed (not shown in heatmap).

In Table 3, we map the views with the steps in the process, for example: The measurements view can be seen as primarily giving support for work and decisions at merge time, but can also be somewhat useful in daily work and at release time.

## 5.2 Validity analysis

In this section, we discuss the validity of the findings in terms of rigor, relevance, generalizability, construct validity, internal validity and reliability.

At the core of *rigor* are carefully considered and transparent research methods [23,40,42,54]. This study was both planned and also conducted as a case study, based on the well-known guidelines written by Runeson et al. [40]. The study involves both quantitative and qualitative data that were collected and analyzed in a systematic manner.

One way for research to be *relevant* is for the research party and the industry party to share a common understanding of the problem and to be able to communicate [12,19,22,42]. This can be a challenge, as Sannö et al. point out [42], because these two parties typically have differences in perspective with respect to problem formulation, methodology and result. The constructs of this study are rather straightforward—challenges, patterns and views pose no major threats to *construct validity*. It is of course possible that we, in academic communication, “speak another language” than the participants of the reference group, which could lead to threats to construct validity. One could argue that the prolonged involvement and frequency of reference group meetings are part of the mitigation for the threats to both relevance and construct validity.

In a paper titled “...Generalizability is Overrated,” Briand et al. argue just that [7]. Similarly, Hevner et al. argue that one ought to make work in an environment which may decrease generalizability [20]. *Generalizability* explores to what extent the findings are applicable to other researchers, practitioners or domains. Very often case studies claim limited generalizability, and this study is no exception. One could argue that developing a similar tool as Tim for a more general audience (perhaps as part of an open-source tool for unit-level testing, etc.) would have improved the study’s generalizability. However, that might not have incorporated the complexities of working in the industry context (with test

selection, hardware selection, parallel branches, etc.), which are at the core of our work on Tim.

Ralph et al. argue that, for action research, it is essential to cover: the evaluation of the intervention, the reactions from the reference group, as well as a chain of evidence from observations to findings [36]. These are all related to the causality in the study and the *internal validity*. As we have discussed above, the motivation for implementing Tim was driven by problems with requirements, technological flora and scalability. To summarize, desirables for the new system were defined, both in previous work and with the reference group. Implementation was done iteratively and evaluated at reference group meetings. Data were collected at meetings and from logging use of Tim. However, one could ask if we implemented a certain view because it enables test results exploration better than any other view, because the reference group wanted it, or because we as researchers, for some other reason, desired to implement it to see what would happen. Furthermore, during one of the reference group meetings a project manager requested “In general, start migrating existing functionality from the old system into the new system, then work with improvements,” which implies the cognitive bias of anchoring—the users (and researchers) were used to the old system and other systems. In other words, the views in Tim are not free from bias.

Threats to *reliability* can be summarized as “would another researcher in this setting produce the same results?” Implementing visualizations as a researcher is very dependent on the skills a researcher has in a tool—a researcher already skilled in a JavaScript framework other than Vue/Vuex might have favored that instead, and a researcher very skilled in native MacOS GUI development would perhaps have implemented a desktop application for Apple computers, whereas a researcher with expert knowledge in pie charts would have favored those, etc. We speculate, however, that those views would have had many similarities to the ones we produced—perhaps a “pie-stack” with top-pies, sub-pies and sunburst charts could all have been implemented with the same patterns we observed. Perhaps this hypothetical pie-stack would work just as well or better than Tim? In short, other researchers might have produced other views, but we argue that at least some of the patterns would have been similar or the same. For example, one would still have to filter and aggregate at the least, and both comparisons and previews would most likely be useful as well, even for a pie-stack.

Related to validity are the two principles in research ethics on *scientific value*, that “research should yield fruitful results for the good of society, and not be random and unnecessary” and *researcher skill*, “the researchers should have adequate skills” [46]. If we, as researchers, implemented views at random with poor or no skills in, e.g., web development, then the research would not only have poor validity, it would also be unethical. To combat these ethical threats, many

person months have been invested in technological skills (learning database, backend and frontend programming) and researcher skills (participating in a research school, etc.), and we have included what is valuable to society (in particular the case company) when prioritizing implementation.

To conclude, there are validity and ethical threats to this study, but as suggested by Merino et al., Munzner, Runeson et al., Strandberg [27,28,40,46], and many others, we have made efforts to mitigate risks one could have expected by means of triangulation (collecting data using diversity), prolonged involvement (both in terms of knowing the domain, by conducting the study over several months, by collecting data from 201 days of use) and by using member checking, etc.

### 5.3 Related work

This subsection on related work is divided into three parts. First, we cover some of the findings by Ahmad et al. that identified eight information needs in software testing, e.g., “Which test cases... have been run on which product?” By covering these eight needs, we get a set of independent criteria with which we investigate the utility of Tim. Second, we discuss three papers from the Q-Rapids research project. This was a large European project that had many goals in common with Tim. Finally, we cover other related work.

#### 5.3.1 Eight information needs for testing

In a recent case study conducted at six companies involving 38 interviewees, Ahmad et al. [1] identified eight information needs for testing (T1–T8) for companies mostly developing industrial systems with a continuous integration process. These represent a set of needs on a tool like Tim. Their findings are very relevant for our work and represent a complementary set of viewpoints when compared to our division of the development process into three steps, or the three problems identified with the previous tool. In this subsection, we discuss these needs with respect to Tim and our findings.

T1 “Which test cases... have been run on which product?” Ahmad et al.’s first information need for testing relates to being aware of which test cases that have been executed on which hardware product. In Tim, this is supported in the Outcomes view (Fig. 8), where a user is shown a list of test cases that have been run (with optional filtering on verdict) on which test system and code branch. A test system in our case is built up of many devices (products) under test that may have the same or different models, and a list of the devices in use in the test system is shown in this view. Therefore, it seems that Tim supports this information need rather well. However, we note that, just like Ahmad et al., we have identified the challenge of hardware details (Sect. 4.5.5). The reference group gave somewhat ambiguous importance to

this challenge—on the one hand, it was not really a challenge, but on the other hand the reference group gave future implementation on hardware details very high importance.

*T2* “Which test cases... have been run on which branch?” We speculate that the challenge identified by Ahmad et al. on knowing what has been tested on which branch stems from the possible bias of looking at test results session by session, instead of aggregating results in one view. Perhaps these companies have not properly addressed the risk of bad data abstraction, identified by Munzner [28] (discussed more in Sect. 4.4)? Tim solves this issue by aggregating results over test systems and also over time, in the Outcomes view (Fig. 8).

*T3* “In which [test system] do specific test cases fail?” We have also seen that some test cases may fail on certain test systems but not on others. The heatmap view specifically targets this information need and identifies verdicts from one or more test cases, on one or more test systems, for one or more code branches, over time, see Fig. 11.

*T4* “Which test [cases]’ execution times have increased recently?” While Tim stores the duration of each test case (and test session) in the test results database, see Fig. 5, visualizing or showing this information has not been prioritized or requested by the reference group. In the measurements view (Fig. 12), we plot non-functional performance of the system under test, but not performance of the test systems or the test framework. Just like Ahmad et al. have identified, the performance of the test environment ought to be measured, visualized, supervised and would be well served with anomaly detection (a challenge we discuss in 4.5.2).

*T5* “What are the...test results of my commits?” When resources for compiling/building and testing is limited, it might not be trivial to understand if a code change was part of a certain build, and if that build was part of nightly testing, which increases the complexity of this information need and challenge. Just like in Ahmad et al.’s study, our reference group desired information on code changes in the same system as is used for TREV. As we discuss in Sect. 4.5.4, this relates to the challenge of integrating systems with each other. On the one hand, one could desire information on code changes to be integrated in a tool like Tim, but on the other hand information on test results could be integrated in a source code management system.

*T6* “What are the unstable areas of the code that require more ... attention?” In addition to the outcomes view, where results are shown without aggregation, Tim also has two views that target this information need rather well: compare branch (Fig. 13) and analyze branch (Fig. 14). Both these views filter results on a certain branch and aggregates data. The compare view aggregates over functional area, which gives a good idea of test coverage and stability, whereas the analyze view aggregates on test case. Both views also indicate test cases that have not been executed.

*T7* “Which of the test cases are flaky?” Tim identifies intermittently failing tests by means of q-score, a metric we proposed in previous work [49]. Rather than considering flakiness to be a binary property, we consider frequency of change as a scalar value between 0 and 100%—a high value indicates that the test frequently changes verdict (e.g., goes from pass to fail and back again several times per week or month). These tests are listed in a table in the outcomes view (Fig. 4). Furthermore, the level of flakiness can also be seen as changes in colors/icons in the heatmap view. For example, in Fig. 11, we can see that, for branch A on test system 2, the test case 418 has changed verdict each time it has been executed (giving it a q-score of 100%), whereas, for branch A, on test system 1, test case 2 has changed verdict once in 8 executions (giving it a q-score of about 14%, because of seven possible changes, it changed verdict once).

*T8* “What is the test execution history of a specific test case?” Tim supports this question in two main ways. First, when a user is viewing an outcome (Fig. 9), the history of the test case in question is shown as a “heatmap” of verdicts (filtered for the branch and test system in question). Additionally, the heatmap view supports this (Fig. 11).

In addition to the eight information needs for testing discussed here, Ahmad et al. identified a total of 27 information needs for testing, code and commit, confidence level, bugs, as well as artifacts [1].

### 5.3.2 The Q-Rapids project

Q-Rapids was a 5 M€ EU-funded research project striving for increasing quality awareness in agile software development. Many of their findings are relevant to our work on Tim. Their project targets software analytics to support agile development and quality awareness. One difference between Q-Rapids and Tim is that Q-Rapids strives to abstract data into strategic indicators (e.g., from “time needed to resolve issues” to “product quality”), while Tim does not abstract data into KPIs. Another difference is that Q-Rapids seems to strive for assessing and improving quality, while Tim strives to enable exploration and visualization of test results. There are also some similarities, e.g., both tools collect data and visualize it, and both tools support navigation from a high level to a low level. As far as we can tell, they have not in their research explored in details the navigational profile, the views implemented in the tools or how fast database questions are, which we cover in our study.

In one of the publications from Q-Rapids, Martínez-Fernández et al. [26] describe the architecture and principles of their tool and how they evaluated it in a case study at four companies. Some challenges they discuss are related to deploying the tool in different industry contexts, such as tailoring the tool, using a suitable terminology, as well as installation and configuration of the tool. Other challenges

almost perfectly overlap with what our data show: making informative visual elements so that users “understand whether a certain value of an indicator/factor is good or bad” is related to our challenge of expectations and plots. Just like us, they find it challenging with integrations to other systems. Unlike us, they identified a need for transparency and more clarity on actions. On the one hand, it might be the case that when a tool like Tim is used exclusively for data related to testing, then this challenge is not as obvious. On the other hand, this might be related to our challenge of expectations—if a certain amount of testing has been conducted and a user can see two hundred passing tests and one failing test, what should he or she do with that knowledge? It is possible to think of a defined set of actions for Tim, e.g., if a test case is flaky, then Tim could suggest looking for any incorrect assumptions on timing or tolerance ranges in the test case, if the test context is well understood, or if there are resources leaks in the software under test, etc.<sup>7</sup>

Also from the Q-Rapids project is a paper by Ram et al. [38] that investigates what is required of a metric for it to be actionable, and if all metrics ought to call to action or not. One metric they mention concerns how well-defined work tasks are: If the required properties that describe a task has changed in the previous month on many tasks, then perhaps an organization is making a poor job at working with requirements or with describing issues. Our impression is that Q-Rapids have explored this type of metrics in some detail. Their paper lists almost 50 similar metrics in an appendix. In a third paper from Q-Rapids, Ram et al. [37] explored how organizations improved when using software metrics programs. They identified that software metrics can have a positive impact on software processes. The authors observed that, by applying measurements on merge requests, industry practitioners would increase the quality of code reviews. Similarly, Ralph proposed that software processes could be de-biased with metrics or processes such as planning poker [35]. In our work with Tim, we have not explored if the development process has actually improved for the users, instead reference group satisfaction has been used as a proxy.

### 5.3.3 Other related work

Garousi and Elberzhager [17] argue that not only test execution ought to be automated, other activities including test results reporting are well served by automation. Similarly, Garousi et al. [18] explored how industry practitioners perceived challenges in software testing activities. The challenge of test results reporting was perceived as the least severe among nine categories (test automation and test management were the most challenging ones). Suggested research topics

for test results reporting were: (i) maintaining and documenting test results; (ii) metrics and visualization to support test result reporting; (iii) appropriate key performance indicators (KPIs); as well as (iv) visual display of test results. It is possible that for less complex testing, the test reporting might be almost trivial<sup>8</sup> and the need for TREV systems is perhaps minimal. We speculate that the challenges with reporting discussed in our study originates in the relatively complex testing conducted at Westermo—testing is done on a system level, many code branches and test systems are tested each night, not all test cases are executed each night, and not all hardware is used in each test case. The testing in this study might serve as a counter-example of test reporting being a trivial problem.

Related to TREV is software visualization, which, according to Diehl, revolve around structure, behavior or evolution of software [10]. An example of a software visualization is the study by Fernandez and Bergel [14], where they aggregate and split data into a set of graphical elements which also involves filtering, comparing and interactivity. However, this work visualizes the source code and not the test results. Nilsson et al. [30] conducted a study at four companies that mostly developed software intensive embedded systems. Their study targeted the challenges of visualizations from continuous integration, including: getting an overview, slow feedback loops and late testing of non-functional requirements. Regarding the overview, they found it “abundantly clear that very few... if anyone... had a... complete overview of all the testing activities going on...”. Brandtner et al. [5] found that users of a system that had “mashed-up” software quality information from continuous integration performed better. We speculate that the start view of Tim fills a very important role in that it serves as a summary, as a mash-up of several sources and as a gateway to other views.

When it comes to how one could visualize, there is an abundance of research on this. Worth mentioning is that the first system Westermo built on top of the test results database (almost 10 years ago) delivered a frugal test report in plain text (pre-formatted to fit terminal windows 80 columns wide). This could be contrasted with creating a 3D-visualization of test results, as implemented by Borg et al. [4]. Their tool reports on design and verification of an application-specific integrated circuit (ASIC) and specifically target: general exploration of large amounts of test results, localization of error-prone areas and to identify coverage gaps. Similarly, Wang et al. [55] visualized path coverage for model-based testing when generating test cases randomly. Furthermore,

<sup>7</sup> These are three of the nine questions we suggest in our previous work on intermittently failing tests [49].

<sup>8</sup> A trivial test reporting scenario could perhaps involve a developer maintaining a small code base of one or only a few lines of code, in one or a few software modules, with one or a small number of unit tests, that run without the need for dedicated hardware, where the feedback loop is faster than a second, where the results are always unambiguous, and where there is no need to discuss results with colleagues.

Orso et al. [32] visualized test coverage in terms of code coverage at the levels of code, file and system. A common metaphor for TREV is the creation of a dashboard (e.g., [9,16]), which is similar to using mash-ups and also similar what was done in the Q-Rapids project.

One of the challenges we have discussed and a potential enhancement one could build on top of a TREV system is anomaly detection (discussed in Sect. 4.5.2). Oftentimes anomaly detection involves identifying outliers that represent system faults in a dataset by using statistics or AI, such that issues do not propagate or escalate [21,29]. We discussed four types of anomalies and specifically want to raise alarms if they occur. The first anomaly concerns flaky tests—this is a hot research area and three interesting topics involve: root causes for flaky tests [59], when tests become flaky [25], and if we can predict if tests are to be flaky [34]. In our previous work, we have investigated intermittently failing tests at Westermo [49]. Another type of anomaly we discussed, that will remain for future work, is to identify issues in starting test sessions, having sessions that stop too early, or sessions in which one test case in the suite corrupts the system state and makes the following tests fail. In addition to predefined anomalies we expect to capture in Tim, one could also implement anomaly detection or automated fault analysis using the log files coming from testing, not only identifying strings from error messages, but also by investigating surrounding events [33,39,58].

#### 5.4 Future work

We would welcome continued research in software TREV and we envision a few possible paths for future work. First, a high-level path oriented toward technology: How could a tool such as Tim be a better tool if more data sources had been integrated? What is a suitable level of high-level metrics or KPI's—would the decision-making process of various stakeholders improve if Tim also had showed such metrics or should they instead be used in other tools? How should the possibly many tools with overlapping features communicate or interface? Second, we imagine future work toward the visual elements in a TREV tool: How would usability studies (perhaps with eye-tracking) help improve a tool such as Tim? What colors and visual elements should be used? How should one measure and, if needed, improve the performance of the frontend? As TREV tools such as Tim evolve, how do usage patterns change? How could the Tim tool better address the phases in the software development and the information needs? Third, a path oriented toward humans: How is the internal cognitive process of the user affected when visualizing test results? One could imagine incorporating models of the testers' cognitive processes such as the ones proposed by Enoiu et al. [13], or by using fMRI visualizations of the brain while a user is exploring results (similar to what was done

by Duraes et al., e.g., in [11]). Last, we envision a research path toward anomaly detection. Westermo already runs about half a million test cases per quarter. As the testing will continue to scale up, an even larger amount of log files and other data is to be expected. Parsing this data using humans is already difficult. Would empowering the humans involved in these processes with additional tools, perhaps artificial intelligence and machine learning, be required in order to receive the results in a meaningful way from a continued scale-up of the testing?

## 6 Conclusions

Software testing is critical for quality assurance of embedded systems such as industrial communication devices. However, the increased complexity and parallelization of the software development lead to challenges for teams when they explore and visualize test results. In this paper, we have reported on the implementation and evaluation of a tool, Tim, for supporting test results exploration and visualization (TREV) in a continuous integration process with nightly testing at Westermo, a company developing embedded systems. Tim is replacing another system with problems of scalability, requirements and technological flora. The implementation was conducted with a reference group and the test framework team at the company over several months. For validation of the tool, data were collected both from reference group meetings in the form of recorded videos that were transcribed, and we also logged usage of the tool. The data were analyzed quantitatively and qualitatively. The main contributions of this paper are (i) four solution patterns for TREV tools in general (filtering, aggregation, previews and comparisons); (ii) the eight views in Tim (start, outcomes, outcome, session, heatmap, measurements, compare branch and analyze branch); as well as (iii) six challenges frequently discussed at reference group meetings (expectations, anomalies, navigation, integrations, hardware details and plots). Future work could target integrations between a larger number of systems, abstracting data into KPIs, usability studies to increase the performance of users, better understanding the human aspects of exploring results or anomaly detection.

**Acknowledgements** This work was sponsored by Westermo Network Technologies AB, the Swedish Knowledge Foundation (Grants 20150277 & 20170214), the European Union's Horizon 2020 research and innovation program (Grants 871319 & 957212) and the ECSEL Joint Undertaking (JU) under Grant Agreement No. 101007350. Mock-ups were drawn in the Inkscape graphics editor, using colors from the Bulma style framework, icons from Font awesome and emojis from OpenMoji. The screenshot was captured in Firefox 88.0.1 on a computer running Xubuntu 18.04.5 LTS, on a monitor with 4k resolution, then modified for anonymity and scaled to 1400 pixels width in the Gimp graphics editor. The open alpha ran on a virtual machine in VMware and was given one CPU och 4 GB of memory.

**Author Contributions** PES, WA and DS contributed to methodology and writing, review and editing. PES was involved in investigation and writing, original draft. WA and DS contributed to supervision.

**Funding** Open access funding provided by Mälardalen University.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

## References

- Ahmad, A., Leifler, O., Sandahl, K.: Data visualisation in continuous integration and delivery: information needs, challenges, and recommendations. *IET Softw.* (2021). <https://doi.org/10.1049/sfw2.12030>
- Ball, M., Hardie, F.: Effects and detection of intermittent failures in digital systems. In: Proceedings of the November 18–20, 1969, Fall Joint Computer Conference, pp. 329–335. ACM (1969)
- Bjarnason, E., Smolander, K., Engström, E., Runeson, P.: A theory of distances in software engineering. *Inf. Softw. Technol.* **70**, 204–219 (2016)
- Borg, M., Brytting, A., Hansson, D.: Enabling visual design verification analytics—from prototype visualizations to analytics tool using the unity game engine. In: Proceedings of the Design and Verification Conference Europe (DVCon EU), Munich, Germany (2018)
- Brandtner, M., Giger, E., Gall, H.: Supporting continuous integration by mashing-up software quality information. In: Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering. IEEE (2014)
- Braun, V., Clarke, V.: Using thematic analysis in psychology. *Qual. Res. Psychol.* **3**(2), 77–101 (2006)
- Briand, L., Bianculli, D., Nejati, S., Pastore, F., Sabetzadeh, M.: The case for context-driven software engineering research: generalizability is overrated. *IEEE Softw.* **34**(5), 72–75 (2017)
- Cooper, W.F.: Electrical control of dangerous machinery and processes. *J. Inst. Electr. Eng. Part II Power Eng.* **94**(39), 216–232 (1947)
- Deissenboeck, F., Juergens, E., Hummel, B., Wagner, S., Parareda, B.M.Y., Pizka, M.: Tool support for continuous quality control. *IEEE Softw.* **25**(5), 60–67 (2008)
- Diehl, S.: *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, Berlin (2007)
- Duraes, J., Madeira, H., Castelhana, J., Duarte, C., Branco, M. C.: Wap: understanding the brain at software debugging. In: 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), pp. 87–92. IEEE (2016)
- Eldh, S.: Some researcher considerations when conducting empirical studies in industry. In: International Workshop on Conducting Empirical Studies in Industry. IEEE (2013)
- Enoiu, E., Tukseferi, G., Feldt, R.: Towards a model of testers' cognitive processes: software testing as a problem solving approach. In: 2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 272–279. IEEE (2020)
- Fernandez, A., Bergel, A.: A domain-specific language to visualize software evolution. *Inf. Softw. Technol.* **98**, 118–130 (2018)
- Fowler, M.: Eradicating non-determinism in tests (blog post). <https://www.martinfowler.com/articles/nonDeterminism.html> (2011). Online, Accessed 2021-07-05
- Froese, M.-E., Tory, M.: Lessons learned from designing visualization dashboards. *IEEE Comput. Graph. Appl.* **36**(2), 83–89 (2016)
- Garousi, V., Elberzhager, F.: Test automation: not just for test execution. *IEEE Softw.* **34**(2), 90–96 (2017)
- Garousi, V., Felderer, M., Kuhmann, M., Herkiloğlu, K., Eldh, S.: Exploring the industry's challenges in software testing: an empirical study. *J. Softw. Evol. Process* **32**(8), e2251 (2020)
- Garousi, V., Petersen, K., Ozkan, B.: Challenges and best practices in industry-academia collaborations in software engineering: a systematic literature review. *Inf. Softw. Technol.* **79**, 106–127 (2016)
- Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. *Manag. Inf. Syst. Q.* **28**(1), 6 (2008)
- Hodge, V., Austin, J.: A survey of outlier detection methodologies. *Artif. Intell. Rev.* **22**(2), 85–126 (2004)
- Hove, S.E., Anda, B.: Experiences from conducting semi-structured interviews in empirical software engineering research. In: International Software Metrics Symposium. IEEE (2005)
- Ivarsson, M., Gorschek, T.: A method for evaluating rigor and industrial relevance of technology evaluations. *Empir. Softw. Eng.* **16**(3), 365–395 (2011)
- Kalloniatis, M., Luu, C.: The perception of color. In: *The Organization of the Retina and Visual System, Webvision* (2007)
- Lam, W., Winter, S., Wei, A., Xie, T., Marinov, D., Bell, J.: A large-scale longitudinal study of flaky tests. *Proc. ACM Program. Lang.* **4**(OOPSLA), 1–29 (2020)
- Martínez-Fernández, S., Vollmer, A.M., Jedlitschka, A., Franch, X., López, L., Ram, P., Rodríguez, P., Aaramaa, S., Bagnato, A., Choraś, M., Partanen, J.: Continuously assessing and improving software quality with software analytics tools: a case study. *IEEE Access* **7**, 68219–68239 (2019)
- Merino, L., Ghafari, M., Anslow, C., Nierstrasz, O.: A systematic literature review of software visualization evaluation. *J. Syst. Softw.* **144**, 165–180 (2018)
- Munzner, T.: A nested model for visualization design and validation. *IEEE Trans. Vis. Comput. Graph.* **15**(6) 921–928 (2009)
- Muruti, G., Rahim, F.A., Bin Ibrahim, Z.-A.: A survey on anomalies detection techniques and measurement methods. In: 2018 IEEE Conference on Application, Information and Network Security (AINS), pp. 81–86. IEEE (2018)
- Nilsson, A., Bosch, J., Berger, C.: Visualizing testing activities to support continuous integration: a multiple case study. In: International Conference on Agile Software Development. Springer (2014)
- Oppermann, M., Munzner, T.: Data-first visualization design studies. In: 2020 IEEE Workshop on Evaluation and Beyond-Methodological Approaches to Visualization (BELIV), pp. 74–80. IEEE (2020)
- Orso, A., Jones, J., Harrold, M.J.: Visualization of Program-Execution Data for Deployed Software. In: *Symposium on Software Visualization*. ACM (2003)
- Parmeza, E.: Experimental evaluation of tools for mining test execution logs. Master's thesis, Mälardalen University (2020)
- Pinto, G., Miranda, B., Dissanayake, S., d'Amorim, M., Treude, C., Bertolino, A.: What is the vocabulary of flaky tests? In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 492–502 (2020)



35. Ralph, P.: Toward a theory of debiasing software development. In: EuroSymposium on Systems Analysis and Design, pp. 92–105. Springer (2011)
36. Ralph, P., Baltés, S., Bianculli, D., Dittrich, Y., Felderer, M., Feldt, R., Filieri, A., Furia, C.A., Graziotin, D., He, P., et al.: ACM SIGSOFT empirical standards. arXiv preprint [arXiv:2010.03525](https://arxiv.org/abs/2010.03525) (2020)
37. Ram, P., Rodríguez, P., Oivo, M., Bagnato, A., Abherve, A., Choraś, M., Kozik, R.: An empirical investigation into industrial use of software metrics programs. In: International Conference on Product-Focused Software Process Improvement, pp. 419–433. Springer (2020)
38. Ram, P., Rodríguez, P., Oivo, M., Martínez-Fernández, S., Bagnato, A., Choraś, M., Kozik, R., Aaramaa, S., Ahola, M.: Actionable software metrics: an industrial perspective. In: Proceedings of the Evaluation and Assessment in Software Engineering, pp. 240–249. ACM (2020)
39. Rosenberg, C.M., Moonen, L.: Improving problem identification via automated log clustering using dimensionality reduction. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–10 (2018)
40. Runeson, P., Höst, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley, New York (2012)
41. Sabljakovic, H., Djordjevic, M.: Migrating and Evaluating a Testing Framework from a Dynamically to a Statically Typed Language. Master's thesis, Mälardalen University (2018)
42. Sannö, A., Öberg, A.E., Flores-Garcia, E., Jackson, M.: Increasing the impact of industry—academia collaboration through co-production. *Technol. Innov. Manag. Rev.* **9**(4), 37–47 (2019)
43. Sedlmair, M., Meyer, M., Munzner, T.: Design study methodology: reflections from the trenches and the stacks. *IEEE Trans. Vis. Comput. Graph.* **18**(12), 2431–2440 (2012)
44. Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* **5**, 3909–3943 (2017)
45. Staron, M.: Automotive software architectures. Springer (2021). <https://doi.org/10.1007/978-3-030-65939-4>
46. Strandberg, P.E.: Ethical interviews in software engineering. In: International Symposium on Empirical Software Engineering and Measurement (2019)
47. Strandberg, P.E., Afzal, W., Ostrand, T., Weyuker, E., Sundmark, D.: Automated system level regression test prioritization in a nutshell. *IEEE Softw.* **34**(1), 1–10 (2017)
48. Strandberg, P.E., Afzal, W., Sundmark, D.: Decision making and visualizations based on test results. In: International Symposium on Empirical Software Engineering and Measurement (2018)
49. Strandberg, P.E., Ostrand, T.J., Weyuker, E.J., Afzal, W., Sundmark, D.: Intermittently failing tests in the embedded systems domain. In: International Symposium on Software Testing and Analysis, ISSTA 2020. ACM (2020)
50. Strandberg, P.E., Ostrand, T.J., Weyuker, E.J., Sundmark, D., Afzal, W.: Automated test mapping and coverage for network topologies. In: International Symposium on Software Testing and Analysis. ACM (2018)
51. Strandberg, P.E., Sundmark, D., Afzal, W., Ostrand, T.J., Weyuker, E.J.: Experience report: automated system level regression test prioritization using multiple factors. In: International symposium on software reliability engineering. IEEE (2016)
52. Tómasdóttir, K.F., Aniche, M., van Deursen, A.: Why and how javascript developers use linters. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 578–589. IEEE (2017)
53. Tukey, J.W.: *Exploratory Data Analysis*, vol. 2. Pearson, London (1977)
54. Vermeulen, F.: On rigor and relevance: fostering dialectic progress in management research. *Acad. Manag. J.* **48**(6), 978–982 (2005)
55. Wang, R., Artho, C., Kristensen, L.M., Stolz, V.: Visualization and abstractions for execution paths in model-based software testing. In: International Conference on Integrated Formal Methods, pp. 474–492. Springer (2019)
56. Ward, M.O., Grinstein, G., Keim, D.: *Interactive Data Visualization: Foundations, Techniques, and Applications*. CRC Press, Boca Raton (2010)
57. Zeileis, A., Hornik, K., Murrell, P.: Escaping RGBland: selecting colors for statistical graphics. *Comput. Stat. Data Anal.* **53**(9), 3259–3270 (2009)
58. Zhang, X., Xu, Y., Lin, Q., Qiao, B., Zhang, H., Dang, Y., Xie, C., Yang, X., Cheng, Q., Li, Z., et al.: Robust log-based anomaly detection on unstable log data. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 807–817 (2019)
59. Zolfaghari, B., Parizi, R.M., Srivastava, G., Hailemariam, Y.: Root causing, detecting, and fixing flaky tests: state of the art and future roadmap. *Softw. Pract. Exp.* **51**(5), 851–867 (2021)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.