Towards a Test Automation Improvement Model (TAIM)

Sigrid Eldh
Radio System and Technology
Ericsson AB, Stockholm
Dept. Math and Comp. Science,
Karlstad University, Karlstad
Sweden
sigrid.eldh@ericsson.com

Kenneth Andersson,
Andreas Ermedahl
Radio System and Technology
Ericsson AB, Stockholm
Sweden
kenneth.l.andersson@ericsson.com
andreas.ermedahl@ericsson.com

Kristian Wiklund
Radio Base Systems
Ericsson AB, Stockholm
Dept. Innovation, Design and Eng.
Mälardalens Högskola, Västerås
Sweden
kristian.wiklund@ericsson.com

Abstract— In agile software development, industries are becoming more dependent on automated test suites. Thus, the test code quality is an important factor for the overall system quality and maintainability. We propose a Test Automation Improvement Model (TAIM) defining ten key areas and one general area. Each area should be based on measurements, to fill the gap of existing assessments models. The main contribution of this paper is to provide the outline of TAIM and present our intermediate results and some initial metrics to support our model. Our initial target has been the key area targeting implementation and structure of test code. We have used common static measurements to compare the test code and the source code of a unit test automation suite being part of a large complex telecom subsystem. Our intermediate results show that it is possible to outline such an improvement model and our metrics approach seems promising. However, to get a generic useful model to aid test automation evolution and provide for comparable measurements, many problems still remain to be solved. TAIM can as such be viewed as a framework to guide the research on metrics for test automation artifacts.

Keywords—software; industry; test automation; measurement; improvement model; test code;

I. INTRODUCTION

To improve the test and software quality, industries have traditionally measured faults/failures and coverage, number of passed/failed test cases and has in addition relied on improvement- and maturity models, such as Test Process Improvement (TPI) [4], Test Maturity Model (TMM) [15] and Test Improvement Model (TIM) [1]. The area is evolving with new generations of these models, including Test Management Approach (TMAP) Next [24], TPI Next [23] and TMMi (integrated) [26]. These models have in part been successful to aid stepwise improvements for industry but are insufficient in describing improvement steps for test automation with enough detail. Most of the existing test improvement models are based on a subjective, assumed, “order” of importance for improvements [25]. Another caveat of these models is the lack of scientific rigor. As a consequence, we propose a new model of improvement that we call Test Automation Improvement Model (TAIM), where most of the evaluation is based on validated metrics. This enables us to define what could be considered a better, more cost-efficient, objective and mature way to conduct test automation.

TAIM defines ten key areas (KA), and one general area (GA). For each KA we aim to provide measurements, allowing the model to provide objectively defined stepwise improvements. One step should mean better to best aid improvements. Research have in many ways answered how to measure test e.g. by usage of coverage [11] or utilizing mutation tests [30]. Both could be viewed as automated approaches of measurement that provides a factual measurement that can be used to compare two test suites. Coverage is beneficial in supporting assessment of quality, but does not answer all quality questions on a test. Other aspects must be taken into account; e.g. is the test suite maintainable? Can the fully automated test execution suite execute faster? Can any other step of the test process be automated, and what are the cost and consequences of such automation? TAIM must be able to provide measurement to support the “best” automation.

We have taken the first step of validating our model on a large complex telecommunication system, where we have measured properties of test suites and performed detailed comparisons of the test- and source code characteristics. This is by no means a new approach, since Nagappan et.al. [41] bears strong resemblance to this initial attempt. Are these metrics good measurements of what we are after? We are still not convinced. We share some highlights of this metrics study. We can support the test community by gathering know-how of earlier published research and seeking facts by applying and replicating measurements on existing test systems and system under test, as well as contribute to a simplification on how to improve.

Moreover, we believe that our presented TAIM – the main contribution of this paper – should challenge the research community and provide a way to combine the body of knowledge in a more deployable manner for industry.

The reminder of this paper is organized as follows: Chapter II describes related work. Chapter III presents the requirements and observations upon which TAIM is built. Chapter IV presents the GA and KA of TAIM. In Chapter V we present our initial case study and some metrics. The last chapter contains conclusions and further work.
II. RELATED WORK

We define testware as the sub-set of software in a system which targets the automation of testing. Testware equally describes all of the materials (artifacts) used to perform a test in all its phases. Gelperin and Hayashi [15] describes in TMM that the primary issue with testware is reuse, and that to enable reuse, one must support the testware by configuration management, modularization and independence. It is recommended that the test environment shall be automated, flexible and easy to use. This is emphasized by Jacobs et al. [18] discussing the TMM and claiming that the test system is paramount in testing and must be addressed in any test process improvement model. However, no new requirements are proposed to expand the TMM to encounter Test Automation aspects. In TMMi [26] adaptions are made to cater for new processes (agile), but no real test automation advancement is proposed. In “TIM - A Test Improvement Model” [1], Ericson et al. dedicate one of the five KA to testware. The KA is, apart from a requirement on configuration management, primarily concerned about what the testware does, such as automated result checking, and not as much about automation per se and adds in the review KA that testware shall be reviewed.

TPI [4] promotes reusable testware, centrally managed testware with a testware product owner, configuration management, proper delivery procedures for testware, and testware architecture. Heiskanen et al. [14] build on TPI to address concerns for automated test generation. In the field of testware management, they include evaluation of when to automate, and propose that a test environment specialist is included in the staff. TPI and TMM were analyzed by Blaschke et al. [7] according to the requirements of the ISO/IEC 15504 standard. This leads to the definition of a new reference model in line with the standard, where they introduce “Test Environment Operation” as one of the four primary life cycle processes of the model with sub-areas in user support and operational use of the test environment. TMAP Next aims at result driven testing [24] with a more business adapted process improvement, this is also viewed in TPI Next [23]. In these two works from Sogeti, Test Tools and Test Environment gets more in the focus including a better discussion on the management of these, but no metrics support are provided. Kakkonen [25] describes the change over from TPI to the new TPI Next and gives an account of how these lack support in agile process organizations and discusses how these models are based on poor assumptions that does not hold for all types of organizations, which confirms the earlier result provided by Eldh [27] for TPI.

Basili’s well-established approach Goal/Question/Metrics Paradigm (GQM) [31][32][33][34], can support our aim in some of our suggested KA. We are challenging GQM by suggesting a more bottom up approach. We do not want to create goals of “better” from a limited viewpoint (e.g. asking industry) but by defining concepts on what is truly measurable. Our combined approach of using top-down GQM together with a more bottom-up approach would aid in identifying the gaps in research on test automation.

We are approaching this area based on measurement. There is an abundance of measurement frameworks and we have used as our main guidance Abran [42] who is focused on software metrics. One can learn from Baker et al. [36], who provide metrics and refactoring specifically for TTCN-3 test specifications. The test code smells for TTCN-3 by Neukirchen and Bisanz [37] is also highly interesting input. Both papers are targeting existing test artifacts to measure and improve test code. JUnit test patterns have been explored by van Druesen et al. [38] and by Zaidman et al. [39]. Their work on refactoring attempts on test code works towards measuring the quality of the test code, with similarities to our initial metrics. Further research is needed to conclusively confirm refactoring as a strong (and cost-effective) approach to maintainability according to Yamashita [40].

We conclude that TAIM fills a gap in the current research, where KA included in testware and thus in the test automation improvement have not been defined. By creating TAIM we provide ourselves with a goal and a research strategy that put our earlier research in a context [9] and aids in a more holistic approach. Thus, our previously published papers could be seen both as validating and motivating parts for the TAIM. The overall ambition aids us to identify areas missing. For example, we have identified technical debt and impediments in test automation [5][6]. We have looked at the automatic build- and regression test management which is performed after a system fix [10], thereby combining several KA in our model. We have worked with triaging and fault localization [8] as well as fault classifications [12] and have attempted to compare requirements of test tools [13].

III. REQUIREMENTS ON TAIM AND OBSERVATIONS

A. Requirements on TAIM

The below list present the requirements on TAIM:

- **Simplified improvement steps** to enable juxtaposing different automation implementations for each KA. E.g. Step 0 is basis as “no automation” for each KA. Step 1 “part” of the area is automated. Then one must consider full automation which can also be implemented “better” on a further scale.

- **Measurements**: Ability to determine that x is better than y and that provide predictable outcome and direct guidance. Scientifically validated metrics, well defined, and preferably available in tools.

- A Cost/Gain ROI (Return on investment) focus; Using the method, technique, approach including cost of competence needed, should be compare to results and savings

- **Availability**: To deploy tools, artifacts, competences

- Test automation viewed in a more end-to-end life-cycle perspective, instead of only focusing on test execution and/or test generation.

- **Guidelines** for each KA with “major steps” based on thorough literature reviews for each KA and subarea to better define the gap between existing knowledge.

- Support for self-assessment and qualitative complementary information.

338
Domain independence (or suggested priority adaptions) and Dependency Analysis (e.g. hardware)
TAIM is based on a view that all aspects of the test process could be partly or fully automated. However, even though high level of test automation is possible, it is not always economically viable. An improvement model must be economically sound to be suggested as a step.
Process, domain and organization independent; TAIM should allow for some variation of when, not dictate by whom and on what and how.

B. Observation on the TAIM Requirements
The first step of test automation in industry is to boost efficiency by making test cases repetitive [2] [3], by automating the existing test suites. This is performed either by automating a selection of regression tests (usually functional tests) or creating a unit test harness. The result is an automated test execution suite, containing no or some support for verdict evaluation. When the artifacts or the user number grow, test artifacts are often put in a test management system [10]. In agile development, the build, integration and regression testing are highly automated, making industries rely heavily on a successful test suite. There is little evidence that the new way of working results in good test automation. If the focus is only to increase the efficiency of testing, then no or little testing is the fastest way to perform the activity. Thus a thoroughness of testing, through e.g. coverage is a necessity. Measuring only faults found is insufficient for the same reason.

Our experience shows that for complex systems, even if high levels of code coverage are reached at one level, there is no guarantee that enough faults are found, nor that the correct aspects of the system has been considered [11]. In our targeted system, the coverage levels are consistently high and faults are fewer, but this is not enough. Generally speaking, the automation suites can be more or less well made, resulting in more costly maintenance of the test suites. This is important when several agile teams have to check the same source code, should be combined and aggregated to an overall quality of all test artifacts, including the test and the domain. A measurement model that can discuss the costs and step-wise improvements of such investments will provide a better support for investing and deploying such techniques. Thus, a model must take all aspects of automation in the test process into account.

Our identified TAIM KAs provide an end-to-end view of a full automation of the entire test process. The first issue we are facing is if each KA should be viewed independently or together with some of the other areas. Our experience from utilizing earlier models has taught us that any model always needs to be adapted when implemented. It seems that within each area [25], [27] it is difficult to define and describe the implicit assumptions and to make the recommendations true for all businesses. We believe that the problem arises since the models are not based on scientifically validated- and generally applicable measurements. Thus, by utilizing a more measurement-based model, the stepwise improvement might be a “continuous” improvement on a floating scale, where a collection of measurements allow for trade-off discussions and priorities for a particular business. At some point is should be possible to agree on which metrics that could be viewed as “better”. Such comparison could for example be achieved by translating all aspects either into time or costs. This however, is already a tradeoff in itself.

Even if we define a test execution to include result comparison (test oracle) and the result of “pass” and have 100% coverage of a set of test cases, it is still possible to do this automation in many different ways. For example, should preference be given to a test suite that executes fast, or to a test suite with small creation effort? Or, should we spend most of the time in the creation of the test case?
Is high test suite maintainability a strong indicator of good test automation (a cost-over-time view)? Or should we rather look at aspects like “adaptability” to the constant software updates (a product-line view)? Thus, should we add aspects like “well-structured test automation suite” – or are we just digging a grave to our 100% automated test execution for 100% statement (or MC/DC) coverage?
Should we promote high readability of test suites (something which are often important for software maintenance)? This does not really fit newer, more automated testing techniques, e.g. search-based testing [20], where the test cases generated are often almost unreadable.
Defining the better “automation” is a challenge. It is not sufficient to just describe the software under test or domain. Thus, for each KA, there are a multitude of practices that needs to be taken into account.

IV. TEST AUTOMATION IMPROVEMENT MODEL
We have so far identified ten KA and one GA. The overall quality of all test artifacts, including the test and the source code, should be combined and aggregated to an overall result. The list below presents the identified areas. For each KA (1-10) it should be possible to identify aspects provided in the GA.

A. General Area
Consists of: Traceability; Defined measurements such as Efficiency, Effectiveness and Cost (return of investment); Analysis e.g. Trends, Data Aggregation; Testware; Standards; Quality of Automation and Competence.
B. Key Areas in TAIM


2) Test Requirements: Specific Test Automation Requirements e.g. tools or environment, Testability, Analysis of System, Analysis of Architecture (e.g. testability, scope, dependencies/slicing – components/integration levels)

3) Test Specifications: Test Case Generation, Test Design Techniques, Pre-process analysis of test, Build in features e.g. Constraints, Properties, Invariants


5) Test Automation Process: Context, Type, Level.

Process Metrics on e.g. Speed

6) Test Execution: Test Case Selection, Priority, Type of test technique (test goal), Regression Tests

7) Test Verdicts: Test Oracles, Post-Process Analysis, Results (e.g. test case verdict, logs, reports)

8) Test Environment (context): Test environment specification & set up. Type (e.g. Simulation, Emulation, Target), Test Data, Certification suites, API’s

9) Test Tools: Tool Selection, Integration, (Interchange), Tool chains, Tool(s) Architecture, Frameworks, API’s, Components, Installation, Upgrade, Changeability

10) Fault/ Defect handling: Change Reports/Anomaly (Failure/Bug) Reports, Classifications, Fault identification, Triaging, Fault localizations, Fault Correction, Fault Prediction

V. OUR CASE - THE INITIAL VALIDATION

Our case study provides an example of typical industrial software, containing a mixture of propriety and open-source solutions. Our case-study target system is a sub-system of a complex large telecommunication system, which utilizes a proprietary programming language, compiler and linker. The programming language is a variant of DSP-C [35] somewhat adapted to our proprietary multicores hardware. The test code and the source code are written in the same language. Specific set up and closure (clean-up libraries are provided in our propriety testing framework. The testing framework partly integrates some open-source software, such as Google Mock [28] and Jenkins [29].

Our initial validation only scratches the surface. In our validation approach we looked at KA 4 for Test Code and attempted to establish some initial measurable points that could provide valuable insight in our model (well aware of [41] and [42]). The efforts involved in only partly investigate one KA made us realize why we need to invite other researchers to improve on TAIM.

A. KA4 - The Architecture of Test Code

Architecture of test code specifically means how the test code in itself is structured. This goes well with use of standards, templates and patterns, implicitly defining an internal structure of the test case. Architecture is not the same as a test framework (but can be). If it does not dictate how the internal order, grouping etc. is created; the actual test suite architecture is seldom defined. In Table 1 we see some result obtained by measuring the internal structure of the test- and source code.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Source Code</th>
<th>Test Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (kLOC)</td>
<td>853,98</td>
<td>1023,7</td>
</tr>
<tr>
<td>#files</td>
<td>3378</td>
<td>1379</td>
</tr>
<tr>
<td>#function</td>
<td>8802</td>
<td>11744</td>
</tr>
<tr>
<td>#constants/kLOC</td>
<td>82,53</td>
<td>290,98</td>
</tr>
<tr>
<td>#Literal assigns/kLOC</td>
<td>6,53</td>
<td>25,93</td>
</tr>
<tr>
<td>#defines/kLOC</td>
<td>6,53</td>
<td>25,93</td>
</tr>
<tr>
<td>Maintainability Index [19]</td>
<td>51,1</td>
<td>39,1</td>
</tr>
</tbody>
</table>

First, in Table 1 we conclude that the amount of test code (in this case, unit test code) is much larger than the source code. This is our main argument for investments in this research, in the context of cost of maintainability. This gives in itself an important message – how much cost, effort etc. that are invested in creating and maintaining the test code – something that cannot be ignored for any serious business.

Another related observation is that, in comparison to the source code, the test code contains a larger set of functions organized in fewer files. This indicates there is a lack of architecture and organization in the test code itself. This is an example of a claim that we hope to better verify and propose steps to improve. Our case study indicates that there is a lack of sufficient test architects and not enough time spent on test architecture for our specific test cases. Those facts are not measurable in the code and are just indirect. Are library functions in use in the test code? Even if we could not in this case lean on much other than design rules (as guidelines) for the written code, we need to investigate better test patterns. Can we measure this?

B. KA4 - The Standard, Templates and Patterns

Is the use of a test design technique a pattern? In Table 1 we can see a much higher use of constants in the test code in comparison to the source code. With simple variation of them (and the literal assigns and defines) [9] we could probably have a better and more efficient test code (using input variables instead of hard coded constants). Thus, if data indicates overuse of hard coded constants it could be viewed as lack of test design techniques and indicating poor quality of the test cases in its use of test data. We note, however, that some constants are often needed to set up the particulars of the multi-core environment needed for the test case, giving that these specific constants should not be varied. Set up constants is part of this test code pattern. Can we distinguish this from other constants? Can one see an evolution of the test cases and patterns provided? Is test code really similar in structure to source code?
Compared to source code, there is a general lack of guidelines, guidance and education on designing the test code. Moreover, very few of the test suites investigated are explicitly utilizing specific test design techniques. We can again confirm that basic competences in test design techniques [9] should be improved.

C. KA4 - The Test Code Language

The test-code language sub-area can be divided in many ways, e.g. Compiled vs. Interpreted; Functional vs. Procedural, or by other type of groupings: Object oriented or family-wise e.g. C, C++, C#. There is very little research providing good estimates for comparing the use of specific test languages (that is often required in e.g. specific test tools) compared to using the same language for tests as the source code is written in. To use the same programming language for unit test as for source code seems obvious, since it is easier to interface between them. Others insist upon limiting the different unit test frameworks and align them with “higher level” test case frameworks. These claims have not been tested sufficiently, as strategies are not scientifically based.

Maybe usability, ease of constructing test cases, sufficient detail and ease of learning the language should be taken into account at this stage of measurements. Another important factor could be availability and cost of tools that support these test code languages. It is easy in our case to lower the score of proprietary languages, since availability of static and dynamic measurements from e.g. the compiler or tools were non-existent, and we have had to implement a tool from scratch (which is very costly compared to using a simple open source tool) to be able to measure our data. Though, there might be other justifications for a specialist language other than tool availability.

D. KA4 - Static and Dynamic Measurements on Test Code

In comparison with other industrial tests at this level, the dynamic measurement coverage, have very high levels for such complex system in addition to being measured at integration level. These are metrics that indicates in a comparison with other industrial test automations, our implementation should be ranked rather high.

On the other hand, we tried several maintainability indexes that all pointed in the same direction. We describe the used maintainability index [19]. In Table 1 this indicates that both the code and test code are overly complex. Especially the test code value is shockingly low since the target is a number above 80 [19]. We realize that this is largely insufficient data at this point and therefore the collected results are somewhat questionable. However, they provide a first indication that test code just does not measure up to source code in quality.

We have looked at code duplications, see Table 2. The duplication tool compares sets of 3 characters for a code block size of 4. We conclude that the amount of code duplication in the test code is much higher than for the source code. The reason might be that test code writing often becomes a copy-paste activity. Is this good or bad? On one hand, reuse should be encouraged – and without doubt, copy-paste is probably very much how test cases evolve when people learn from each other. On the other hand, proper structuring of test code would encourage the creation of functions or libraries, together with a more keyword- or template-driven approach to writing the test code. Maybe this is not a static metric and instead evidence of code patterns? What are really big enough chunks of duplication? Is not always code this regular?

It is costly to refactor test code as the business gain could be discussed. The perspective should be how to value a test case and its likelihood to find faults. The cost of fixing old test cases should be compared to make new test cases for uncovered areas. The solution is not always straight forward. Further, this is in itself an activity that could be automated. Should automatic test case transformation be a part of TAIM? I would assume that any automatic transformation in regards to testing is an important aspect that has to be considered for the model.

TABLE 2. CODE DUPLICATION (BLOCK SIZE: 4 AND CHARACTERS: 3).

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Source Code</th>
<th>Test Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Files</td>
<td>1971</td>
<td>593</td>
</tr>
<tr>
<td>Lines</td>
<td>205776</td>
<td>248626</td>
</tr>
<tr>
<td>Duplicate LOC</td>
<td>99756</td>
<td>437666</td>
</tr>
<tr>
<td>Total Duplicate Blocks</td>
<td>17672</td>
<td>77298</td>
</tr>
</tbody>
</table>

To collect the above data, both static and dynamic measurements had to be performed. We have implemented a measurement tool, with very basic static measurement common to most static analysis tools, with metrics visible in STREW [41] in addition dynamic measurements e.g. coverage. Moreover, in [9], we could see that open source coverage tools, yielded different results on the same code for the same measurement. It is therefore extra important to discuss the good, better, best when it comes to the quality of the measurements in the data collected, as well as the tool in itself.

VI. CONCLUSION & FURTHER WORK

TAIM can be viewed as work-in-progress and as a research challenge. Our quest for metrics that objectively compare data from the testware is inspiring for further work. We have only started to validate our proposal of gathering measurements by addressing mainly the KA test code. We expect the sub-areas within the KA to evolve as we mature in gathering evidence and validation by better and wider collection of metrics in real systems. Further work is directed to automatically create patterns. It is important to compare TAIM KA with existing models. We intend to further explore measurements and meta-data from the test automation process for a series of languages. We strive to establish a basic set of metrics to further improve and validate the TAIM with aid of the research community.

ACKNOWLEDGMENT