Automatic Test Generation to Improve Scrum for Safety Agile Methodology

Mario Barbareschi*
mario.barbareschi@unina.it
University of Naples Federico II
Naples, Italy

Salvatore Barone*
salvatore.barone@unina.it
University of Naples Federico II
Naples, Italy

Valentina Casola*
casolav@unina.it
University of Naples Federico II
Naples, Italy

Salvatore Della Torca*
salvatore.dellatorca@unina.it
University of Bergamo
Dalmine, Italy
University of Naples Federico II
Naples, Italy

Daniele Lombardi*
daniele.lombardi4@unina.it
University of Naples Federico II
Naples, Italy

ABSTRACT

Continuous compliance and living traceability, i.e., assure the technical quality of the software during the incremental flow of the agile process and trace the requirements' implementation at any time during the development cycle, are two of the most challenging aspects of adopting agile methodologies in the safety critical domain. This is even more true when either user requirements are unstable, the knowledge of the product to be delivered is not enough, or there is no clear interfaces between various hardware/software subsystems, as it may be in a research and development context. In order to reduce the overall cost of these activities, in this manuscript, we discuss benefits resulting from adopting a semi-automatic method to perform continuous compliance and living traceability. The method aims to finding inconsistency between artifacts produced at the end of each iteration by exploit automatic generation of unit tests and coverage metrics. We validated the applicability of the proposed methodology over a real case study from the railway domain, proving it can find inconsistency between several regulations-required artifacts, including the requirements specification, the architectural specification, test specifications and their implementation, and the software implementation.

CCS CONCEPTS

• **Hardware** → *Safety critical systems*; • **Software and its engineering** → **Software testing and debugging**; *Agile software development*.

KEYWORDS

Non-intrusive testing, Automatic Test Generation, Abstract Syntax Three Analysis.

ACM Reference Format:
Mario Barbareschi, Salvatore Barone, Valentina Casola, Salvatore Della Torca, and Daniele Lombardi. 2023. Automatic Test Generation to Improve Scrum for Safety Agile Methodology. In *The 18th International Conference on Availability, Reliability and Security (ARES 2023), August 29–September 01, 2023, Benevento, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3600160.3605061

1 INTRODUCTION

Software development in safety-critical domains remains very challenging, albeit the substantial body of work from the scientific literature pertaining to this field and experiences gained by engineers in years. The latter has flown into several rigorous development processes from international standards, such as those from the International Electrotechnical Commission (IEC), International Organization for Standardization (ISO) and Comité Européen de normalisation en électronique et en électrotechnique (CENELEC), which recommend safe versions of the well-known waterfall model [3, 4]. Anyway, when either user requirements are unstable, the knowledge of the product to be delivered is not enough, or there is no clear interfaces between various hardware/software subsystems, as it may be in a research and development context, waterfall-based processes may not suit. Even if agile methodologies helped to overcome these limitations, they still encounter obstacles when referring to secure and safe systems, where the adoption of automatic tools to design, develop, test and continuously integrate components is conflicting with the need of copying with strict standards, that mainly refer to the traditional waterfall models [2].

Among the many, two of the most challenging aspects of adopting agile methodologies in the safety critical domain are continuous compliance and living traceability. While the former is essential for the agile team to monitor and assure the technical quality of the software during the incremental flow of the agile process, the latter is required to generate a clear trace about the user requirements implementation at any time during the development cycle [2, 12, 14].

Continuous compliance and living traceability imply that the product has to be continuously subjected to Verify & Validate (V&V)

*All authors contributed equally to this research.

ARES 2023, August 29–September 01, 2023, Benevento, Italy

ACM ISBN 979-8-4007-0772-8/23/08…$15.00
https://doi.org/10.1145/3600160.3605061

activities, and that a new software increment can be released at the end of each iteration only after being verified and validated against the applicable standards. Hence, proof-of-conformance evidence is the result of a complex research activity, which has conducted the team to the requirements and architecture that software must have for its intended application. Besides requiring substantial effort from the several teams involved in the project, these activities represent a significant share in the cost items of project management.

In this paper, we discuss how agile methodologies can benefit from a semi-automatic method to perform continuous compliance and living traceability, to discover any inconsistency between artifacts produced at the end of each iteration. The method we propose exploit automatic generation of unit tests and coverage metrics to find any inconsistency between several regulations-required artifacts, including the requirements specification, the architectural specification, test specifications and their implementation, and the software implementation. We discussed how unit tests and coverage measurement can be exploited to monitor the quality of artifacts, providing means to monitor and verify their overall consistency whether the context is research and development oriented. Furthermore, we also discussed how unit tests can be automatically generated leveraging the Abstract Syntax Tree (AST) while targeting the Modified Condition/Decision Coverage (MC/DC) coverage, and, finally, we validated the applicability of the proposed methodology over a real case study from the railway domain.

The remaining of the paper is structured as follows: Section 2 and Section 3 provide the reader with the technical background concerning the software development in the railway domain, focusing on agile development methodologies, while Section 4 and Section 5 discuss our automatic test-case generation approach and its employment, respectively. Finally, Section 6 draws the conclusions.

2 SOFTWARE DEVELOPMENT IN THE RAILWAY DOMAIN

In this Section, we briefly discuss some technical background and regulatory aspects concerning the development of software in the railway domain.

The standard regulating the life-cycle of software in the mentioned domain, i.e., the CENELEC EN 50128 [9], provides a set of requirements with which the development, the V&V, the deployment, and even the maintenance of any safety-related software intended to be used in the railway control and protection shall comply. One key concept is the Safety Integrity Level (SIL), that, as in the IEC EN 61508 [17], is defined in terms of probability of dangerous failures per hour. Four different levels of SIL are defined by the mentioned standard, with level 1 being the least dependable, and level 4 the most dependable. Anyway, the EN 50128 merges SIL levels 1 and 2, and levels 3 and 4, requiring the same techniques and measures to be applied. In the following, we focus SIL-3 and SIL-4 systems, since they are relevant to our case-study.

2.1 Design and implementation phases

Concerning the design of software, the EN 50128 suggests adopting both common good design and programming practices, and additional safety-related restrictions. Adopting a modular approach, for instance, is compulsory, in order to limit the complexity of software and distinguish concerns of each single component, and, furthermore, the regulation also mandates the software shall be able to detect faults, and provide the basis for countermeasures, to minimize the consequences of potential failures.

Pertaining to the implementations, besides adopting common good programming techniques, the regulation suggests a suitable coding-standard and style-guide shall be properly defined. The purpose of the latter is twofold: on one hand, the use of a style-guide deals with the readability of the source code, while, on the other hand, enforcing a coding-standard avoids potential faults which may origin from the adopted programming language.

2.2 V&V phase

An essential role within the software life cycle is fulfilled by V&V activities: their purpose includes, but it is not limited to, finding errors that may have been introduced during the software development. From the V&V perspective, the EN 50128 defines an error as a defect, mistake, or inaccuracy which could result in failure or in a deviation from the intended performance or behavior. Hence, detecting errors is the same as detecting different behaviors from those specified. Furthermore, because errors are defined regarding both requirements and code, the mentioned regulation focuses on requirements-based testing to detect errors before they can become faults or failures [15].

Pertaining to the V&V techniques, the suggested ones include dynamic analysis, functional black-box-testing, traceability and test-coverage, and one between formal proof, static analysis and software error-effect analysis. Besides, test-case shall be defined leveraging boundary-values, equivalence classes and input partitions derived from the requirement specification, and designed (i) to confirm that software performs the intended function, which is commonly verified through black-box testing campaigns, (ii) to check how internal parts of the component interact to carry out the intended functions, which is typically confirmed through black/white-box testing, and (iii) to confirm that all part of the software are tested, through white-box testing.

2.2.1 Structural coverage analysis and metrics. Software testing can only find failures, but it can never be used to prove that no error exists [19]. Therefore, regulations suggest using structural coverage analysis as a completion criterion for the testing effort, and requirements coverage analysis to determine which requirements have and have not been tested. The most commonly adopted structural coverage criterion is branch coverage and compound-condition coverage, using either Multiple-Condition Coverage (MCC) and MC/DC.

The former is a stronger criterion than MC/DC: it guarantees to find every error that is caused by logical decisions, since it requires every entry/exit point in the program to be invoked at least once, and all possible combinations of the outcomes of conditions within each decision to be taken at least once. Hence, given n conditions, the MCC criterion requires 2^n test cases to be specified to accomplish complete coverage.

The exponential trend in the number of test cases is the reason MC/DC is preferred to MCC. Indeed, given n conditions, the number of test cases to be defined to satisfy complete coverage grows

linearly with n. Furthermore, it has been proved that MC/DC is as effective as the MCC criterion in finding errors caused by logical decisions [18]. The criterion requires (i) that every entry/exit point in the program to invoked at least once, (ii) that every decision in the program to take all possible outcomes at least once, (iii) that every condition in a decision to take all possible outcomes at least once, and (iv) that each condition in a decision has shown to independently affect the outcome of the concerned decision.

The latter can be shown by either (a) varying just that condition while holding fixed all other possible conditions, or (b) varying just that condition while holding fixed all other possible conditions that could impact the outcome.

The (b) option allows coping with strongly coupled conditions and with short-circuit logic, i.e., software optimizations that consist of skipping the evaluation of some Boolean expressions which do not influence the decision outcome [5].

Concerning the definition of the minimum set of test-cases to be generated to fulfill the MC/DC criterion, the most challenging part is to show the independent effect of each condition in a decision [7, 8]. Indeed, to show independence, there are at least two cases for each condition where only the condition itself and the decision outcome are toggled. These are called *independent pairs*, and since there may be multiple of the latter for each condition, multiple equally good minimum MC/DC coverage sets can be defined. Consider, for instance, the $Q = A \vee (B \wedge C)$ decision, for instance, whose truth-table is reported in Table 1. Regarding the condition A, for instance, the $\langle 0, 4\rangle$, $\langle 1, 5\rangle$ and $\langle 2, 6\rangle$ independent pairs hold, while the pair $\langle 1, 3\rangle$ and $\langle 2, 3\rangle$ hold for condition B and C, respectively. Hence, the minimum sets of tests $\{1, 2, 3, 4\}$, $\{1, 2, 3, 4\}$, and $\{1, 2, 3, 6\}$ can be defined.

Table 1: Unique-Cause MC/DC with independent pairs

#	A	B	C	$A \vee (B \wedge C)$	A	B	C
0	0	0	0	0	4		
1	0	0	1	0	5	3	
2	0	1	0	0	6		3
3	0	1	1	1		1	2
4	1	0	0	1	0		
5	1	0	1	1	1		
6	1	1	0	1	2		
7	1	1	1	1			

2.2.2 Non-intrusive testing. Structural coverage analysis is usually done by instrumenting the source code to observe information about the taken paths, executed statements and evaluated conditions. Instrumenting decreases the performance of the code significantly, and may be not viable in resource limited embedded systems, because of restrained memory and processing capabilities. Furthermore, it leads to the *probe-effect*, i.e., the timing behavior of the System Under Test (SUT) is altered, possibly hindering the 'testing capabilities.

Modern microprocessors feature Embedded Trace Unit (ETU), that deliver runtime information to debug ports of the processor, allowing for non-intrusive test and debug [10, 11, 23, 24]. The Intel Processor Trace (Intel PT) [16] and the ARM CoreSight [1] are examples of this technology. Both are extensions of Intel and ARM architectures, respectively, that collects information about software execution, e.g., control flow, execution modes, and timings information and formats it into highly compressed binary packets for tracing purpose.

3 SCRUM FOR SAFETY: AN AGILE METHODOLOGY FOR SAFETY-CRITICAL SOFTWARE SYSTEMS

Recently, a novel agile methodology for the development and innovation of safety-critical systems in the railway domain was introduced in [2, 6]: authors developed an extension of the Scrum methodology, namely Scrum for Safety (S4S). The main concept of the S4S workflow is the Safe-Sprint. A Safe-Sprint is a time-boxed iteration that produce a new software increment verified and validated against the standards [2]. The workflow of a Safe-Sprint is described in Fig. 1. Through Safe-Sprints, S4S addresses four main conflicting aspects that exist between agile methodologies and the safety-critical context: (i) documentation, since is a mandatory activity in the safety-critical context but optional in agile methodologies; (ii) requirements, since traditional safety-critical development processes discourage requirement changes, while in an innovation process they have to be progressively refined; (iii) project life-cycle, since safety-critical projects are developed neither iteratively nor incrementally; (iv) testing, since in agile methodologies it is done iteratively during the development, while in the safety-critical system it is done at final stages of development [21]. The latter is addressed by doing several testing phases, in which test developers check whether all the requirements – Scrum user-stories – are correctly implemented and software behaviors as expected. Furthermore, all the new requirements, if any, could adversely impact the already available features; thus a Regression Testing step is mandatory in order to preserve the already achieved technical quality. Finally, with the Integration Test as shown in Fig. 1, integration developers check the software behavior on the target hardware. A fundamental aspect of S4S is the independence of between the design and testing activities, i.e, those who are responsible for testing and integration cannot participate in the software design process; otherwise, verification could fail its objective [2].

One of the core principle of S4S is the continuous V&V, since each developed subsystem must be verified and validated in order to prove that the developed product satisfies its specification. S4S also extends the set of Scrum role with the figures of the Quality Assurance Team, i.e, Verifiers, Validators and Assessors, which are strictly related to critical software development, and adapts their activities in an agile perspective [2]. The purpose is the continuous compliance in order to monitor and assure the technical quality of the software during the agile process, since the Quality Assurance Team has to check the produced Scrum-increment against the software requirements specification, and the applicable standards. S4S allows exploiting automatic Quality Assurance tools and external experts, if any.

It is worth to mention that in the S4S-context, the documentation is only an output. Such documents, i.e., the description of both the development life-cycle and the implemented product used by an assessor, are updated whenever there is a change in any of

the stages of the development process, e.g., a change in requirements. Meaning that proof-of-conformance evidence requested by an assessor is, in that case, the result of a complex research activity, which has conducted the team to the requirements and architecture that software must have for its intended application [2].

4 IMPROVING SCRUM FOR SAFETY THROUGH AUTOMATIC TEST GENERATION FOR MC/DC COVERAGE

The methodology proposed in this paper is applicable under such conditions, that is, in safety-critical systems developed with agile methodologies, i.e., S4S, in a research and development context. In such context, requirements are not stable and are subject due to changes over time. Hence, it is mandatory to ensure that all the artifact produced during the development process are continuously refined and aligned to each other. Specifically, the Software Requirements Specification (SwRS), the Software Architecture Specification (SAS), tests specification and implementation, and the produced software have to be perfectly aligned.

However, in complex systems which evolve quickly over time, managing the continuous alignment between all those artifacts manually would require more time of the system-development itself. Therefore, new methodologies are needed in order to manage and monitor the alignment-state between those artifacts.

The methodology proposed in this paper addresses the alignment management between software requirements specification and test specification and implementation.

4.1 Automatic generation of unit tests

As we mentioned, in order to automatically generate test-cases, we resort to the AST representation of software. The latter is the result of the syntax analysis step of the compilation phase, and it often serves as an intermediate representation of the program onto which compilers work through the compilation and linking steps. Therefore, it represents only structural and content-related details of the program, yet it preserves variable types and their declaration within the source code, the order of statements, the identifiers within the code and their assignment statements, and even the left and the right-hand side operands of binary operators. As depicted in Figure 2, each node of the AST denotes a language construct of the analyzed code, and information pertaining to branches, loops, function-calls, and so forth can be gathered by simply traversing it.

Since the aim is unit testing, we first search for specific nodes within the AST corresponding to function definition. The latter nodes constitute the starting point of the test-generation procedure. Then, since the aim is covering statements involved, for instance, in defense programming, assertive programming, and so forth, we search for *if-then* statements. Furthermore, we also search for *parameter-declaration*, *variable-declaration*, *constant-declaration*, and *literal* nodes, to collect parameters and variables involved in conditions and decisions of the mentioned *if-then* statements.

Decisions and conditions that potentially compose them are identified by searching for *binary-operator* nodes within the AST. Decisions are decomposed in conditions, and based on the data-type of parameters/variables/constants being involved and on the binary operator defining the condition, a proper input assignment is generated to make the concerned condition either *true* or *false*. Consider the variable ≠ value, for instance. To make it or *true* or false, the input assignment variable := value and variable ≠ value' are generated, with *value'* being plausible for the executed code yet such that the condition will be evaluated as *false*. The latter value can be by either obtained through profiling of the application, or user specified.

After suitable input assignments for conditions have been determined, the independent pairs are defined, and a minimum set of test cases for MC/DC is selected for each of the decision within the concerned function. These are implemented, and hence executed, using non-intrusive debugger/traces, as discussed in Section 2.2.2. The latter enables introducing very precise modifications to the value of variables during the execution, allowing to recreate preconditions and conditions for each of the test case.

Last, concerning oracles, i.e., the expected result to be compared with the result obtained during the execution, unfortunately they can be inferred only in a few, simple, cases. Oracle for assert expressions, for instance, can be derived almost straightforwardly, since failure of an assertion causes the system to crash. As for the other test cases, checking for invariant properties – which can be learned by the program execution itself – can be adopted to define oracles in a semi-automatic way. Nevertheless, the scientific literature suggests that inferring perfect invariants is nearly impossible, especially when dealing with very complex software systems [13, 22, 25–27].

4.2 Exploiting unit tests in quality assurance

Auto-generated unit tests are executed before the Quality Assurance stage and the results are exploited in the latter phase by the related team together with requirements coverage tests results. It is worth noticing that the concerned results are the code coverage data – which are, as well, a safety metric in S4S – since they show how much the system requirements and the implemented software are aligned. Specifically, the Quality Assurance team has to investigate the not covered code, since it could be caused by: (i) tests not specified and/or implemented; (ii) one or more new requirements, whose code it's been implemented, however tests to cover that requirements are not been specified and/or implemented; (iii) one or more no longer existing requirements, whose code has not been deleted, and therefore it implements a no longer required behavior; (iv) changes in one or more requirements, which has been specified by new code that has not tested yet; (v) dead or redundant code; (vi) unreachable code.

Therefore, through such information, it is possible to investigate the overall quality of the system and the alignment between its artifacts.

5 CASE STUDY

We applied the approach discussed in the previous Section to a real-world safety-critical application from the railway domain. The considered software implements part of the functionalities of the European Rail Traffic Management System/European Train Control System (ERTMS/ETCS) standard, and it consists of many periodic real-time tasks, whose execution flow, albeit quite complex, can be summarized as tasks either behaving as *producers*, or as *consumers*.

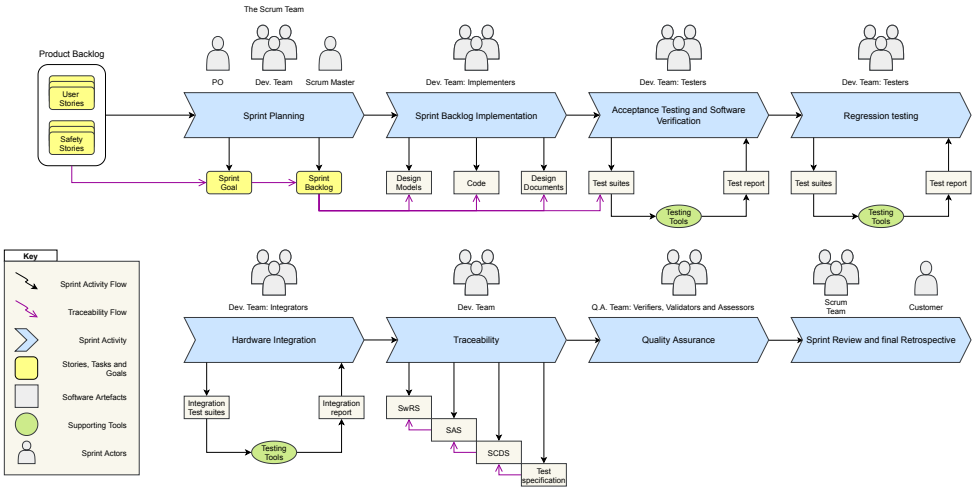

Figure 1: S4S Safe-Sprint Structure [2]

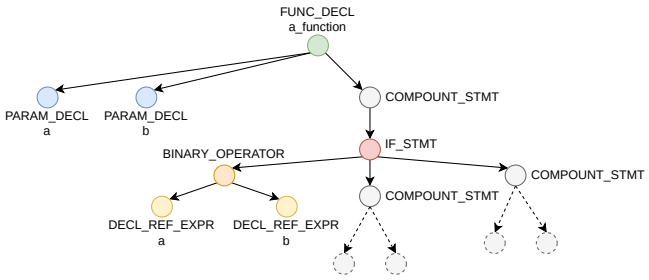

Figure 2: Example of AST

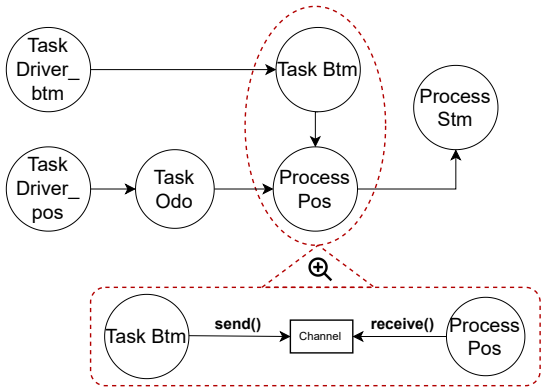

Figure 3: An application from the ERTMS/ETCS standard.

The considered software monitors the location of the train on the rail route. It consists of different processing units: *Pos* is the one assigned to calculate the precise position, based on the odometric information and that from the eurobalises, respectively, obtained by interfacing with the *Odo* and *Btm* units. The latter perform decoding and pre-processing operations of the information coming from the *Driver_Odo* and *Driver_Btm* tasks, which implement the device driver functions.

The whole system operates under memory segregation constraints, and any communication between software units is arbitrated by the Real-Time Operating System (RTOS) through message-passing inter-process comunications (IPCs). The RTOS also orchestrate communications between tasks and the underlying hardware.

The mentioned software runs on four ARM® Cortex-A53, and it is two-out-of-two (2oo2) redundant, meaning that the software is executed by two replicas, which receive the same inputs and are expected to provide the same output. The deployment to specific cores is statically defined by the vendor, to guarantee a higher system predictability.

Given the complexity, we consider only a subset of the mentioned software, as shown in Figure 3, to make the discussion plain. Specifically, we focus on the communication between *Task Btm* and *Task Pos*, by considering several software components being part of the latter tasks.

In order to capture the system behavior, we relied on the ARM ETMv4 [1] embedded into the Zynq Ultrascale+ architecture. Furthermore, we exploited the Lauterbach Trace32 debugger/tracer [20] to end to process execution traces to obtain coverage statistics.

Table 2: Initial structural coverage

Component	Statement (%)	Conditions (%)	Decisions (%)
A	92.105%	92.105%	94.736%
B	90.0%	90.0%	90.0%
C	85.294%	85.294%	90.625%
D	92.875%	92.875%	94.736%
E	81.294%	83.2%	88.6%

The concerned components underwent a previous unit test campaign, the definition of which was derived directly from the SwRS, and the achieved coverage is reported in Table 2.

Let us consider the component A first: by applying the approach described in the previous Section, thirteen test cases are generated, i.e., five more than those manually specified starting from the SwRS,

whose coverage data are shown in the Table 2. By analyzing the coverage results of the remaining five test cases, it was found that: (i) a test case cover the code related to a suppressed requirement; therefore, there was a misalignment between the SwRS and the implemented software; (ii) two test cases cover dead code; and (iii) the last two test cases cover an existing requirement, whose software was implemented, but the related test specification was missing in the SwRS.

Further, similar conclusions emerged when analyzing the coverage data of the components B, C and E.

Component D, on the other hand, deserves a separate discussion, since only two more test cases were auto-generated. Nevertheless, from the subsequent analysis, we found that the portions they covered were, in fact, redundant code. Moreover, despite the automatic generation procedure, the code was still found to be not fully covered, requiring a thorough inspection, the result of which was the discovery of unreachable code, probably caused by a poor implementation of the specification.

6 CONCLUSION

In this paper, we discussed how agile methodologies can benefit from the automatic generation of unit tests. In particular, we discussed how unit tests and coverage measurement can be exploited to monitor the quality of artifacts, providing means to monitor and verify their overall consistency whether the context is research and development oriented. Furthermore, we also discussed how unit tests can be automatically generated leveraging the AST while targeting the MC/DC coverage, how the latter tests can be exploited to verify the consistency of several regulations-required artifacts, including the requirements specification, the architectural specification, test specifications and their implementation, and the software implementation. Finally, ve validated the applicability of the proposed methodology over a real case study from the railway domain.

REFERENCES

[1] ARM Limited. 2013. *CoreSight Architecture Specification v2.0, issue D.* Technical Report. ARM Limited. https://documentation-service.arm.com/static/5f9009d5f86e16515cdc0417?token=

[2] Mario Barbareschi, Salvatore Barone, Riccardo Carbone, and Valentina Casola. 2022. Scrum for safety: an agile methodology for safety-critical software systems. *Software Quality Journal* (July 2022). https://doi.org/10.1007/s11219-022-09593-2

[3] Mario Barbareschi, Salvatore Barone, Valentina Casola, Pasquale Montone, and Alberto Moriconi. 2022. A Memory Protection Strategy for Resource Constrained Devices in Safety Critical Applications. In *2022 6th International Conference on System Reliability and Safety (ICSRS)*. IEEE, 533–538.

[4] Mario Barbareschi, Salvatore Barone, Alfonso Fezza, and Erasmo La Montagna. 2021. Enforcing Mutual Authentication and Confidentiality in Wireless Sensor Networks Using Physically Unclonable Functions: A Case Study. In *Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, September 8–11, 2021, Proceedings 14*. Springer, 297–310.

[5] Jan A. Bergstra, A. Ponse, and D. J. C. Staudt. 2013. Short-circuit logic. *arXiv:1010.3674 [cs, math]* (March 2013). http://arxiv.org/abs/1010.3674 arXiv: 1010.3674.

[6] Riccardo Carbone, Salvatore Barone, Mario Barbareschi, and Valentina Casola. 2021. Scrum for Safety: Agile Development in Safety-Critical Software Systems. In *Quality of Information and Communications Technology (Communications in Computer and Information Science)*, Ana C. R. Paiva, Ana Rosa Cavalli, Paula Ventura Martins, and Ricardo Pérez-Castillo (Eds.). Springer International Publishing, Cham, 127–140. https://doi.org/10.1007/978-3-030-85347-1_10

[7] John Joseph Chilenski. 2001. *Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion.* Technical Report. United States. Federal Aviation Administration. Office of Aviation Research.

[8] Cyrille Comar, Jerome Guitton, Olivier Hainque, and Thomas Quinot. 2012. Formalization and comparison of MCDC and object branch coverage criteria. In *Embedded Real Time Software and Systems (ERTS2012)*.

[9] Comité européen de normalisation en électronique et en électrotechnique. 2011. *Railway applications - Communication, signalling and processing systems Software for railway control and protection systems.* Technical Report. Comité européen de normalisation en électronique et en électrotechnique.

[10] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. 2018. Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing. In *Runtime Verification*, Christian Colombo and Martin Leucker (Eds.). Vol. 11237. Springer International Publishing, Cham, 43–63. https://doi.org/10.1007/978-3-030-03769-7_5 Series Title: Lecture Notes in Computer Science.

[11] Normann Decker, Boris Dreyer, Philip Gottschling, Christian Hochberger, Alexander Lange, Martin Leucker, Torben Scheffel, Simon Wegener, and Alexander Weiss. 2018. Online analysis of debug trace data for embedded systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 851–856. https://doi.org/10.23919/DATE.2018.8342124 ISSN: 1558-1101.

[12] Brian Fitzgerald, Klaas-Jan Stol, Ryan O'Sullivan, and Donal O'Brien. 2013. Scaling agile methods to regulated environments: An industry case study. In *2013 35th International Conference on Software Engineering (ICSE)*. 863–872. https://doi.org/10.1109/ICSE.2013.6606635 ISSN: 1558-1225.

[13] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Transactions on Software Engineering and Methodology* 24, 4 (Sept. 2015), 1–49. https://doi.org/10.1145/2699688

[14] Geir Kjetil Hanssen, Tor Stålhane, and Thor Myklebust. 2018. *SafeScrum® – Agile Development of Safety-Critical Software.* Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-99334-8

[15] Kelly J Hayhurst. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage.* DIANE Publishing. Google-Books-ID: aqMz3xtU6HsC.

[16] Intel. 2021. *Intel® 64 and IA-32 Architectures Software Developer's Manuals.* Technical Report. Intel. https://cdrdv2.intel.com/v1/dl/getContent/671200

[17] International Electrotechnical Commission. 2010. *Functional safety of electrical/electronic/programmable electronic safety-related systems.* Technical Report. International Electrotechnical Commission.

[18] Susanne Kandl and Sandeep Chandrashekar. 2015. Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. *Computing* 97, 3 (March 2015), 261–279. https://doi.org/10.1007/s00607-014-0418-5

[19] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. 1999. *Testing Computer Software.* John Wiley & Sons.

[20] Lauterbach Development Tools. 2021. *ARM Debugger.* Technical Report. Lauterbach GmbH. https://www2.lauterbach.com/pdf/debugger_arm.pdf

[21] Fergal McCaffery, Kitija Trektere, and Ozden Ozcan-Top. 2016. Agile – Is it Suitable for Medical Device Software Development?. In *Software Process Improvement and Capability Determination (Communications in Computer and Information Science)*, Paul M. Clarke, Rory V. O'Connor, Terry Rout, and Alec Dorling (Eds.). Springer International Publishing, Cham, 417–422. https://doi.org/10.1007/978-3-319-38980-6_30

[22] ThanhVu Nguyen, Matthew B. Dwyer, and Willem Visser. 2017. Symlnfer: Inferring program invariants using symbolic states. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 804–814. https://doi.org/10.1109/ASE.2017.8115691

[23] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla, and P. Martín-Holgado. 2019. Online Error Detection Through Trace Infrastructure in ARM Microprocessors. *IEEE Transactions on Nuclear Science* 66, 7 (July 2019), 1457–1464. https://doi.org/10.1109/TNS.2019.2921767 Conference Name: IEEE Transactions on Nuclear Science.

[24] José Rufino, António Casimiro, Felix Dino Lange, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. 2018. Non-intrusive Runtime Verification within a System-on-Chip. *Ada User Journal* 39, 4 (2018), 4.

[25] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. 2012. Understanding user understanding: determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 188–198. https://doi.org/10.1145/2338965.2336776

[26] Chunhui Wang, Fabrizio Pastore, and Lionel Briand. 2018. Automated Generation of Constraints from Use Case Specifications to Support System Testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 23–33. https://doi.org/10.1109/ICST.2018.00013

[27] Anand Yeolekar, Divyesh Unadkat, Vivek Agarwal, Shrawan Kumar, and R. Venkatesh. 2013. Scaling Model Checking for Test Generation Using Dynamic Inference. In *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*. 184–191. https://doi.org/10.1109/ICST.2013.29 ISSN: 2159-4848.