SmartDelta

ITEA 3

SmartDelta

Automated Quality Assurance and Optimization in Incremental Industrial Software Systems Development

D3.4 - Delta-oriented Quality Assurance Methodology

Submission date of deliverable: Nov 30, 2024

Edited by: Eduard Enoiu (MDU) and SmartDelta WP3 partners

Project start date	Dec 1, 2021
Project duration	36 months
Project coordinator	Dr. Mehrdad Saadatmand, RISE Research Institutes of Sweden
Project number & call	20023 - ITEA 3 Call 7
Project website	https://itea4.org/project/smartdelta.html & https://smartdelta.org/

Contributing partners	WP3 partners
Version number	10
Work package	WP3
Work package leader	Eduard Paul Enoiu (MDU)
Dissemination level	Public
Description *(max 5 lines)*	D3.4 aims to describe the methodology for delta-aware quality assurance for industrial systems based on functional and extra-functional testing, monitoring, automated static and dynamic analysis, and regression testing.

Executive Summary

This deliverable presents the final version of the Delta-Oriented Quality Assurance (DOQA) methodology developed within the SmartDelta project. It outlines the approaches for ensuring software quality in incremental development scenarios, particularly industrial systems. The methodology integrates automated testing, functional and extra-functional property assurance (e.g., security, performance, resource consumption), regression testing, and static and dynamic analysis to address the unique challenges of continuous software evolution.

Key advancements include automated test generation, augmentation, and prioritization strategies tailored for delta-specific changes, enabling efficient resource use and improved CI/CD workflows. The methodology introduces tools and techniques for requirements management, anomaly detection, and static analysis. In addition to addressing functional requirements, the methodology incorporates model-based testing for non-functional properties. Methods such as combinatorial test generation, anomaly discovery for microservices, and change-based regression test selection leverage techniques like semantic analysis, static requirement checking, and test amplification. SmartDelta enhances DOQA processes, reduces testing cycles, and ensures system quality across iterative increments.

Integrating these tools into agile practices, supported by guidelines for implementation in real-world industrial contexts, will benefit practitioners and researchers. This deliverable also explores the methodology's broader applications, aligning its tools with specific use cases and evaluating their effectiveness through performance metrics. Recommendations for future work include expanding tool capabilities and exploring additional integrations to optimize delta-aware DOQA processes further.

© 2024 SmartDelta Consortium

Table of Contents

Executive Summary 2

Document Glossary 4

2. State of the Art 6

2.1. Model-Based Testing of Industrial Product Lines 6
2.2. Maintaining/Improving the Quality of Regression Tests 7
2.3. Property-based testing for non-functional requirements 7
2.4. Test co-evolution and repair 8
2.5. Test reuse across products 9
2.6. Change-based regression test selection 10
2.7. Mutation Testing (for Simulink) 14
2.8. Reflections and Identified Gaps 15

3. SmartDelta Methods for Quality Assurance 16

3.1. Combinatorial Test Generation (SEAFOX) 16
3.2. Requirement and Test Specification Static Checker (NALABS) 17
3.3. Discovery of Interactions and Anomalies for Microservices (DIA4M) 19
3.4. Model-Based Test Case Generation (IFAK-TCG) 25
3.5. Test Amplification (FOKUS-CBTS) 25
3.6. VARA+: Variability-Aware Requirements Management and Analysis 29
3.7. SoHist - Historical Quality Evolution Tool 33
3.8. SONATA 36
3.9. UI Test Generator - User Interface Test Design 37
3.10. MUT4SLX – Mutation Testing for Simulink 40
3.11. DRACONIS 42
3.12. Test Effort Estimator 45
3.13. PyLC 46
3.14. GW2UPPAAL 50
3.15. Testing Utilizing EARS Notation in PLC Programs 51
3.16. Jazure 55
3.17. Smellyzer 56
3.18. Relink 58
3.19. Telemetry Anomaly Analyzer (TAA) 59

4. A Methodology for Delta-Aware Quality Assurance 60

5. Broader Context of the SmartDelta Methodology 66

5.1. Selected Tools and Their Role in WP3 66
5.2. Example: How WP3 Tools Fit Into a Real Use Case 68

6. Selected Performance Metrics – An Overview 68

7. Summary and Conclusions 71

© 2024 SmartDelta Consortium

Document Glossary

Acronym	Definition
AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
CI/CD	Continuous Integration / Continuous Deployment
GUI	Graphical User Interface
ML	Machine Learning
NLP	Natural Language Processing
QA	Quality Assurance
TCS	Test Case Selection
TCP	Test Case Prioritization
TSM	Test Suite Minimization
EARS	Easy Approach to Requirements Syntax
SPENAT	Safe Petri Net with Attributes
PBT	Property-Based Testing
POM	Project Object Model
XML	Extensible Markup Language
CSV	Comma-Separated Values
SoC	Separation of Concerns

© 2024 SmartDelta Consortium

1. Introduction

In this deliverable, we outline the methodology for automated quality assurance and analysis for extra-functional properties (e.g., resource consumption, security, performance) based on the model specification of the corresponding requirements that target real industrial continuous development challenges. In WP3, we develop automated test generation and analysis techniques for functional and extra-functional properties (e.g., resource consumption, security, performance). We build our generation algorithms on the model specification of the corresponding requirements developed in WP1 and WP2 and results from testing and operations of earlier deltas. We develop techniques for automated test generation, model-based variant testing, automated static and dynamic analysis, consistency checking, and verification of the feature variant artifacts. Towards this goal, we develop techniques that can be used in the selection, generation, and execution of test cases, exercising the evolving systems' behavior concerning the supplied quality characteristics and testing for resource consumption given specific system requirements.

This WP3 methodology aims to maximize automation in ensuring the satisfaction of specified non-functional requirements in the CI/CD context for each delta version. The creation of comprehensive guidelines for the adopters of SmartDelta quality assurance methods and tools is essential. The methodology will also serve as teaching material for IT-oriented universities. This methodology relates to the WP3 tasks: T3.1 Delta-oriented Test Case Generation and Selection for Functional Requirements, T3.2 Delta-oriented Test Case Generation and Selection for Extra-Functional Quality Requirements, T3.3 Static Analysis of Quality Attributes Evolution, and T3.4 Delta Oriented Regression Testing in Continuous integration.

The methodology includes a suite of techniques addressing various aspects of delta-oriented quality assurance in industrial software systems. By focusing on the unique characteristics of each delta, the methodology ensures that the quality assurance processes are both efficient and effective using the following methods and techniques:

- Delta-Specific Analysis and Prioritization. Techniques are developed to identify and prioritize meaningful changes in each delta, enabling targeted quality assurance efforts.
- Test Case Design and Optimization. While test generation and selection are important, the methodology extends to generating test cases that address functional, non-functional, and extra-functional requirements such as performance, security, and resource consumption to improve subsequent tests for future deltas.
- Static and Dynamic Analysis. Static analysis techniques are introduced to address vulnerabilities, code smells, and maintainability concerns, ensuring that non-functional requirements are consistently met across iterations. Dynamic analysis techniques focus on operational behaviors to identify anomalies and optimize system performance over successive increments.
- Variant and Configuration Management. For highly configurable software, the methodology provides approaches to manage the complexity of variant testing, prioritizing and selecting test cases from previous product iterations.

This approach ensures that delta-oriented quality assurance is not limited to testing but also includes software artifact analysis, risk management, and optimization techniques to address the needs of industrial software development in a continuous integration context.

Section 2 reviews the state of the art in model-based testing, regression test selection, test co-evolution, and test reuse, identifying gaps that the SmartDelta methodology seeks to address.

© 2024 SmartDelta Consortium

Section 3 provides a detailed description of the tools and techniques developed within WP3, highlighting their purpose, features, and applications. Section 4 introduces the methodology for delta-aware quality assurance, focusing on how delta-specific changes are identified and tested while integrating various tools and techniques. Section 5 contextualizes these contributions by situating them within the broader SmartDelta methodology, showing how the tools address specific use cases. Section 6 summarizes several performance metrics, providing an overview of the tools' effectiveness and contributions to enhancing quality assurance processes. Finally, the document concludes in Section 7 with the outcomes, reflecting on the contributions of WP3 and showing some recommendations for future work.

2. State of the Art

In this section, we outline the state of the art in delta-oriented quality assurance, including requirement management and analysis, product line testing, regression testing, test amplification, and test reuse.

2.1. Model-Based Testing of Industrial Product Lines

Test modeling and test case generation focus on inter- and intra-product lines. The context of versions of systems related by common features and a shared development history that opens new frontiers for model-based testing, such as avoiding retesting the same feature or evaluating how different combinations affect the performance of individual features, become pertinent considerations. Moreover, there may be advantages to adopting variability modeling and test generation methods that work on the same artifacts. Industrial software can be highly configured during continuous development and varying feature selection. McGregor[1] studied the construction of software families as one aspect of configurability. How to handle variability testing in PLs has been surveyed in the literature; however, it needs evaluation on industrial systems[2],[3],[4].

Model-Based Testing (MBT)[5] for functional and non-functional properties uses test-ready models as input for automated test generation and execution. There are several benefits to the use of MBT[6], including easier test maintenance, higher test coverage, and documentation of the behavior in models. It is necessary to model the variability in the intended PL to employ MBT in highly configurable systems. On a conceptual level, there are at least two different modeling languages

[1] McGregor, J. D., Muthig, D., Yoshimura, K. and Jensen, P. [2010]. Successful software product line practices,IEEEsoftware27(3): 16–21

[2] Sinnema, M. and Deelstra, S. [2007]. Classifying variability modeling techniques, Information and Software Technology 49(7): 717–739.

[3] Lamancha, B. P., Usaola, M. P. and de Guzman, I. G. R. [2009]. Model-driven testing in software product lines, International Conference on Software Maintenance, IEEE, pp. 511–514.

[4] Lee, J., Kang, S. and Lee, D. [2012]. A survey on software product line testing, International Software Product Line Conference, ACM, pp. 31–40

[5] Utting, Mark, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches." Software testing, verification and reliability 22.5 (2012): 297-312.

[6] Pretschner, Alexander, et al. "One evaluation of model-based testing and its automation." Proceedings of the 27th international conference on Software engineering. 2005.

© 2024 SmartDelta Consortium

for testing evolving systems: Feature Modelling (FM) and Orthogonal Variability Modelling (OVM)[7] and feature-based and delta-oriented models for testing, including product selection techniques[8] [9].

2.2. Maintaining/Improving the Quality of Regression Tests

Regression testing gives developers confidence that existing functionality works as intended throughout the software evolution. This process depends on the quality of both test cases and the testing process itself. Recent studies have shown that automated regression testing is susceptible to *flakiness*[10] [11] [12] [13]. Tests are expected to evolve with the software to cover new or changing requirements. During this process, design or coding issues may introduce technical debt in the form of test code smells. In a recent literature review, Aljedaani et al. identified 22 tools covering 66 *test smells* [14]. Such tools allow developers to get an overview of the existing issues in their test suites.

Regression testing is often expensive. The regression test suites can be extremely large and thus time-consuming to execute. This is particularly true for system testing, where the test suites can grow very large, even for simple coverage criteria. This means re-running every test case in the test suite will be expensive. Therefore, regression test selection techniques assume an important role in the overall regression test strategy, even more so in the rapid development processes of CI/CD.

2.3. Property-based testing for non-functional requirements

Property-based Testing (PBT) [15] focuses on the inputs and outputs of a System-Under-Test (SUT), where the SUT is usually stateless. Testers define the general structure of valid inputs for the system under test (SUT) and a set of properties representing the expected behavior for these inputs. Based on this structure, a wide range of increasingly complex random inputs is generated and applied to the SUT. The system's responses are then monitored to verify that the outputs conform

[7] F. Roos-Frantz, D. Benavides, and A. Ruiz-Cortes, “Feature Model to Orthogonal Variability Model Transformation towards Interoperability between Tools,” p. 9.

[8] V. Hafemann Fragal, A. Simão, M.R. Mousavi, and U.C. Türker. Extending HSI Test Generation Method for Software Product Lines. Computer Journal, 62(1): 109-129, 2019.

[9] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M.R. Mousavi and I. Schaefer. A Classification of Product Sampling for Software Product Lines. Proceedings of 22nd International Systems and Software Product Line Conference. ACM, 2018.

[10] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests", Proceedings of the 40th International Conference on Software Engineering, 2018.

[11] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude and A. Bertolino. "What is the Vocabulary of Flaky Tests?“, Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). Association for Computing Machinery, New York, NY, USA, 492–502, 2020, DOI:https://doi.org/10.1145/3379597.3387482

[12] A. Shi, W. Lam, R. Oei, T. Xie and D. Marinov, "iFixFlakies: a framework for automatically fixing order-dependent flaky tests", Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019, 2019. doi:10.1145/3338906.3338925

[13] W. Lam, K. Muşlu, H. Sajnani and S. Thummalapenta, "A study on the lifecycle of flaky tests", Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. doi:10.1145/3377811.3381749

[14] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C.D. Newman, A. Ghallab, and S. Ludi. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In Evaluation and Assessment in Software Engineering (EASE 2021). Association for Computing Machinery, New York, NY, USA, 170–180. https://doi.org/10.1145/3463274.3463335

[15] D. R. MacIver, “In praise of property-based testing.” https://increment.com/testing/in-praise-of-property-based-testing/, 2019. [Online; acessed 25-March-2022].

© 2024 SmartDelta Consortium

to the specified properties, ensuring the SUT behaves as anticipated across various inputs. Thus, testers verify the SUT on families of cases. This approach was first popularized through the QuickCheck library in Haskell[16] and later found wider adoption in other programming communities such as Erlang[17] and Python[18]. The complexity of property-based testing lies in ergonomic specification languages[19], meaningful exploration of the input space[20,] and reduction of the faulty examples to their minimal form for better readability and easier debugging[21]. Property-based testing has also been applied to non-functional testing. For instance, it has been applied to testing non-functional requirements of web services[22].

2.4. Test co-evolution and repair

The evolution of software systems due to fault fixes, functionality modification, or source code refactoring may cause previous test cases to become obsolete and non-executable. Due to the high-cost maintenance and time consumption of previous test cases, software developers often discard them. TestCareAssistant is another technique for automated repairing of obsolete test cases, combining data-flow analysis with program diffing. The technique analyses the impact of method declaration changes on the executability of test cases to repair the test compilation errors [23]. Furthermore, B. Daniel et al. proposed a tool, ReAssert, to automatically repair unit tests[24]. The purpose of the ReAssert tool is to suggest repairs for the broken tests, and if the user agrees with the suggested changes, then the tool modifies the test code in an automated fashion. However, due to the tool's limitations for only lack of complex control flows or operations on expected values, B. Daniel et al. propose another technique based on symbolic execution, named symbolic test repair[25]. The technique is based on repairing test cases by changing the literal in the test code and integrating it into the ReAssert general test repair process. Another tool, ScrIpT repAireR (SITAR), proposed by Z. Gao et al., automatically evolves unusable low-level test scripts for GUI-based applications[26]. The technique behind SITAR is to utilize human inputs and repair transformations to create an abstract test for each test script, mapping it to an annotated event-flow graph (EFG) to create a new synthesized repaired test script. During the process, SITAR also repairs the GUI

[16] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," ACM SIGPLAN Notices, vol. 46, no. 4, 2000.

[17] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in Proceedings of the ACM SIGPLAN Workshop on Erlang, 2006.

[18] D. R. MacIver, A. F. Donaldson, and many other contributors, "Hypothesis: A new approach to property-based testing," Journal of OpenSource Software, vol. 4, no. 43, 2019.

[19] M. Gligorić, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in IEEE International Conference on Software Engineering (ICSE), 2010.

[20] A. Löscher and K. Sagonas, "Targeted property-based testing," in 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017.

[21] D. R. MacIver and A. F. Donaldson, "Test-case reduction via test case generation: Insights from the hypothesis reducer," in European Conference on Object-Oriented Programming (ECOOP), 2020.

[22] Bai, Yu, et al. "A non-functional property based service selection and service verification model." *International Conference on Ubiquitous Intelligence and Computing*. Springer, Berlin, Heidelberg, 2011.

[23] M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM 2010), pages 1–5. IEEE, 2010.

[24] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In ASE, 2009. http://mir.cs.illinois.edu/reassert/ .

[25] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, page 207218, New York, NY, USA, 2010. ACM.

[26] Z. Gao, Z. Chen, Y. Zou, SITAR: GUI test script repair, IEEE Trans. Softw. Eng. 42 (2016) 170–186 .

objects used in the checkpoints, which can be executed automatically to validate the revised software. Additionally, Javaria et al. proposed an automated model-based approach to repair the test cases based on the DOM (Domain Object Model) strategy to maintain the same DOM coverage and fault-finding capability for evolving web applications[27]. The purpose of the approach is to generate repaired test scripts for an evolved web version by taking two different versions of a web application and a test suite of an earlier version as input.

In the automotive testing domain, with the increase of Simulink modeling and the use of behavioral models, a major concern is the evolution of software tests and their maintenance. To address this concern, E.J. Rapos et al. proposed a toolset, SimEvo, to assist Simulink developers in test evolution alongside their source models[28]. The proposed tool consists of an existing impact analysis tool; SimPact [29], and a new tool; SimTH, dedicated to the automated generation and adaptation of Simulink test harnesses. SimTH requires only a Simulink behavioral model as input to generate the test harness for an automotive model. The main tasks to create the automated functional test harness are; the inclusion of the system under test (SUT) components, input and out management to the SUT, environment simulation through simulating standard calls and their accurate response to the simulated environment, and standardization of the generated test harness models. The inclusion of the SimPact tool is to maintain evolving Simulink models due to the changes made to models on test cases and harnesses. SimPact identifies and mitigates the impact of these changes to guide the use of SimTH.

In SmartDelta, we are improving the state of the art by focusing on co-evolution and repairing system integration test cases. Based on the analysis of the changes, we are using impact analysis for functional and extra-functional properties.

2.5. Test reuse across products

One of the main obstructions to quality assurance of software systems is the effort required to test the system rigorously. Reusing software artifacts can help enhance the quality and productivity of testing software systems. Software test reuse can be applied to numerous artifacts, i.e., built-in tests, model-based testing, test environments, and domain-based testing, to reduce development time and enhance the quality of software systems [30].

To construct reusable tests for software product lines (SPL), which share common characteristics, Reuys et al. proposed a Scenario-based Test case Derivation (ScenTED) model-based technique to ensure the reuse of test cases through preservation of variability during test case derivation, as is the case during the domain modeling process to derive software product families[31]. In addition, Bucaioni et al. leverage model-driven engineering to derive test scripts for product variants within

[27] Javaria Imtiaz, Muhammad Zohaib Iqbal, & Muhammad Uzair khan (2021). An automated model-based approach to repair test suites of evolving web applications. *Journal of Systems and Software, 171, 110841.*
[28] E. Rapos and J. Cordy, "SimEvo: A Toolset for Simulink Test Evolution & Maintenance," in 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), Västerås, Sweden, 2018 pp. 410-415.
[29] E.J. Rapos and J.R. Cordy. SimPact: Impact analysis for simulink models. In Proceedings of the 33rd International Conference on Software Maintenance and Evolution, ICSME ’17. IEEE Press, 2017.
[30] Rajeev Tiwari and Noopur Goel. 2013. Reuse: reducing test effort. SIGSOFT Softw. Eng. Notes 38, 2 (March 2013), 1–11.
[31] Reuys, A., Kamsties, E., Pohl, K. and Reis, S. 2005, Model Based System Testing of Software Product Families, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2005, Volume 3520.

© 2024 SmartDelta Consortium

the product family[32]. Results from their evaluation show that the approach reduced development effort while overcoming consistency drawbacks. Furthermore, Olimpiew et al. proposed a feature-oriented model-based testing method for SPLs for automated selection and configuration of reusable test specifications. In feature-based test derivation, the proposed method selects several reusable test specifications to test and configure a given application derived from SPL[33].

Test reuse approaches for GUI applications transform human-designed GUI tests from one app to another, which shares similar functionalities through semantic similarity among textual information of GUI widgets [34]. To evaluate such approaches, Zhao et al. proposed a framework, FrUITeR, which automatically validates UI test reuse cases based on evaluation metrics[35]. The metrics determine the correctness of mapped GUI events from a source to a target app and the usefulness of transferred tests in practice. Moreover, G. Hu et al. proposed AppFlow for synthesizing highly robust, reusable UI tests utilizing machine learning[36]. AppFlow provides machine learning capabilities to automatically detect common UI screens and widgets, which allows developers to write modular tests for the main functionality of an app category. After detecting UI screens and widgets, AppFlow can quickly synthesize complete tests from the modular library to test a new app in the same category.

In SmartDelta, we are improving traditional reuse methods to the level of evolving industrial systems. If artifacts are reused, their testware can be reused based on analysis of the changes, using impact analysis for functional and extra-functional properties.

2.6. Change-based regression test selection

Automated test suites grow along with the software systems under test, often making it too costly to execute entire test suites. Runeson, for example, reported that some unit test suites take hours to run[37]. Consequently, the last decade has seen an increasing interest in *test selection*, seeking to

---

[32] Bucaioni, A., Di Silvestro, F., Singh, I., Saadatmand, M., & Muccini, H. (2022). Model-based generation of test scripts across product variants: An experience report from the railway industry. Journal of Software: Evolution and Process, e2498.
[33] Olimpiew, E. M. and Gomaa, H. 2009. Reusable Model-based Testing, ICSR 09: Proceedings of the 11th International Conference on Software Reuse, pages 76-85, Berlin, Heidelberg, 2009, Springer-Verlag. Doi:10.1007/978-3-642-04211-9_8
[34] Mariani, Leonardo et al. "Semantic Matching of GUI Events for Test Reuse: Are We There Yet?." *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2021.
[35] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: a framework for evaluating UI test reuse. In Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE 20).
[36] Hu, G., Zhu, L., & Yang, J. (2018). AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 269–282). ACM.

[37] P.Runeson,“A survey of unit testing practices,”IEEE Software,vol.23, no. 4, pp. 22–29, 2006.

© 2024 SmartDelta Consortium

identify those test cases relevant to the most recent changes. Since 2008, five literature surveys have been published, illustrating the vast body of knowledge available on the topic [38], [39], [40], [41], [42].

Traceability links. During test selection, a software analysis tool constructs a test mapping table, maintaining traceability links between the code under test and the corresponding test cases. Researchers have investigated numerous heuristics for recovering these traceability links: control flow analysis, data flow analysis, firewalls, slicing, and clustering. Recently, people also incorporated artificial intelligence techniques like machine learning[43] and game theory[44]. Others exploited information from artifacts beyond the source code, such as defect tracking systems[45] and test logs[46], [47]. Yet, for tool builder purposes, all of these heuristics (as shown in Table 1) can be roughly classified according to two dimensions: the way they collect the traceability links and the granularity at which they do so[48].

Table 1: Dimensions for Classifying Test Selection Heuristics

Collecting: Static vs. Dynamic
• *Static approaches* extract the traceability links from the source code or some representation of it. • *Dynamic approaches* derive the traceability links from execution traces recorded during actual test runs.

[38] E. Engström, M. Skoglund, and P. Runeson, "Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review," in Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. ACM, 2008, pp. 22–31.

[39] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[40] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.430

[41] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran, "Regression test selection techniques: A survey," Informatica (03505596), vol. 35, no. 3, 2011.

[42] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," ACM Computing Surveys, vol. 50, no. 2, pp. 29:1–29:32, May 2017.

[43] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in Proceedings ISSTA 2017 (the 26th ACM SIGSOFT InternationalSymposiumonSoftwareTestingandAnalysis). NewYork, NY, USA: ACM, 2017, pp. 12–22.

[44] N. Kukreja, W. G. J. Halfond, and M. Tambe, "Randomizing regression tests using game theory," in Proceedings ASE2013 (the 28th IEEE/ACM International Conference on Automated Software Engineering). Piscat- away, NJ, USA: IEEE Press, 2013, pp. 616–621.

[45] E.Engström, P.Runeson, andG.Wikstrand, "An empirical evaluation of regression testing based on fix-cache recommendations," in Proceedings ICST2010 (Third International Conference on Software Testing, Verification and Validation). Piscataway, NJ, USA: IEEE Press, April 2010, pp. 75–78.

[46] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in Pro- ceedings ICSE '02 (the 24th International Conference on Software Engineering). New York, NY, USA: ACM, 2002, pp. 119–129.

[47] E. Ekelund and E. Engström, "Efficient regression testing based on test history: An industrial evaluation," in Proceedings ICSME2015 (IEEE International Conference on Software Maintenance and Evolution). Piscataway, NJ, USA: IEEE Press, Sept 2015, pp. 449–457.

[48] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in Proceedings FSE 2016 (the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering). New York, NY, USA: ACM, 2016, pp. 583–594.

Granularity: Fine-grained vs. Coarse-grained
The level of detail chosen for the elements in the mapping table. ● method ←→ test method ● class ←→ unit test ● file ←→ test case ● . . .

Continuous Integration. However, all of the heuristics are designed to reduce large regression test suites that run every so often on a test execution environment with plenty of resources. However, with the introduction of continuous integration, the context of the test selection problem shifted entirely. Test engineers within the Firefox project reported that there needs more time for comprehensive testing due to the rapid release cycles, forcing test teams to narrow their scope and perform deeper testing in selected areas[49].

Microsoft asserted that the change rate and the scale of their code base demands ``*[...] a fast, simple algorithm that works well in practice and does not attempt an expensive data flow algorithm to determine which new code will be covered by which existing tests*'' [50]. In a later paper, Microsoft argued that due to the complex branch dependencies, it is impossible to run *all* of the tests for *all* of the changes across *all* configurations: to avoid heavy-weight dependency analysis, they exploit historic test execution data at component level[51]. Regarding the classification in Table 1, they adopt a *dynamic* and *coarse-grained* test selection heuristic.

In the same vein, Google acknowledged that "*In continuous integration, however, testing requests arrive at frequent intervals, rendering techniques that require significant analysis time overly expensive, and rendering techniques that must be applied to complete sets of test cases overly constrained. Furthermore, the degree of code churn caused by continuous integration quickly renders data gathered by code instrumentation imprecise or even obsolete.*" [52] There is also a *dynamic* and *coarse-grained* test selection heuristic that allows scaling up the regression testing selection[53]. Not surprisingly, open-source projects followed suit; some projects in the Apache Foundation [54 adopted the Ekstazi tool]. A report on a test selection approach is also applied to WebKit[55].

---

[49] M. Mäntylä, F. Khomh, B. Adams, E. Engström, and K. Petersen, “On rapid releases and software testing,” in 2013 IEEE International Conference on Software Maintenance. Piscataway, NJ, USA: IEEE Press, Sept 2013, pp. 20–29.

[50] A. Srivastava and J. Thiagarajan, “Effectively prioritizing tests in de- velopment environment,” in Proceedings ISSTA ’02 (the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis). New York, NY, USA: ACM, 2002, pp. 97–106.

[51] K.Herzig, M.Greiler, J.Czerwonka, and B.Murphy, “The art of testing less without sacrificing quality,” in Proceedings ICSE ’15 (the 37th International Conference on Software Engineering). Piscataway, NJ, USA: IEEE Press, 2015, pp. 483–493.

[52] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments,” in Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 235–245.

[53] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, “Taming google-scale continuous testing,” in Proceedings ICSE-SEIP ’17 (the 39th International Conference on Software Engi- neering: Software Engineering in Practice Track). Piscataway, NJ, USA: IEEE Press, 2017, pp. 233–242.

[54] M. Gligoric, L. Eloussi, and D. Marinov, “Practical regression test selection with dynamic file dependencies,” in Proceedings ISSTA 2015 (the 2015 International Symposium on Software Testing and Analysis). New York, NY, USA: ACM, 2015, pp. 211–222.

[55] J. Jasz, L. Lango, T. Gyimothy, T. Gergely, A. Beszedes, and L. Schrettner, “Code coverage-based regression test selection and prioritization in WebKit,” in Proceedings ICSM ’12 the 2012 IEEE International Conference on Software

Again, to scale up, Ekstazi and the WebKit approach settled on a *dynamic* and *coarse-grained* test selection heuristic.

Test Suite Composition. A critical factor mostly neglected in evaluating test selection heuristics is the composition of the test suite itself. Indeed, the drive towards test automation forced many development teams to adopt one of the many *xUnit* test harnesses. While the names imply that these tools are meant for "unit testing'', there exists various flavors of what such a unit under test might be. Meszaros, for instance, stated that in modern testing, xUnit is used to exercise a system at three levels of granularity: system, component, and unit[56]. Massol, on the other hand, distinguishes between logic, integration, and functional unit tests[57]. Binder differentiates between method, class, and component tests[58]. Yet, the nature of the test selection problem changes considerably when the unit under test is fine-grained (a single method or a class), medium-grained (a component, a coherent set of classes), or coarse-grained (a system test, a functional test).

Summary. The need to select a subset of existing regression tests for execution to cope with time constraints is not new. Today's agile methods and development processes further underline the necessity of fast feedback after changes. Continuous integration tools provide the technical harness to execute regression tests regularly and often. However, executing all of the tests for all of the changes across all configurations is just infeasible. Ultimately, one needs a test selection heuristic that selects the tests based on the changes pushed into the continuous integration system. The heuristic should satisfy the following requirements:

- Fast: The selection heuristics should be ordered faster than executing the test suite.
- Reduce: the selection heuristic should reduce the test suite substantially.
- Test composition: the selection heuristic should differentiate between the various flavors of tests incorporated into the continuous integration system. (unit tests, component tests, system tests, functional tests, integration tests)
- Inclusiveness. Does the heuristic include all relevant tests? Are there false negatives?

In the following, we present a few of these methods for regression testing.

Full Regression Testing. Introduced as a comprehensive examination of all available test cases, Full Regression Testing is often employed in practice as a standard strategy, known as the "Retest-All" approach. Despite its thoroughness, it demands significant resources, potentially occupying developer capacities.

Informal Approaches. An ad-hoc, non-systematic approach where tests are executed as needed without specific planning and a random method (this strategy randomly selects a subset of tests for regression testing). Both methods are typically used when time constraints prevent a full regression test.

Test Case Prioritization (TCP). Aims to optimally order test cases for regression testing to detect critical errors early on. By prioritizing test cases, desirable properties such as high fault detection rate, code coverage, and brief execution times are emphasized. Common methods include the Coverage-based, History-based, and Requirement-based approaches.

Maintenance (ICSM). Washington, DC, USA: IEEE Computer Society, 2012, pp. 46–55. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2012.6405252

[56] G. Meszaros, xUnit Test Patterns. Addison-Wesley, 2007.

[57] V. Massol, JUnit In Action. Manning Publications Co., 2004.

[58] R. V. Binder, Testing Object-oriented Systems: Models, Patterns, and Tools. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

© 2024 SmartDelta Consortium

Test Suite Minimization (TSM). Also referred to as Test Suite Reduction, the goal of TSM is to decrease the number of tests within a test suite. It focuses on reducing obsolete and redundant test cases to save time.

Test Case Selection (TCS). TCS encompasses targeted test case selection from a larger pool. Both automated (algorithmic) and manual (knowledge-based) approaches exist. (Automated) TCS methods work as described below:

(1) Source Code Analysis: The significance and potential implications of code changes can be determined by examining source code comments and version history.
(2) Machine Learning and AI: Leveraging artificial intelligence to discern patterns in code alterations, allowing for predictions regarding future modifications. (also referred as Predictive Test Selection)
(3) Dynamic Binding: This method links source code and test code, adapting test cases to changes in the source code.
(4) Dependency Analysis: This technique focuses on identifying interdependencies between different segments of code and test cases, ensuring that pertinent tests are selected when code is altered.
(5) Change Impact Analysis (CIA) is an essential tool for detecting effects caused by code changes. It systematically evaluates possible side effects of modifications. Within the CIA, methodologies like dependency and traceability analysis are sometimes utilized.
(6) Traceability Analysis: Tracks the relationship between requirements or design documents and test cases, ensuring that changes in one area are properly reflected and tested in another.

The Predictive Test Selection (PTS) is emerging within the TCS area, which is gaining traction with advancements in machine learning.

2.7. Mutation Testing (for Simulink)

Mutation testing is the state-of-the-art technique for evaluating the fault-detection capability of a test suite. The technique deliberately injects faults into the code and counts how many of them the test suite catches. While the principle applies to any automated test suite, it is mainly used for unit and component tests.

A few early proof-of-concept tool prototypes illustrate how to apply mutation testing in the context of SIMULINK. Simulink models the system under test using blocks that transform input signals into output signals, modeling the behavior of the cyber-physical system under implementation. Engineers simulate the behavior in the Simulink environment, manually verifying whether the final output signal matches the expected outcome. Once the model satisfies the requirements, code is generated and deployed on a real-time embedded platform. Today, Simulink has two different test execution environments – Signal Builder and Simulink Test. In both these environments, facilities are available to assert whether the output signal is the same as the expected baseline signal, as such imitating the fully automated xUnit behavior of passing (green) or failing (red) tests.

Today, there is little research on mutation testing for Simulink models, as witnessed by a recent systematic literature survey conducted by Papadakis et al.[59] This can be expected because Simulink is very specific to particular application domains, such as automotive and avionics. However, those areas require a high degree of reliability and trust, and their development is based on stringent safety standards. So, one may expect mutation testing to eventually be adopted in this context as

[59] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation Testing Advances: An Analysis and Survey," in Advances in Computers. Elsevier, 2019, vol. 112, pp. 275–378. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015

© 2024 SmartDelta Consortium

well. Based on a literature survey, we identified three relevant papers, all on technology readiness levels: 1 (technology concept formulated) or 2 (experimental proof of concept).

The first report we found was a paper by Hanh and Bihn that specified six categories of mutation operators varying from type to expression operator [60]. These mutation operators are defined based on the grammar of Simulink (mutating signals and blocks) and as such are quite comprehensive. No analysis is based on a fault model of frequently occurring faults in realistic applications. The paper reports on applying these models to a quadratic model and reports mutant execution results. No replication packages are provided. A second report concerns the SIMULTATE tool introduced to automate mutant generation for Simulink models[61]. The tool employs a Python API to inject the faults into model blocks using an interactive user interface. For the mutant generation, SIMULTATE injects all mutants in a single pass, but each mutant should be enabled or disabled individually. Compared to the earlier work of Hanh and Bihn, SIMULTATE supports mutating signals only. There is no automated test execution; engineers are expected to manually verify whether the mutated output signal differs from the original one. FIM is the third proof-of-concept tool we have identified[62]. FIM inherits mutation operators of SIMULATE tool and supports ten operators for signals and five operators for blocks. The tool mainly focuses on safety analysis via extensive fault injection capabilities. The tool provides single and multi-mode options for generating a single model with multiple faults or multiple models with a single fault. It uses the MATLAB command line interface to trigger the tool and a custom library from which the block can be changed easily. As such, the mutant generation is faster than SIMULTATE as it does not trigger the MATLAB user interface. FIM is evaluated with an Aircraft Elevator Control System (AECS) from the avionics-aerospace domain. The paper only reports the generation of mutants, not their execution.

2.8. Reflections and Identified Gaps

The review of state-of-the-art approaches in delta-oriented quality assurance highlights some advancements, particularly in testing, regression strategies, and anomaly detection. However, several challenges still need to be solved in the scalability of current methods for highly configurable systems. Furthermore, operational data integration into quality assurance processes remains limited, restricting testing and analysis potential. Despite progress, anomaly detection still struggles with accuracy and efficiency, particularly in fault localization within dynamic environments.

To bridge these gaps, SmartDelta integrates feedback loops from operational and development data, which enables quality assurance processes to continuously adapt and evolve their techniques in response to quality attributes. Furthermore, advancing automation techniques for test reuse, generation, and augmentation ensures that tests evolve alongside the systems they evaluate and reduce manual intervention.

[60] L.T.M.Hanh and N.T.Binh, “Mutation operators for simulink models,” in 2012 Fourth International Conference on Knowledge and Systems Engineering. Danang, Vietnam: IEEE, 2012, pp. 54–59.

[61] I. Pill, I. Rubil, F. Wotawa, and M. Nica, “Simultate: A toolset for fault injection and mutation testing of simulink models,” in 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). Chicago, IL, USA: IEEE, 2016, pp. 168–173.

[62] E. Bartocci, L. Mariani, D. Nicˇkovic´, and D. Yadav, “Fim: Fault injection and mutation for simulink,” in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1716–1720. [Online]. Available: https://doi.org/10.1145/3540250.3558932

© 2024 SmartDelta Consortium

3. SmartDelta Methods for Quality Assurance

This toolset contains toolkits for automated generation and checking of model specifications for feature variant artifacts by bringing novel techniques supporting automated verification of product deltas through model specifications at development and runtime. In addition, a toolset for automated reuse analysis and recommendation of artifacts for each delta version was developed.

3.1. Combinatorial Test Generation (SEAFOX)

Method Description

SEAFOX is a combinatorial testing tool that was developed at Mälardalen University. This tool can take a PLCopen XML file as input, which contains information about the program to be tested. Then, the tool parses the file by extracting the needed information that will be used to generate test cases, such as the name of the input parameters and their respective data types. Another option is to manually provide the parameters and data types in the tool. Further, the SEAFOX tool creates test cases by generating the inputs using one of the available algorithms implemented in the tool: Random, Base choice, or Pairwise algorithm. These input values can then be exported as comma-separated values in a CSV file where each line is a new set of input parameters representing a certain test case.

Improvements

Combinatorial test generation can be used to augment manual tests in CI environments where testing cycles are short. When SEAFOX is used, automatically generated test cases could be added to existing regression suites and be run within continuous integration. For this purpose, we plan to integrate into build environments (e.g., Maven), such that new test suites are re-generated and selected with each new build of a system.

Purpose and Features

SEAFOX is a combinatorial testing tool specifically developed for IEC 61131-3 compliant PLC software. Its primary purpose is to automate the generation of test cases for industrial control systems. By using combinatorial testing techniques, SEAFOX aims to enhance the efficiency and effectiveness of testing processes.

Features of SEAFOX:

- Supports various combinatorial testing strategies, including pairwise, base choice, and random testing.
- Designed to work with PLC software development environments, particularly those that follow the IEC 61131-3 standard.
- Provides a graphical user interface for loading programs, defining input parameters, and generating test cases.
- Can import PLC programs in the PLCopen XML format, facilitating seamless integration with development environments like CODESYS.
- Users can manually enter parameter ranges and constraints for more tailored test case generation.

Usage Instructions

- Start SEAFOX and load the desired PLC program or function block stored in an XML file.

- Enter the input parameter names, data types, and ranges manually or import them from the XML file. Choose the combinatorial testing strategy (pairwise, base choice, or random) from the provided options.
- Specify the values each parameter can take. This can be single values, closed intervals, or a combination of both. Generate test cases by clicking on the generate button to create the test cases based on the chosen strategy.
- Export test cases by saving the generated test cases as a CSV file for further use or analysis. Run test cases by importing the test cases into the target PLC development environment (e.g., CODESYS) and execute them to validate the software.

Case Studies

SEAFOX was used to generate test suites for a train control management system (TCMS) developed by Alstom. The system contained 45 programs with an average of 1076 lines of XML code. SEAFOX's pairwise testing was compared with manual testing by industrial engineers, demonstrating that it achieved high code coverage and fault detection rates comparable to manually crafted test suites. In another study, SEAFOX was integrated with CODESYS IDE to facilitate automated testing of PLC programs. The generated test cases achieved 94% decision coverage on average, with random testing achieving the highest coverage at 98.15%. This integration streamlined the testing process and enhanced the reliability of industrial PLC software.

3.2. Requirement and Test Specification Static Checker (NALABS)

Method Description

In many embedded system domains, requirement engineering is performed manually by engineers who hand-craft those artifacts using natural or semi-structured languages and use them for other stages of software development, such as testing them against the system under test. Issues in natural language artifacts (i.e., requirements), such as ambiguities or vague specifications, can lead to higher costs during system development.

Previous research found that test specifications written in natural language contain a rather high degree of cloning and bad structure, which can influence the cost of maintaining and executing test cases. For requirements as well as for system test cases written in natural language, so-called bad smells have been established as indicators to identify poorly written natural language artifacts. Based on several natural language smells observed in previous studies, we established a set of indicators for requirement flaws and defined dictionary-based metrics to automatically detect these smells in natural language artifacts.

NALABS is a new tool that aims at quick analysis of requirements by detecting problematic requirements. The aim of NALABS is to help in bringing more evidence on the industrial use of bad smells for detecting specification quality defects.

Since requirement checking can be very costly and time-consuming, it is important to automate the verification of requirements using predefined rules. For CI environments where development cycles are very short, NALABS is used within continuous integration early on in the requirement engineering phases. For this purpose, we plan to integrate into requirement management environments (e.g., DOORS) but also use GitHub Actions to automate this workflow, such that new requirements and test specifications are checked with each new build of the architecture and design documents.

© 2024 SmartDelta Consortium

Purpose and Features

NALABS takes a set of spreadsheets that contain requirements in ID-description pairs as input. The tool outputs a list of metrics and quality scores for every requirement. Additionally, poorly chosen words are highlighted in the context for the user to review.

The set of checks run on the requirements are as follows:

- Number of Words (NW): Measures the length of each requirement.
- Number of Vague Phrases (NV): Detects ambiguous phrases.
- Number of Conjunctions (NC): Counts conjunctions such as "and" "or" "but"
- Number of Reference Documents (NRD1 and NRD2): Identifies external references.
- Optionality (OP): Flags optional terms like "may" or "could"
- Subjectivity (NS): Detects subjective language.
- Weakness (WK): Identifies weak verbs such as "support" or "process"
- Automated Readability Index (ARI): Calculates readability scores.
- Imperatives (NI1 and NI2): Counts imperative verbs and checks their context.
- Continuances (CT): Identifies terms like "etc" or "and so on"

Usage Instructions

NALABS has been implemented as a desktop application. It allows users to interact with the tool using a graphical user interface. The tool expects requirements to be written in an Excel document, with one requirement per row. Information identifying the requirement ID and specification columns must be provided to run the tool.

These settings are found under Edit/Settings in the menu. In the Excel view, you should then choose the REQ ID and Text column in the requirement Excel document. Documents are opened through the File menu. An example of the result view is shown in Figure 1. Metrics are displayed in separate columns. Smells found are highlighted and color-coded in the given context.

File Edit View

Id	Text	NW	NC	NV	Optionality	Subjectivity	NR	NR2	Weakness	Imperatives	Imperatives2	Continuances	ARI
2F0803-SRS/813	In case of Non -Priority Communication (other Cat Events): time between event trigger and Wayside confirmation of successful receipt shall be <= 30 mins	24	2	0	0	0	0	0	0	1	0	0	71.38
2F0803-SRS/812	The MCG grants the WCA maximum 50 seconds time to send the response back	14	0	0	0	0	0	0	0	0	0	0	51.93
2F0803-SRS/811	In case of Priority Communication (Cat A Events): time between event trigger and Wayside confirmation of successful receipt shall be <= 30 secs	23	2	0	0	0	0	0	0	1	0	0	69.35
2F0803-SRS/810	2F08 MCG Max Lifetime Unavailability Allowed = 3 telegram cycles	10	0	0	0	0	0	0	0	0	0	0	59.50
2F0803-SRS/799	GIVEN () WHEN (Request High Priority Communication = TRUE AND GSM Presence = TRUE) THEN (Monitor Wayside Communication) shall set Assign GSM Bearer = TRUE	26	3	0	0	0	0	0	0	1	0	0	68.00
2F0803-SRS/798	GIVEN WLAN presence = TRUE WHEN () THEN (Monitor Wayside Communication) shall set Assign WLAN Communication Bearer = TRUE	19	2	0	0	0	0	0	0	1	0	0	65.79
2F0803-SRS/797	GIVEN T2W File Transfer Status Response = FALSE WHEN Number of tentatives is greater than a configurable limit THEN (Execute Wayside Communication) shall set TDS Event - T2W File Upload Failed = TRUE	33	2	0	0	0	0	0	0	1	0	0	77.82

Figure 1: The GUI interface of NALABS.

NALABS is also qualified as a command line interface (CLI). The CLI variant supports both Excel and JSON files as inputs and outputs the result in JSON format. A working Python environment is needed to install and run the NALABS CLI. Given a fresh virtual environment, the prerequisites may be installed using pip install -r requirements.txt. The tool can then be run using the following

© 2024 SmartDelta Consortium

command: python NALABS.py <input_file> <id_column> <text_column> <output_file>. More information may be found at https://github.com/eduardenoiu/NALABS.

Case Studies

The NALABS tool evaluated 661 security requirements spanning 46 documents[63]. This analysis focused on identifying requirement smells or indicators of potential issues like vagueness, ambiguity, and excessive complexity, which can restrict the use of these security requirements. Using NALABS, the analysis concentrated on 12 smell indicators, including metrics such as number of conjunctions, vagueness, and Automated Readability Index, to assess the quality of NL security requirements. Specific indicators showed a statistically significant presence, suggesting these are common problem areas in security specifications. For instance, the ARI scores revealed a generally difficult to very difficult reading level across the analyzed requirements. The results also underscored nine moderate to strong correlations between different smell indicators, further showing patterns in the language of security requirements. For example, requirements exhibiting high counts in conjunctions often displayed increased complexity in logical structure, which could hide intended functionality and increase implementation risks.

3.3. Discovery of Interactions and Anomalies for Microservices (DIA4M)

Method Description

Related to the efficiency of DevOps engineers who manage the next-generation communication platform with cloud-based and microservice architectures (CPaaS – Communication Platform as a Service), we aim to increase the efficiency of the operations teams on the work/time unit. A monitoring and tracking system prepared using open-source Elasticsearch, Kibana, and Grafana products is currently used. It is provided to generate alarms at desired values by feeding them with the data taken from the platform.

As a result of the investigations, most of the time is spent by the operation teams monitoring the system status to keep the systems alive. The status of the number of ports used, which is of great importance for traffic, memory, processor, disk, and telecom systems, is monitored instantly. In addition, licenses that may expire are determined. Individual engineers carry out all these monitoring processes daily. The established central monitoring system ensures that the data of all the monitored systems are collected in a single place and that this data is correlated and alarmed at the desired values. In telecommunication platforms, it is necessary to monitor the system, take quick action against possible and observed errors, and ensure the continuous operability of the system. However, debugging and problem-addressing processes can take a long time in microservice architecture-based platforms with a high number of users and heavy traffic. This difficulty arises from the necessity of detecting microservice interactions. In the monitoring and anomaly detection category of the SmartDelta quality assurance dimension, we want to design a new tool called DIA4M that will map microservice interactions and locate anomaly and fault status faster.

Improvements

For CPaaS, a microservice architecture-based telecommunications platform with many users and heavy traffic, the error-adding process may vary depending on the system components. In the examinations, it has been observed that the operation teams spend the most time in the error-

[63] R. Hallberg, 'Finding Quality Problems In Security Requirements Using NALABS', Dissertation, 2023.

© 2024 SmartDelta Consortium

addressing process to eliminate the errors experienced in user scenarios, pointing to the routing and service components. In the tool, log patterns will be extracted from the microservice log data, and the interaction map of the mentioned microservice will be created by estimating the previous and next microservices with which the current microservice interacts at a particular moment. Data from the CPaaS platform can be used for interaction and topology discovery. The DIA4M tool will have features, as shown in Table 2.

Table 2. DIA4M tool features.

To-Do	Milestones	April 2024 Completion Rate	November 2024 Completion Rate
Interactive Node Based Mapping Visualizer	#1	% 100	% 100
Topology Based Exploratory Data Analysis through the Visualizer	#1	% 100	% 100
Feature Discovery (Filter logs by keywords, search etc.)	#1	% 100	% 100
Resource Trend Prediction (Trend analysis of microservice resources using ML methods)	#2	% 75	% 100
Service Summary (Fault detection and localization using ML methods)	#2	% 75	% 100
Categorical Comparison of Log Data (Revealing differences with previous versions)	#2	% 75	% 100
Elastic Cloud Integration	#3	% 50	% 100
Anomaly Detection Module (for both logs and Elastic agents)	#3	% 50	% 100
GPT Integration for troubleshooting	#3	% 50	% 100

Purpose and Features

During the QA phase of a system developed using microservice architecture, there are plenty of issues, bugs, regressions, and problems detected very late in the cycle. Throughout our DevOps process, we need to have a very clear vision of the situation, prioritize what we want to do, and know where we start. For this, we developed the DIA4M solution to struggle with microservice complexity and empower DevOps excellence.

Features of DIA4M:

- Interactive node-based mapping visualizer
- Topology-based exploratory data analysis through the visualizer
- Feature discovery (filter logs by keywords, search, etc.)
- Resource trend prediction (trend analysis of microservice resources using ML methods)
- Service summary (fault detection and localization using ML methods)
- Categorical comparison of log data (revealing differences with previous versions)
- Elastic cloud integration
- Anomaly detection module (for both logs and Elastic agents)
- GPT integration for troubleshooting

Usage Instructions

We have developed a new DI4M setup using Docker Compose. This new setup allows you to run DI4M APIs and databases with just one command, significantly simplifying the deployment and development process on different machines by leveraging Dockerized environments. Here is a comprehensive guide to setting up DI4M with Docker Compose (see repository view in Figure 2).

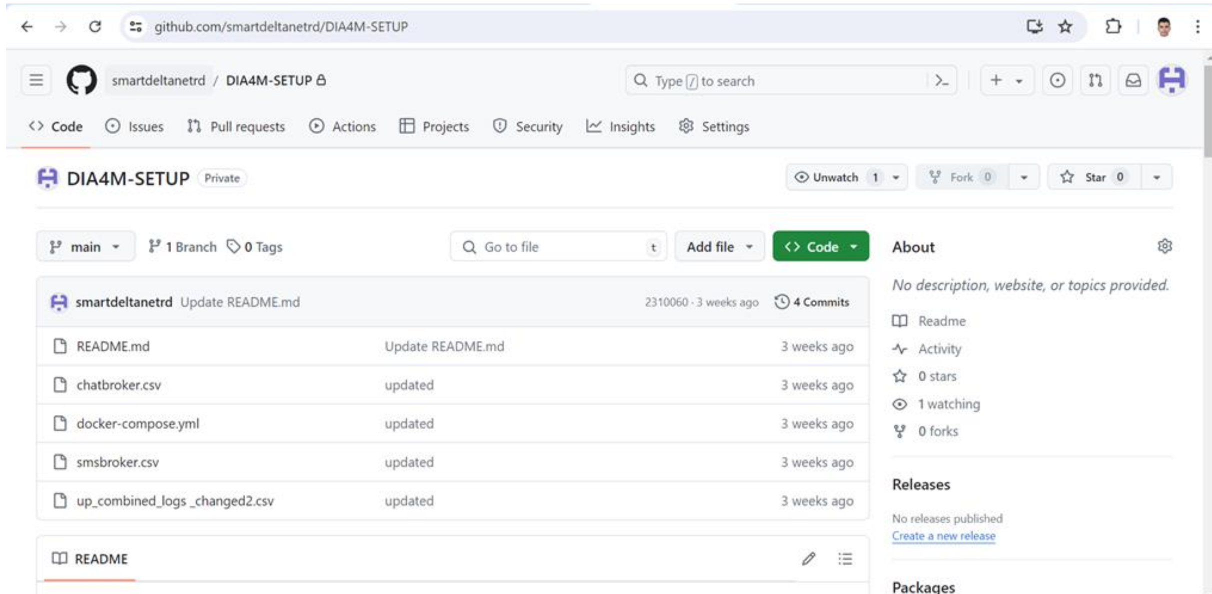


Figure 2: The DIA4M Repository.

To install Docker and Docker Compose, create a folder called SmartDelta: mkdir SmartDelta. Navigate to the SmartDelta folder: cd SmartDelta. Clone the repositories:

git clone -b dev-eren https://github.com/smartdeltanetrd/smartdelta-frontend.git
git clone -b dev-eren https://github.com/smartdeltanetrd/smartdelta-backend-ts.git

For each repo create a new branch(dev-yourName) and switch to your new branch for development. Download the docker-compose.yml file from the setup guide repository and move it to the root of SmartDelta. Run the following command in the root of the SmartDelta folder: docker compose up. Wait a few minutes for the containers to be created and started. If you get an error related to the mongo_db file permission run sudo chmod -R 755 /mongo_db on MACOS or Linux. If you use Windows, then run icacls C:\mongo_db /grant %username%:F /T icacls C:\mongo_db /grant Users:(RX) /T. Verify the setup: Navigate to http://localhost:5000. You should see the following JSON response: {"message":"Hello World"}. This indicates that the Node.js API is running without issues. Navigate to http://localhost:3000 to check if the front end is running without issues.

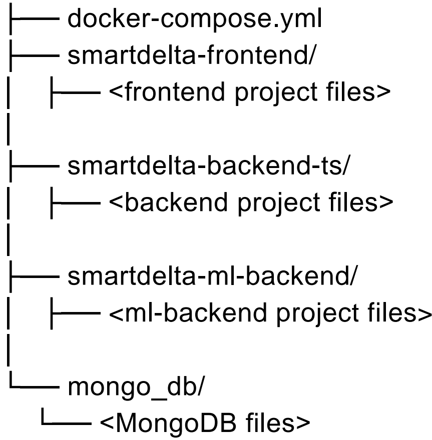
In the frontend app: Click on the file upload menu item and upload one of the CSV files in the repository. After uploading the file, you will be prompted to use the mapping visualizer. Go back to the file upload page to see the uploaded file list in the managed file part. If you see the uploaded file as a list item, it means the frontend, MongoDB, and Node.js API are running without issues.

Logs: To see the logs of the frontend app, open a terminal under the root of smartdelta-frontend and run docker logs -f smartdelta-frontend-1. To see the live changes of the Node.js API logs, open a terminal under the root of smartdelta-backend and run: docker logs -f smartdelta-api-1

Data Science Flask API Setup: Under the root of the SmartDelta folder, clone the smartdelta-ml-backend repository: git clone https://github.com/smartdeltanetrd/smartdelta-ml-backend.git

Follow the guideline on this link. **Directory Structure.** After completing the setup steps, your SmartDelta directory should look like this:

SmartDelta/



```
├── docker-compose.yml
├── smartdelta-frontend/
│   ├── <frontend project files>
│
├── smartdelta-backend-ts/
│   ├── <backend project files>
│
├── smartdelta-ml-backend/
│   ├── <ml-backend project files>
│
└── mongo_db/
    └── <MongoDB files>
```

In this version (v1.3) of the DIA4M tool, we are set to introduce key features and improvements, enhancing distributed analysis capabilities. Our primary focus is on optimizing the tool's functionality and adaptability within the context of the Smart Indoor Gardening IoT system—a senior project led by Eren Tarak, a developer of DIA4M and a senior student at the Department of Information Systems and Technologies, Bilkent University. Figure 3 shows the DIA4M frontend repository feature-based folder structure.

..	
analyse	Elastic Integration and Service Summary
compare	Elastic and gpt integration
elastic	Elastic and gpt integration
featureDiscovery	Elastic and gpt integration
k8	Cluster Integration Refactored
listingFiles	Elastic Integration and Service Summary
uploadFile	Resource Usage Sidebar and user tour added
userTours	Comparison dialog ui first phase done
visualize	Elastic Integration and Service Summary

DIA4M Node.js API Controllers

AnalysisController.ts	Elastic Integration routes and controller
AttachmentController.controller.ts	Feature Extraction, Model Training route developed
ElasticApmController.ts	Bug fixed
GptController.ts	Elastic apm and gpt integration
KubernetesController.ts	k8s strategy added

Figure 3: Frontend repositories for DIA4M.

© 2024 SmartDelta Consortium

Case Studies. Case Study 1: IoT-based Smart Systems and DIA4M

Smart Indoor Gardening IoT System is an on-demand scaling cloud-based AWS Lambda FaaS Node.js backend project designed to enhance individual indoor gardening experiences through real-time monitoring of temperature, soil moisture, light intensity, and air quality. The system's advanced plant leaf disease detection neural network model gives users insights, enabling proactive management and remote access to detailed updates. The high-level software architecture diagram of DIA4M is shown in Figure 4.

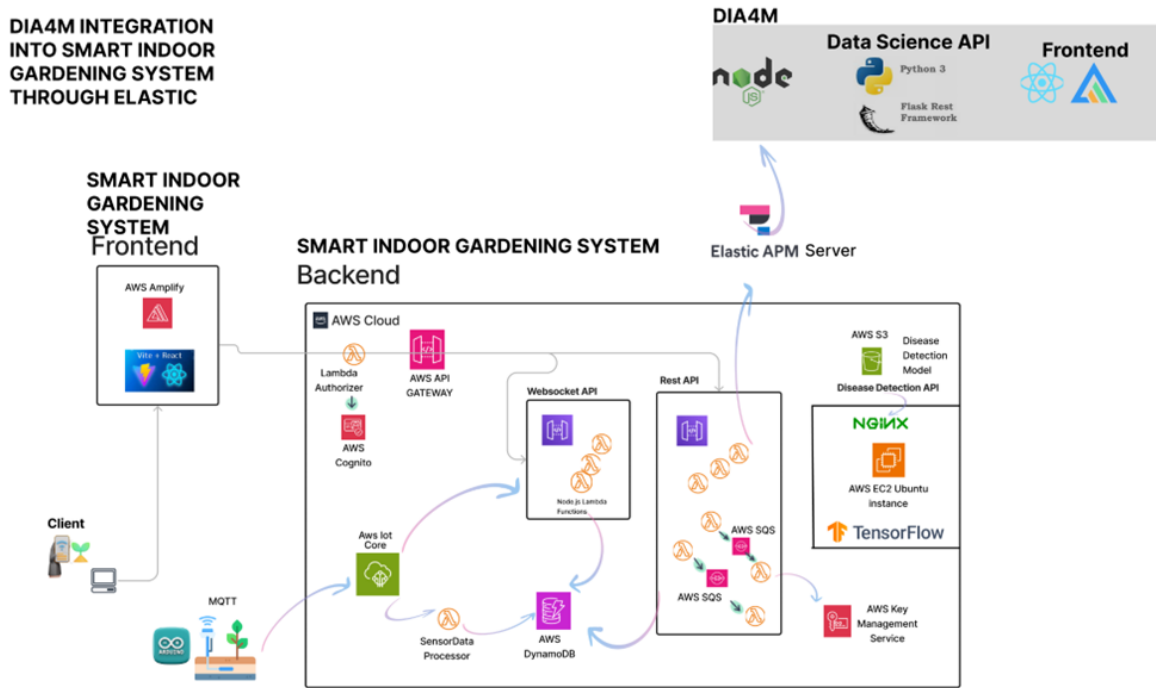
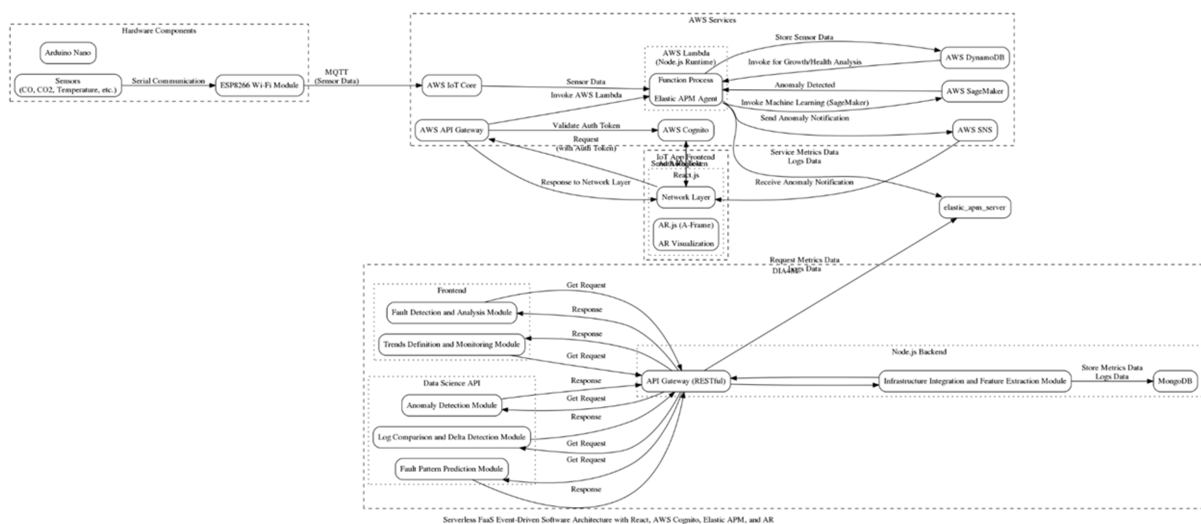


Figure 4: A Detailed Block Diagram of DIA4M.

The integration demonstrates DIA4M's adaptability and effectiveness in real-world scenarios, especially in an environment exposed to environmental sensor data. Performance monitoring and troubleshooting at both the service function and system levels are crucial to ensure the stability of intelligent systems (as for the architecture shown in Figure 5).



© 2024 SmartDelta Consortium

Figure 5. Software Architecture for Anomaly Detection and Monitoring.

Case Studies. Case Study 2: High Throughput Log Processing

Use Case: Companies like Spotify and Netflix use Apache Kafka for log processing. These large-scale platforms generate huge log data across hundreds of microservices.

Solution: Kafka provides a high throughput and fault-tolerant solution, making it ideal for constructing a distributed tracing system and integrating OpenTelemetry with DIA4M (as shown in Figure 6). We researched the benefits of a scenario where DIA4M evolves into a central hub supporting both Elastic APM agents and vendor-neutral open-source collectors like OpenTelemetry. The evolution would bring significant advantages:

- **Flexibility**: Integration with a wide array of telemetry receivers (e.g., Prometheus, Jaeger, OTLP), processors (e.g., sampling, batching), and exporters (e.g., statsd, OTLP, Jaeger) provides a versatile and customizable observability solution.
- **Scalability**: Supporting diverse telemetry backends ensures that DIA4M can scale efficiently and adapt to various distributed systems' needs.
- **Versatility**: Accommodating a broader range of use cases and technologies makes DIA4M a more comprehensive observability platform.
-

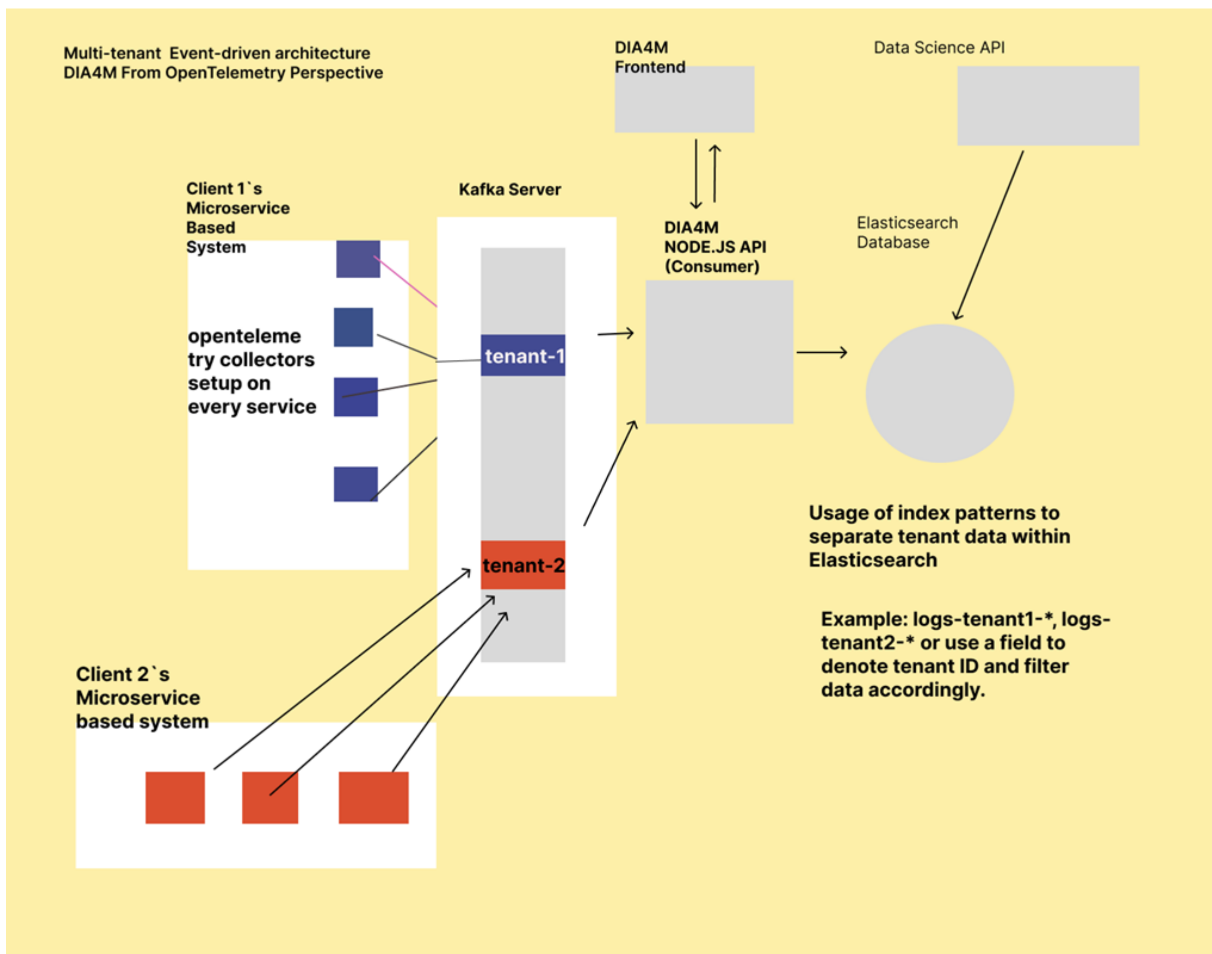


Figure 6: Integration of OpenTelemetry with DIA4M.

© 2024 SmartDelta Consortium

3.4. Model-Based Test Case Generation (IFAK-TCG)

Method Description

Based on a model of the demanded behavior (specification model) and with respect to specified test goals, test cases with test data are generated systematically by using a suitable algorithm[64]. Common modeling notations are graph-based models like UML State Machines, UML Activity Diagrams, and high-level Petri nets. Additionally, textual modeling languages are used as input models of existing test generation tools. Differences between the existing test generation methods and tools can be found primarily in the handling of internal and external data in the used input modeling notations. These differences are also reflected in the features of the test generation tools.

The specified test goals are very important for the test generation results. Normally test goals are coverage criteria regarding the used specification (requirements), including a specification model as the basis for the test generation. Generally, the specification model is graph-based, so graph-based coverage criteria (e.g., all nodes, all edges, all paths) are very common as test goals.

Other coverage criteria regarding the coverage of requirements properties have also been established as test goals. These are linked to associated elements of the specification model (usually nodes, edges) to enable the test generation method to generate the required test cases for an efficient test suite about the test goals. The method for test generation is based on Petri net unfoldings of SPENAT (Safe Petri Net with Attributes) models. It is possible to map a UML State Machine on a SPENAT, so a UML State Machine can also be used as an input specification model for test generation with this method[65].

In SmartDelta, IFAK improves its requirement-based test case generation and execution for functional requirements, extending it to non-functional properties such as performance. Furthermore, IFAK developed methods for automatic requirement prioritization regarding quality properties. The aim is an automatic test case prioritization for efficient regression testing.

3.5. Test Amplification (FOKUS-CBTS)

Method Description

The term “test amplification” was recently emphasized by the EU Horizon 2020 project STAMP (Software test amplification for DevOps). Test amplification is a test automation approach, where (usually) manually written testware (such as test cases, test data, test oracle, test configurations, test model) are automatically analyzed and processed by aiming at leveraging them in further test activities. The term purposeful is essential since different goals might be relevant, where the amplification may target. Test amplification might be used to increase the overall coverage of the system under test concerning its specification by means of further generated test cases or test data. Another target might be to derive further testware to uncover a particular kind of defect (e.g., fault). This is similar to the increase of coverage but different in the sense that the coverage of the

[64] Krause, J. (2011). Testfallgenerierung aus modellbasierten Systemspezifikationen auf der Basis von Petrinetzentfaltungen. Dissertation, Otto von Guericke University, Shaker Verlag

[65] Reider, M., Magnus, S., Krause, J. (2018). Feature-based testing by using model synthesis, test generation and parameterizable test prioritization. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (pp. 130-137)

specification is not required but the coverage of an anticipated defect. A third target might be modifying test (design) models if such models exist.

Tools and methods for test amplification currently focus on amplification of unit test cases or test configuration. For these approaches, prototypic open-source tooling support exists. However, no fully-fledged test amplification approach on a black-box level is currently available, not to mention delta analysis and delta generation.

Change-based Test Selection for Regression Testing

Change-based test selection is a form of test amplification that targets a purposeful selection of regression tests for faster execution and a shorter response cycle. The change-based test selection method we developed is based on changes to the code base that regularly happen in iterative-increment development. In many projects, automated regression tests are used to safeguard the development activities immediately right after code changes have been committed into the code base by the developers. As the software product grows, it is no possible to effectively run all the regression tests immediately to the code commit so developers have to wait for the results of the regression tests for a longer time than desired. This is particularly true for component integration tests which are usually executed using a CI pipeline.

The problem with regression tests is that most of the regression tests in a large regression test suite are less interesting to the developer who committed the code change. As a result, only those regression tests with a good potential to detect or uncover unwanted side-effects implicitly introduced are of higher interest to the developers. However, since side-effects are not predictable beforehand, one never knows which regression tests should be executed to safeguard code changes. As a result of this uncertainty, all regression tests are often executed brute-force. As described, this may lead to long waiting times on the developers' side. Other approaches strive to execute only a feasible number of regression tests, which leads to uncertainties concerning the ability to detect unwanted side effects.

Our method (and the developed prototype) is based on the following aspects:

- Abstract Syntax Tree (AST): It leverages AST analysis to gain deep insights into the codebase's structure and dependencies.

- Class Dependency Analysis: The plugin identifies intricate relationships between code components by analyzing class dependencies.

- Change Impact Analysis: This technique, widely recognized in literature, helps assess the impact of code changes on the entire system.

- Integration with Modern Libraries: The plugin combines these analysis techniques with modern libraries like JGit. This integration enhances the delta calculation process, making it faster and more efficient.

The prototype offers a practical and efficient solution to regression testing challenges in iterative-incremental development models. The plugin returns tests based on code changes and comprehensive coverage of potential side effects.

Improvements

We implemented the first version of a change-based test selection tool called CBTS Plugin. It is a regression selection tool for unit tests (either on the component or component integration test level)

© 2024 SmartDelta Consortium

integrated into a Maven-based CI pipeline. The CBTS Plugin has been integrated into the Vaadin CI pipeline and is already able to analyze delta changes made to the code base by the developers and select a set of regression tests that have a good potential to detect unwanted side-effects of the changes made to the code.

Although the automated execution of the selected test cases is not yet possible, the tool already operates in Vaadin's CI pipelines. Further technical integration and evaluation of the tool's effectiveness are subject to the next reporting period.

Purpose and Features

CBTS implements two different modes, i.e., Normal mode and Side-effect mode. Normal mode selects regression tests based on changes to the sources directly connected with a test case. Side-effect mode selects regression tests based on a calculation of the possibility to which a side-effect might have been introduced into otherwise unchanged areas of the source code. Therefore, CBTS introduces a classification of changes that may cause a side effect. Type 1 refers to changes to interfaces at the Java code level. These involve modifications that can influence how classes implement specific behaviors. Examples include adding, removing, or altering methods within interfaces. The impact of these changes is generally low, rated as 1. Type 2 involves changes to class inheritance at the Java code level. These adjustments affect class inheritance relationships and potentially impact the overall structure of class hierarchies. For instance, inheriting from a new abstract superclass is an example. These changes also have a low impact, rated as 1. Type 3 relates to changes in class imports at the Java configuration level. This includes alterations to import statements, which indicate modifications in external dependencies or libraries within Java files. Examples include importing a different library version or introducing new external dependencies. The impact here is high, rated as 10. Type 4 encompasses structural changes in code at the Java code level. These involve refactoring or restructuring the code logic, potentially leading to semantic defects. Structural changes can affect both modified and unmodified code areas. Examples include rearranging code blocks, changing method signatures, or leaving unchanged code areas impacted differently due to the refactoring. These changes are rated as having a low impact, scoring 1. Type 5 pertains to accessing external files at the Java code level. This involves code that reads content from external sources like files or services. An example is reading content from a different JSON file. The impact of these changes is considered high, with a rating of 10. Type 6 involves the dependency on external libraries at the Maven configuration level. These modifications introduce dependencies on external libraries, potentially altering the code's behavior and functionality. Examples include adding or removing dependencies in the POM file, which affects the project's build settings. These changes have a high impact, rated as 10. Type 7 refers to POM dependencies at the Maven configuration level. These involve modifications to the central pom.xml file of the project, such as adding or removing dependencies or modifying build configurations. The impact of these changes is medium, rated as 5. Type 8 includes other Maven configurations beyond the POM file. These involve setting changes, such as switching profiles or altering build phases, often in configuration files like settings.xml. An example is switching server configurations in the settings.xml file. The impact of these changes is medium, rated as 5.

A high-level workflow of how CBTS operates on a regression test suite is shown in Figure 7.

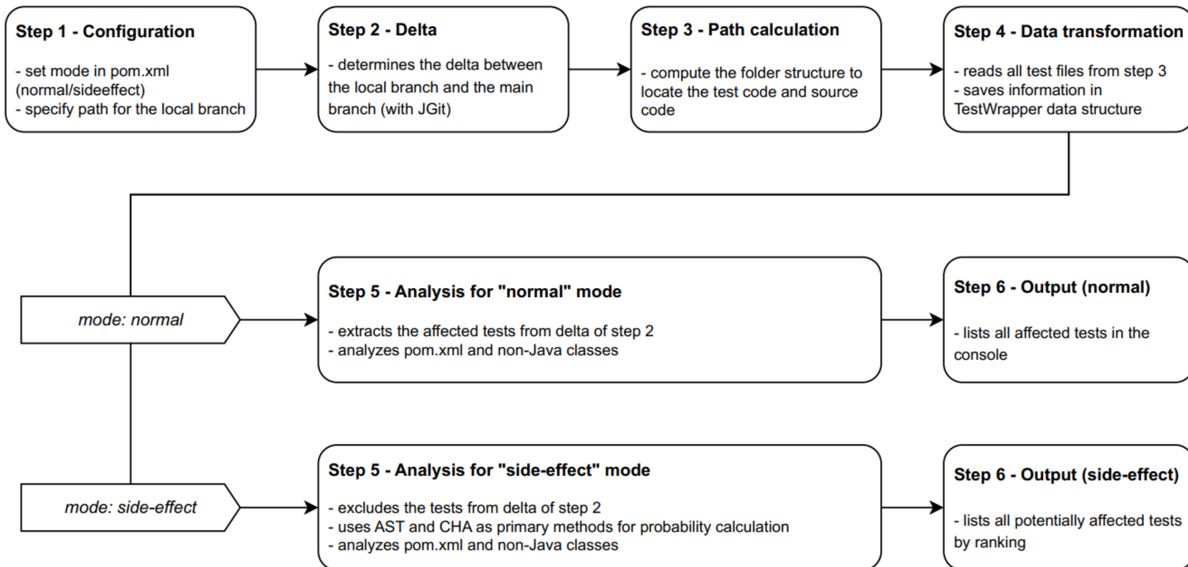


Figure 7: High-level workflow of CBTS.

Usage Instructions

The integration process of the CBTS tool into the Vaadin environment involves several steps. Initially, the necessary repositories must be obtained, including the CBTS plugin from Fraunhofer FOKUS Gitlab and a forked version of the Vaadin Flow repository. Subsequently, project configurations are set up using an Integrated Development Environment (IDE), where specific variables are edited to define the local path to the Vaadin flow repository in the CBTS plugin. Following the configuration, the CBTS plugin is executed using Maven, confirming its successful run.

The next step involves running the pom.xml of the desired Vaadin flow module using Maven. A Vaadin branch customized for the project should be selected. The CBTS tool implements both the normal and the side-effect modes according to 1. This requires the user to manually select the preferred approach by entering the desired mode in the prototype's pom file.

The CBTS tool is automatically initiated before the complete Vaadin workflow is executed, ensuring a seamless integration process. After calculating the potential side effects per test, all eligible tests are listed by ranking in the console.

Case Studies

CBTS has been applied to the SmartDelta Vaadin use case[66]. This case study aims to select "good" regression tests based on a classification of side effects. The case study focuses on optimizing regression testing in the Vaadin software project by classifying side effects to improve test selection efficiency. It categorizes side effects by impact levels, ranging from low (minor changes to interfaces or class inheritance) to medium (alterations in Maven dependencies) and high (changes in imports, external library dependencies, or access to external files). The results demonstrate reductions in regression test execution time while maintaining the ability to detect critical issues. Targeting tests affected by specific changes improved the balance between testing efficiency and risk mitigation. Applied within the CI/CD pipeline, this method improved pull request reviews and highlighted the feasibility of integrating change-based test selection in industrial software development.

[66] Balink, Robert, Marc-Florian Wendland, and Yuriy Yevstihnyeyev. "Selecting “good” regression tests based on a classification of side-effects." 2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, 2024.

© 2024 SmartDelta Consortium

3.6. VARA+: Variability-Aware Requirements Management and Analysis

Method Description

Software products are often developed in increments over existing products or their versions. In many project-based industries, such as the railway industry, a newly requested product frequently shares some commonalities with one or more existing products the company has already developed[67]. Therefore, when a new product must be delivered to a new customer, engineers must conduct a reuse analysis on the existing products to find reuse candidates that could satisfy the new customer's requirements. Furthermore, a requirement engineer must also decide which team can implement the new requirements. The reuse analysis and requirements assignment help companies save time and resources by avoiding redundant analysis, development, and testing efforts. In addition, reusing an already certified component or assigning the development to a team that has implemented similar components before might increase confidence in the new product.

While manual reuse analysis could save time and resources, it could also be prone to human error, dependent on the experience of some key engineers, and could be time-consuming. An automated requirements-level reuse recommender could aid the process of reuse identification at an early stage and enable engineers to conduct quick and meaningful reuse identification[68]. Furthermore, smart requirements assignments can also increase the speed of the development cycle and confidence in the product.

VARA+ assumes similar requirements can be used as proxies to retrieve similar software for reuse[69],[70]. Therefore, VARA+ uses state-of-the-art natural language models to exploit semantic similarities between requirements for reuse recommendation of existing software assets. In addition, VARA+ formulates the requirement assignment problem as a classification problem and uses deep-learning-based models to predict relevant teams for implementing new requirements. Results from evaluation in the railway industry show that VARA+ could recommend reuse with an average accuracy of around 80% and assign new requirements to correct teams with around 76% of average accuracy. The VARA+ toolchain also helps improve the quality of requirements. In particular, it is equipped with a NALABS engine adapted to identify smells in requirements, which in turn helps improve the quality of requirements.

Improvements

In SmartDelta, a web-based graphical user interface of the VARA+ tool with end-to-end requirements management features is partially implemented. We plan to improve the accuracy by adding more sophisticated language models and learners to predict software similarity for better retrieval and reuse recommendations. In addition, a requirements identifier and assignment extension to VARA+ are made within the scope of the SmartDelta project. Finally, the NALABS features were adapted for identifying requirements and quality issues.

---

[67] Abbas, M., Jongeling, R., Lindskog, C., Enoiu, E. P., Saadatmand, M., & Sundmark, D. (2020, October). Product line adoption in industry: an experience report from the railway domain. In Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A (pp. 1-11).

[68] Abbas, M., Saadatmand, M., Enoiu, E., Sundamark, D., & Lindskog, C. (2020, December). Automated reuse recommendation of product line assets based on natural language requirements. In International Conference on Software and Software Reuse (pp. 173-189). Springer, Cham.

[69] Abbas, M., Ferrari, A., Shatnawi, A., Enoiu, E. P., & Saadatmand, M. (2021, April). Is requirements similarity a good proxy for software similarity? an empirical investigation in industry. In International Working Conference on Requirements Engineering: Foundation for Software Quality (pp. 3-18). Springer, Cham.

[70] Abbas, M., Ferrari, A., Shatnawi, A., Enoiu, E., Saadatmand, M., & Sundmark, D. (2022). On the relationship between similar requirements and similar software. Requirements Engineering, 1-25.

© 2024 SmartDelta Consortium

Purpose and Features

The toolchain's main features aim to enable more efficient requirements processing in a requirement-centric development process. Below, we summarize the main features.

Requirements Identification

A typical first step in project-based companies is identifying technical specifications from large tender documents. VARA+ uses language models as binary classifiers to identify requirements from tender documents.

Requirements Quality

When the requirements are identified, the VARA+ toolchain uses an adapted version of the NALABS core engine to identify potential quality issues in the requirements' text and fix them.

Smart Requirements Allocation

Requirements are often treated as work items and must be assigned to be developed and tested, often by several teams at a large company. VARA+ toolchain uses language models together with traditional AI to smartly assign requirements to various teams at a company and generate explanations on the assignments to help enable a more informed requirements allocation process.

Reuse Identification

VARA+ uses requirements similarity as a proxy for software similarity to recommend the reuse of existing software in new projects.

Usage Instructions

Not all sub-tools are integrated into the toolchain. Nevertheless, we provide the instructions for sub-tools and the toolchain below:

- Requirements Identification: Available at https://github.com/a66as/REFSQ2023-ReqORNot
- Requirements Quality: Part of the toolchain.
- Smart Requirements Allocation: Part of the toolchain
- Reuse: Part of the toolchain
- Toolchain: Not publicly available and partially implemented: https://github.com/a66as/Vara-Tool

Case Studies

Case 1: In particular, the *requirement identification* part of the toolchain uses the BERT large language model for identifying requirements in tender documents with an average accuracy of 82%. The toolchain also allows checking the quality of the extracted technical specifications from the tender documents. Particularly, VARA+ compute metrics, such as Automated Readability Index, Complexity, and subjectivity, allow quality assessment of the extracted requirements. We evaluated various binary classifiers for our requirements identification sub-tool to select the best one for the pipeline. In particular, as shown in Figure 8, we consider weighted random (W. Rand.), Support Vector Machine (SVM), multinomial Naive Bayes (NMB), Decision trees (DT), Logistic Regression (LR), Random Forest (RF), BERT and its variants, and LSTM and its variants. We achieve an average accuracy of 82% in requirements identification in large tender documents from Alstom with the BERT language model. Results also show that the BERT-based requirements identification approach performs the best in terms of precision (P), recall (R), and harmonic mean (F1 score) with an average accuracy of 82%.

© 2024 SmartDelta Consortium

Pipeline	Setup	Weighted Average			Macro Average			Avg. A.	Time (mins)	
		Prec.	Rec.	F1	Prec.	Rec.	F1	A	Tr	Ts
W. Rand.	Freq. based	.49	.49	.49	.49	.49	.48	.49	-	-
SVM	Norm., PCA	.79	.79	.79	.80	.78	.78	.78	.70	.02
pSVM	Norm., PCA	.78	.78	.78	.79	.77	.77	.78	.74	.09
NB	Norm.	.74	.69	.69	.73	.71	.69	.69	<.01	<.01
pNB	Norm.	.74	.68	.67	.72	.70	.68	.68	.29	.07
DT	Norm.	.72	.72	.72	.71	.71	.71	.71	<.01	<.01
pDT	Norm.	.71	.71	.71	.71	.71	.71	.71	.29	.07
LR	Norm., PCA	.79	.79	.78	.79	.77	.78	.79	.30	<.01
pLR	Norm., PCA	.78	.78	.78	.79	.76	.77	.78	.41	.07
RF	Norm.	.79	.79	.79	.79	.78	.78	.79	<.01	<.01
pRF	Norm.	.79	.78	.78	.79	.77	.77	.78	.38	.07
LSTM	FT custom	.77	.77	.77	.76	.76	.76	.77	1	.02
pLSTM	FT custom	.75	.75	.75	.75	.75	.74	.75	1	.08
LSTM	FT pre-train	.75	.75	.75	.74	.74	.74	.75	2	.02
pLSTM	FT pre-train	.72	.72	.72	.72	.72	.72	.72	1.2	.08
LSTM	GLV custom	.77	.77	.77	.77	.76	.76	.77	2	.02
pLSTM	GLV custom	.76	.76	.76	.76	.75	.76	.76	1.2	.09
LSTM	GLV pre-train	.78	.78	.78	.78	.77	.78	.78	2	.02
pLSTM	GLV pre-train	.78	.78	.78	.78	.77	.78	.78	1.3	.08
SciBERT	uncased	**.82**	.81	.81	**.82**	.80	.80	.81	34	.25
pSciBERT	uncased	.80	.78	.76	.81	.75	.75	.78	32	.30
RoBERTa	base	.81	.81	.81	**.82**	.80	.80	.81	39	.27
pRoBERTa	base	.80	.79	.79	.81	.78	.78	.79	37	.32
BERT	base, cased	**.82**	**.82**	.81	**.82**	**.81**	**.81**	**.82**	35	.29
pBERT	base, cased	.79	.79	.79	.79	.79	.79	.80	32	.32
BERT	base, uncased	**.82**	**.82**	**.82**	**.82**	**.81**	**.81**	**.82**	34	.29
pBERT	base, uncased	.80	.80	.80	.80	.79	.79	.80	32	.33
XRBERT	base	**.82**	.81	.81	**.82**	.80	**.81**	.81	57	.29
pXRBERT	base	.78	.77	.77	.78	.76	.76	.77	41	.25
DisBERT	base, cased	.81	.81	.81	.81	.80	.80	.81	31	.13
pDisBERT	base, cased	.80	.80	.80	.80	.79	.79	.80	25	.18
DisBERT	base, uncased	.81	.81	.81	.81	**.81**	.80	.81	31	.15
pDisBERT	base, uncased	.80	.80	.70	.81	.78	.79	.80	29	.21
XLNet	base	.81	.81	.80	.81	.80	.80	.81	47	.36
pXLNet	base	.81	.80	.80	.81	.79	.79	.80	47	.42
S-BERT	10% train	.75	.75	.75	.75	.74	.75	.75	24	.14
pS-BERT	10% train	.73	.73	.73	.72	.72	.72	.73	18	.20
Mini-LM	10% train	.74	.74	.74	.74	.74	.74	.74	7	.04
pMini-LM	10% train	.72	.72	.72	.72	.72	.71	.72	6	.10
S-BERT	20% train	.77	.77	.76	.76	.76	.76	.77	45	.17
pS-BERT	20% train	.74	.74	.74	.74	.74	.74	.74	37	.20
Mini-LM	20% train	.75	.75	.75	.75	.74	.74	.75	14	.03
pMini-LM	20% train	.72	.72	.72	.72	.72	.72	.72	11	.10

Figure 8: Evaluation results of VARA+ requirements identification

Case 2: A railway vehicle typically consists of more than 20 sub-systems; once the requirements are extracted, they must be allocated to various teams responsible for developing and testing those sub-systems. In this regard, the VARA+ toolchain provides the REQA approach (shown in Figure 9) for allocating requirements to various teams. The approach combines large language models with case-based recommender systems to assign requirements to teams and generate useful explanations to enable a well-informed allocation decision.

Pipeline	Setup	Weighted Average P	Weighted Average R	Weighted Average F1	Macro Average P	Macro Average R	Macro Average F1	Avg. A. A
W. Rand.		.11	.11	.11	.07	.07	.07	.11
SVM	Norm., PCA	.65	.62	.60	**.73**	.53	.58	.64
pSVM	Norm., PCA	.66	.64	.62	.72	.56	.60	.65
MNB	Norm.	.56	.53	.47	.55	.31	.32	.52
pMNB	Norm.	.54	.54	.48	.50	.31	.32	.54
DT	Norm.	.48	.46	.46	.50	.46	.47	.46
pDT	Norm.	.49	.48	.48	.50	.46	.46	.48
LR	Norm., PCA	.64	.59	.56	.71	.43	.48	.59
pLR	Norm., PCA	.65	.61	.58	.73	.47	.51	.61
RF	Norm.	.61	.58	.57	.69	.52	.57	.58
pRF	Norm.	.61	.59	.58	.69	.54	.58	.59
SciBERT*	uncased	**.68**	.67	**.67**	.71	**.66**	**.67**	.67
pSciBERT	uncased	.64	.64	.63	.69	.61	.63	.64
RoBERTa	base	.66	.66	.65	.70	.64	.65	.66
pRoBERTa	base	.61	.61	.58	.64	.54	.55	.61
BERT	base, cased	.67	.67	.66	.69	.63	.64	.67
pBERT	base, cased	.65	.64	.62	**.73**	.57	.60	.64
BERT	base, uncased	**.68**	**.68**	.66	**.73**	.63	.65	**.68**
pBERT	base, uncased	.67	.66	.64	.71	.62	.64	.66
LSTM	FT custom	.53	.50	.49	.48	.43	.43	.50
pLSTM	FT custom	.58	.57	.56	.57	.54	.53	.57

Figure 9: REQA Evaluation with various pipelines

As shown in Figure 10, we evaluate various classifiers for the REQA approach on Alstom's use case. In particular, we evaluate weighted random (W. Rand.), Support Vector Machine (SVM), multinomial naive Bayes (NMB), Decision trees (DT), Logistic Regression (LR), Random Forest (RF), BERT and its variants, and LSTM. Results show that the BERT-based REQA approach performs the best in terms of precision (P), recall (R), and harmonic mean (F1 score) with an average accuracy of 68%.

Case 3:

Stats.	Random		**TFIDF**		D2VT		D2VW300		FTT		FTW300	
%→	**A**	**E**	**A**	**E**	**A**	**E**	**A**	**E**	**A**	**E**	**A**	**E**
AVG.	11.73	2.24	**74**	**57.4**	10.71	2.24	65.2	41.45	56.53	43.30	59.38	44.35
SSD	5.01	2.93	3.87	4.26	4.59	1.67	3.72	3.42	4.80	4.75	6.25	5.25
VARS	25.17	8.60	14.99	18.21	21.10	2.81	13.86	11.71	23.07	22.57	39.12	27.57

Figure 10: Reuse recommendation results

In most safety-critical domains, the development of new systems starts with natural language requirements. The VARA+ toolchain uses Natural Language Processing algorithms together with clustering on existing requirements to learn a recommendation model for reuse from existing requirements. The learned model is then queried to generate reuse recommendations of existing software for new upcoming requirements.

Initial evaluation at Alstom shows (in Figure 10) that the VARA+ toolchain can recommend the reuse of existing software components with an average accuracy (A) of around 78% (later improved to 80%) with the term frequency-inverse document frequency approach (TFIDF). The same TFIDF-based approach also produced better results regarding the exact match ratio metric (E). We also evaluated Doc2Vec and FastText with self-training (D2VT & FTT) and pertained (D2VW300 &

© 2024 SmartDelta Consortium

FTW300). Furthermore, a qualitative evaluation with engineers at Alstom shows that the tool generates insightful and useful reuse recommendations.

In the scope of SmartDelta, we plan to further improve this toolchain with a dashboard and more fine-grained reuse recommendations & requirements allocation.

3.7. SoHist - Historical Quality Evolution Tool

Method Description

Software quality assurance is a technique to guide and evaluate software development to deliver high-quality software products[71]. To ensure this high code quality, the open-source software quality framework *SonarQube*[1] (Community Edition) can be helpful. Running SonarQube static analysis capabilities, triggered by a commit on GitHub or GitLab, gives you an "up-to-date" profile about your current code artifact and its correlated measurements of technical debts [72] [73].

[1]SonarQube – A Common Platform for Code Analysis
SonarQube claims that over 100,000 organizations use the open-source platform to analyze their code artifacts [74]. Evaluating multiple lines of code, SonarQube identifies code smells, warns about security concerns, and delivers various lines of code metrics, including indicators for

- Code complexity (Cognitive Complexity, Cyclomatic Complexity, etc.)
- Duplications (Duplicated Lines, Duplicated Files, etc.)
- Issues (New Issues, Open Issues, Reopened Issues, etc.)
- Maintainability (Code Smells, Maintainability Rating, Technical Debt, etc.)
- Reliability (#Bugs, Security Hotspots, Security Review Rating, etc.)
- Tests (Test Coverage, Conditions by Line, etc.) [75] [76]

Considering code changes – also known as "Deltas" - SonarQube has a visualization type, the evolution graph. This chart visualizes how a project has developed over time regarding different code quality aspects [77]. Using this platform encourages companies to follow good coding guidelines.

Nevertheless, having a current analysis is good, but seeing the projects' total (commit) history is even better. A complete overview of the evolution of code quality enables developers to find correlated vulnerabilities. For instance, multiple versions of a third-party plugin led to increased bugs. Going through the software evolution shows you when the issue first came up. In addition, examining the project's history provides the project's managers valuable insights into the project's progress.

In the latest version, SonarQube 9.7 Community Edition (10. Nov. 2022), the evolution of a project can arise if you have integrated SonarQube in your continuous integration pipeline since the start of your project. A commit on the main branch triggers an analysis. In such a way, you can build up

[71] M.-C. Lee, "Software Quality Factors and Software Quality Metrics to Enhance Software Quality Assurance," *Br J Appl Sci Technol*, vol. 4, no. 21, pp. 3069–3095, Jan. 2014, doi: 10.9734/BJAST/2014/10548
[72] "GitLab Integration | SonarQube Docs." https://docs.sonarqube.org/latest/analysis/gitlab-integration/ (accessed Nov. 10, 2022)
[73] "GitHub Integration | SonarQube Docs." https://docs.sonarqube.org/latest/analysis/github-integration/ (accessed Nov. 10, 2022)
[74] "About | SonarQube." https://www.sonarqube.org/about/ (accessed Nov. 11, 2022)
[75] M. T. Baldassarre, V. Lenarduzzi, S. Romano, and N. Saarimäki, "On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube," *Inf Softw Technol*, vol. 128, Dec. 2020, doi: 10.1016/J.INFSOF.2020.106377
[76] "Metric Definitions | SonarQube Docs." https://docs.sonarqube.org/latest/user-guide/metric-definitions/ (accessed Nov. 09, 2022)
[77] "Activity and History | SonarQube Docs." https://docs.sonarqube.org/latest/user-guide/activity-history/ (accessed Nov. 10, 2022).

© 2024 SmartDelta Consortium

your evolution graph. SonarQube's intended approach leads to limitations and challenges in evaluating a complete Git history.

(1) *Comparability*: SonarQube and possible plugins update over time so that the quality measurement approaches could change. That means, for your analysis, if you update SonarQube or plugins, the comparability of your code artifacts suffers.

(2) *Historical Analysis at any time*: Just consider whether you take over a system or have a long-term project that expects a SonarQube integration. How can you analyze the history of the related project? SonarQube focuses on the integration of "current" commits.

(3) *Flexibility*: One only wants to regard a specific time range of commits, analysis commits of (a) specific person(s) or a particular branch.

That is where SoHist comes into place and is developed within SmartDelta.

Improvements

SoHist is built from scratch to enhance the current limitations (1), (2), and (3). It uses multiple advantages of SonarQube and adds new features. Figure 11 illustrates the tool setup.

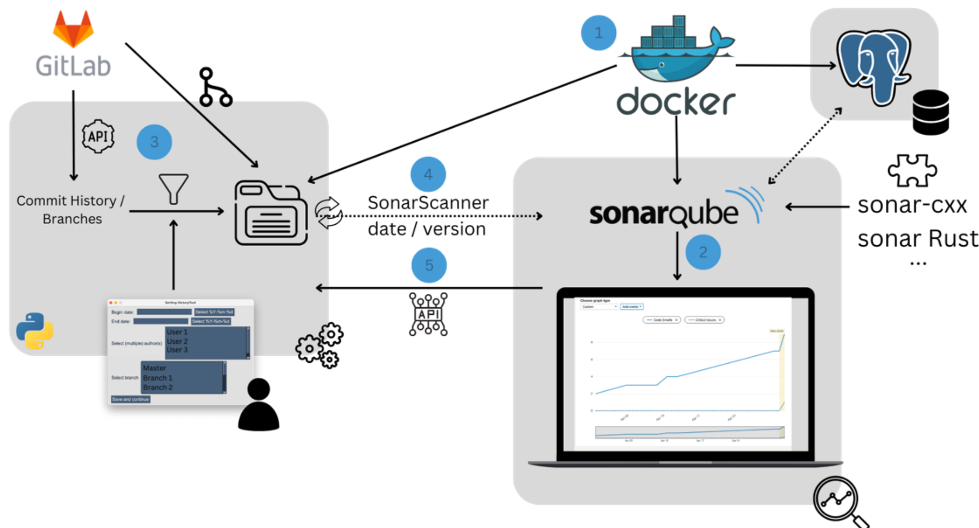


Figure 11. SoHist Tool Setup and Overview.

(1) Docker is used to deploy the different components: a) PostgreSQL, b) SonarQube, and c) SoHist.
(2) SonarQube is executed self-hosted in a container and is accessible for run code analysis. The integration of add-ons (e.g., sonar-cxx) can automatically be included.
(3) The user will be granted access to the GitLab repository to be analyzed. Afterward, the tool summarizes all the necessary information from the repository, and the user can select specific filter values such as time range, specific committers, or a particular branch.
(4) The tool automatically executes the Sonar Scanner to run the analysis depending on the provided selection criteria. Gradually, the evolution of the projects emerges.
(5) After a run, the tool can access information from SonarQube via Rest calls and display different code quality aspects over time.

© 2024 SmartDelta Consortium

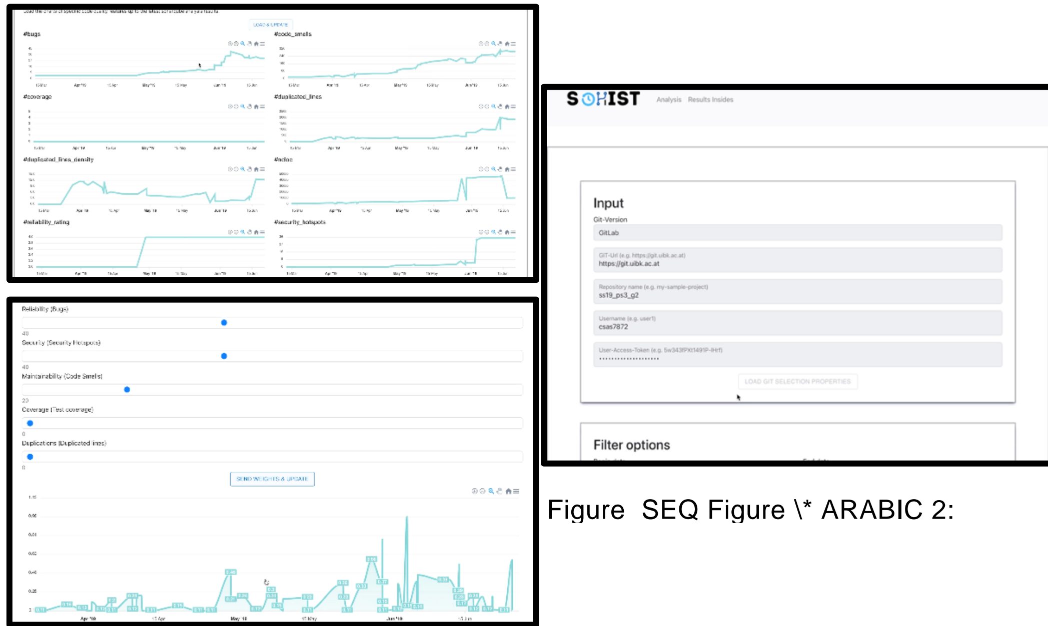


Figure 12. SoHist Dashboard.

Purpose and Features

SoHist is one of the tools developed within SmartDelta that facilitates the accurate analysis and determination of quality implications of each change and increment to a system (as shown in Figure 12). SoHist addresses these issues by building on SonarQube's strengths and providing additional features to assess and prioritize technical debt. The containerized application enables companies to connect to their GIT repository and execute retro-perspective code analysis with several filtering options (committers, time range, or branch).

Usage Instructions

To implement SoHist, users should initiate a Docker configuration and enter specific project details, including GitLab credentials and identifiers, into the SoHist web interface. This setup allows SoHist to access historical commit data from SonarQube, enabling detailed retrospective analysis of code quality metrics.

Users may then apply filters to focus on data by date, branch, or contributor, allowing for targeted technical debt analysis across different project areas. Further, setup and usage instructions are available at https://github.com/bdornauer/sohist.

Case Studies

In the first case study[78] involving c.c.com Moser GmbH, SoHist analyzed energy consumption patterns related to firmware updates on the company's BLIDS sensors. The primary focus was identifying how specific code changes impacted battery performance, as the sensors rely on periodic battery replacement. Using SoHist's retrospective analysis capabilities, c.c.com Moser GmbH accessed code quality metrics, including McCabe Cyclomatic Complexity and Lines of Code, which prior research had shown to influence energy consumption in embedded systems. Through this approach, c.c.com identified patterns linking specific design decisions with increased energy demands, which enabled them to optimize the firmware's efficiency over time. As a result, c.c.com

[78] Dornauer, Benedikt, et al. "SoHist: A tool for managing technical debt through retro perspective code analysis." *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 2023.

© 2024 SmartDelta Consortium

can now correlate code quality improvements directly with reduced energy drain, significantly extending battery life across deployments.

In the second case study, Software AG utilized SoHist to conduct a historical quality analysis on one of its research projects. The analysis focused on tracking technical debt indicators, specifically bugs and code smells, across the project's Master branch. Early results indicated a buildup of technical debt due to prioritizing new features over code quality during the project's initial stages. Software AG understood how these issues evolved by employing the Weighted Code Evolution Significance visualization in SoHist. The analysis revealed a critical point in March when a quality assurance process was introduced. Following this process change, SoHist data showed a decrease in code quality issues, notably bugs and code smells, showing some evidence of the new process's effectiveness in reducing technical debt. This historical insight allowed Software AG to quantitatively evaluate the process change's benefits, validating its impact on software reliability.

3.8. SONATA

Method Description

Software development companies aim to reduce the time spent on test generation when creating a new product or delta. Knowing about previously developed products or variants is essential to correctly choose which tests to apply. SONATA addresses this need by employing code analysis and graph modeling techniques to optimize the testing process in software description.

SONATA is a tool that takes a Java code repository as input and extracts important elements to recommend test cases. For example, it obtains a class diagram with classes, methods, attributes, and relationships. These entities are then modeled in a knowledge graph, a structured representation that facilitates analysis and comparison with previous developments. The semantic matchmaking technique establishes correspondences between new developments and previous versions of products or variants. This technique utilizes programming semantics and algorithms to detect similarities and differences in the code's structure and functionalities. Graph pattern comparison helps identify recurring or unique patterns in current developments in relation to previous ones.

Once these comparisons are established, SONATA recommends a set of relevant tests based on the tests applied to similar developments. This approach saves time by reusing relevant tests and ensures a more comprehensive test coverage tailored to the new product or variant.

Improvements

Currently, the development team spends a considerable amount of time generating tests. Additionally, when dealing with a delta, all necessary tests are not guaranteed to be performed to ensure quality. SONATA aims to better understand the structure and functionality of the code by accurately representing classes, attributes, methods, relationships, and other elements through a knowledge graph. This representation allows SONATA to apply semantic matchmaking capabilities to better analyze code similarities and differences. As a result, SONATA can provide relevant test case recommendations, ensuring that the most appropriate tests are selected to maintain and improve the quality of the software.

© 2024 SmartDelta Consortium

Purpose and Features

The main objective of SONATA is the automatic recommendation of test cases for new deltas or new software developments. By automating the recommendation process, SONATA aims to ensure that all necessary tests are identified and executed, thereby maintaining and enhancing the quality of the software.

Features:
SONATA searches for projects similar to the current development within the company, facilitating the reuse of relevant test cases. This feature leverages the knowledge graph to identify comparable code structures and functionalities, enabling the tool to suggest tests that have been effective in past projects. SONATA visually represents the class diagram, including methods, attributes, and relationships extracted from the source code. SONATA also recommends test cases based on code analysis and comparisons with previous projects. By analyzing the similarities and differences in the codebase, SONATA ensures that the recommended tests are relevant and comprehensive.

Usage Instructions

SONATA has been developed as a web application. To use SONATA, follow these steps. Start SONATA and load the repository: You can provide a GitHub repository URL. After loading the repository, select the Analyze Repository option. SONATA will generate and display a class diagram based on the source code from the provided repository. This diagram visually represents the code structure and includes classes, methods, attributes, and relationships. Choose the Find Similar Projects option to view a list of projects within the company that are highly similar to the provided repository. This feature helps identify relevant past projects, facilitating the reuse of existing cases and ensuring comprehensive test coverage. After analyzing the repository and identifying similar projects, click on the Generate Test Cases button. SONATA will recommend relevant test cases based on the analysis and comparisons with similar developments.

Case Studies

SONATA is currently being tested with eight internal repositories at Izerties. This case study involves various software developments within the company, allowing the team to assess SONATA's ability to accurately analyze code structures, identify similarities with existing projects, and provide appropriate test recommendations. Using these repositories, Izertis can determine how well SONATA integrates into their development workflow and how it enhances the efficiency and quality of their testing processes. The insights gained will help refine SONATA´s algorithms and ensure that the tool effectively meets the specific needs of Izertis´s development team.

3.9. UI Test Generator - User Interface Test Design

Method Description

At Vaadin, we want to allow our customers and users to maintain the quality of the applications built with our platform. One of the most time-consuming parts of the quality assurance process for web application development is the creation of the user interface tests. Selenium is an open-source set of tools and libraries to support browser automation that simplifies the mentioned process. Vaadin is providing a set of tools on top of Selenium. Vaadin UI Test Generator is based on Selenium and allows users to create user interface units and end-to-end tests. It is specifically designed for testing Vaadin applications and provides several features that have advantages over default Selenium testing capabilities:

- Helper methods to interact with Vaadin components and elements in the shadow DOM of Vaadin components
- Automatic suspension and resumption of client-side execution for asynchronous testing

© 2024 SmartDelta Consortium

- API for conducting visual regression tests in the form of screenshot comparison
- Parallel testing for Vaadin applications

Vaadin, in a recent release, added official support for UI unit testing into the UI Test Generator based on the Karibu Testing library. It enables unit testing to be handled without launching a browser or a web server. With the mentioned above tooling, there are three main approaches for user interface testing of Vaadin applications:

- UI Unit testing - testing without the need to run both the browser and the servlet container
- End-to-End Testing - testing via simulation of user interaction with an application: performs the tasks specified using Java code and verifies that the expected actions take place in the application
- Testing with Selenium - manual handling of asynchronous actions and DOM manipulations for End-to-End testing

Vaadin constantly enhances the existing tooling and uses it internally for testing the parts of the platform within smoke testing, component testing, and platform testing. The mentioned tools are used to create the tests, but the process of creation is still manual.

In the context of the SmartDelta project, Vaadin has made significant strides in designing and developing a new tool dedicated to automating the generation of user interface tests. Our objectives encompass analyzing application code to extract critical components such as routes, interactive user interface elements, events, and predefined patterns within the platform. These extracted components serve as the foundation for generating user interface tests, adhering to predefined rules established by Vaadin. These rules encompass interactive elements and patterns found within the platform, including forms, inputs, layouts, routing, and more. Leveraging existing tools from Vaadin's UI Test Generator, this process results in the creation of initial tests that are ready for further enhancement and amplification by developers. This same methodology extends to internal test applications, enhancing the quality assurance process for deltas.

Our tool prototype is now accessible on GitHub as a Maven plugin, providing a robust foundation for ongoing development and collaboration. This prototype comprises a parser and generator, with the parser utilizing Eclipse JDT to analyze the source code of applications and the generator producing basic tests for views based on interaction elements using Roaster, a high-level API internally dependent on Eclipse JDT. The prototype has undergone verification on internal starter projects, ensuring its functionality and effectiveness. We have established a collaborative partnership with Izertis&Sotec, focusing on further developing the tool to align with their key performance indicators (KPIs). This collaboration involves testing the tool on projects featuring Java backend and Javascript/Typescript frontend technologies.

Improvement

Building on the completed achievements, we have laid out plans for the upcoming development project, encompassing several areas. Our commitment extends to generating mock tests and meaningful and comprehensive user interface tests, elevating the overall testing process, and exploring the use of Generative Artificial Intelligence, including LLMs and other AI models, to enhance test generation and parsing in the tool. Our research is around understanding and modeling interactions between diverse UI elements to enable more effective testing. We delve into research on automatic test migration features, enabling seamless adaptation of tests to the next versions of applications. In addition to its applicability within the Vaadin framework, we want to generalize the tool's utility, making it relevant to a broader spectrum of projects and increasing its versatility.

© 2024 SmartDelta Consortium

Purpose and Features

The UI-test generator is a Java tool that is available as a Maven plugin or a Java library and designed for integration into any code generation software. Its primary purpose is to generate user interface (UI) tests for web applications. This process begins with a YAML configuration file that enumerates the views to be tested. By analyzing the source code and the HTML output in the browser for each view, the tool generates a Gherkin file that outlines the steps for the test. Subsequently, it produces the corresponding source code for the user interface elements described in the Gherkin file.

The UI test generator provides a powerful solution for user interface tests for web applications. It follows a two-phase process: first, a parser reads a file containing views of the web application, analyzes the source code and the generated HTML structure, and defines the necessary UI tests in Gherkin format. Second, a generator uses these Gherkin files to produce executable test code. The tool is language-agnostic, currently supporting Java and TypeScript for application code, and framework-agnostic, supporting Playwright for both Java and TypeScript and UI Test Generator for Java. This independence ensures that the generated tests can be applied to any web application, regardless of the underlying framework.

Usage Instructions

In the application project, there should be a YAML file specifying the views to be tested, as shown in the following view:

```yaml
- className: com.example.DataGridView
  route: data-grid
  file: src/main/java/com/example/DataGridView.java
  framework: flow
- className: HelloWorld
  route: hello-world
  file: src/main/frontend/views/hello-world.tsx
  framework: react
```

The tool can be executed in two ways: as a Maven plugin if the project uses Maven for compilation or as a Java library that can be used with other tools. To use the library as a Maven plugin, start the application to start listening at http://localhost:8080. Once the application runs, execute the view parser using the Maven command `mvn com.vaadin:ui-tests:parse`. This step processes the application views to extract the necessary information for test generation. Next, generate the tests by executing the command `mvn com.vaadin:ui-tests:generate`. This will create test cases based on the parsed views, ready to be executed in your development environment. To run the generated Java tests, use the Maven command mvn test. If the tests were generated in TypeScript, execute them using the command npm test.

To use the library as a Java library, you can use the static methods provided by the `TestCodeGenerator` class to automate test generation and integration. Start by configuring the OpenAI API key using the `configureKeys(String aiToken)` method, where you provide a valid API token. To generate tests, use the `generateTests(UiRoute, TestFramework)` method. The first parameter includes basic information about the view, such as the source code and rendered HTML, as specified in the YAML format. The second parameter defines the type of test to be generated, such as UI Test Generator for Java, *Playwright Java,* or *Playwright TypeScript*. This method returns the generated code as a string in Java or TypeScript. Once the tests are generated, you can write the code to the filesystem using the `writeUiTest(UiRoute, source)` method. The `addTestDependencies(UiRoute, projectRoot)` method adds the necessary

dependencies for the selected framework to complete the integration. This modifies the pom.xml or package.json file in the specified project directory to include the required dependencies.

Case Studies

Integration with Vaadin Copilot. The UI Test Generator tool is integrated into Vaadin Copilot, starting with version 24.5. Vaadin Copilot, an AI-powered development assistant, allows developers to efficiently build and modify UI components through drag-and-drop functionality and real-time code updates within supported IDEs. With the integration of the UI Test Generator, Copilot users can now automatically generate basic UI tests directly within the development environment. This integration streamlines the testing setup for views built with Flow, Hilla, or React, allowing Vaadin to validate the tool with customers and identify potential improvements.

Planned Testing with Izertis Applications. To further expand the applicability of the UI Test Generator, we plan to collaborate with Izertis to test the tool on their provided applications. This will involve generating and executing UI tests across various application views, offering valuable insights into the tool's versatility and performance in different real-world scenarios. Feedback from these tests will be essential for refining the tool's capabilities and ensuring it meets the needs of a wide range of web applications, regardless of the framework or technology stack used.

3.10. MUT4SLX – Mutation Testing for Simulink

Method Description

Within SmartDelta University of Antwerp NEXOR leverages its expertise with test automation to improve the quality of families of systems. We offer a proof-of-concept tool (named MUT4SLX) for automatic mutant generation and test execution of Simulink and Stateflow models.

Mutation testing is a recommended practice in industrial standards and regulations such as ISO 26262 and IEC 61508 for functional safety. It complements the traditional coverage measures for Simulink and Stateflow (block coverage, signal coverage, transition coverage, …). The technique injects artificial faults in the system under test (based on a list of mutation operators), subsequently executing the test suite to see whether it is strong enough to catch the injected fault.

“Shift-left” is a commonly adopted paradigm in the automated software testing community, emphasizing that tests should be executed against the system under test as early as possible. In this paradigm, test suites are expected to be strong, catching defects before they are deployed into production. Code coverage (i.e. statement, branch, etc.) is commonly used to measure the strength of a test suite, although it is known to be a poor indicator of its actual strength. In the academic literature, mutation testing is acknowledged as the state-of-the-art technique to assess the fault detection capacity of a test suite. The technique injects artificial faults in the system under test (based on a list of mutation operators), subsequently executing the test suite to see whether it is strong enough to catch the injected fault.

Improvement

Several experience reports illustrate that mutation testing can support a “shift-left” testing strategy for software systems coded in textual programming languages like C++. However, mutation testing tools for visual languages like Simulink and Stateflow are missing. MUT4SLX bridges this gap.

© 2024 SmartDelta Consortium

MUT4SLX is currently validated within a single company and is estimated at TRL level 4 (Validated in the Lab). We intend to bring it to TRL level 5 (Validated in relevant industrial environment) via future pilot projects.

To bring the proof-of-concept to TRL level 5, we plan the following improvements:

- Integration with git-based build pipelines
- Traceability: Requirement-based testing
- Additional mutation operators (tailored to the needs of the pilot projects)
- Scalability (Parallel test execution & mutant execution)

Purpose and Features

As shown in Figure 13, MUT4SLX comprises two main components: (i) mutant generation and (ii) mutant execution. The mutant generation component is responsible for reading relevant parameters from the configuration and loading the model, as well as identifying the possible mutants. Users can use the CSV output generated by this component to preview the number of mutants, ignore certain mutants, or prioritize specific mutants before mutant execution takes place. The mutant execution component executes the given test cases on each of the mutants to count the number of killed and live mutants. Currently, two test execution engines are supported: Signal Builder and Simulink Test.

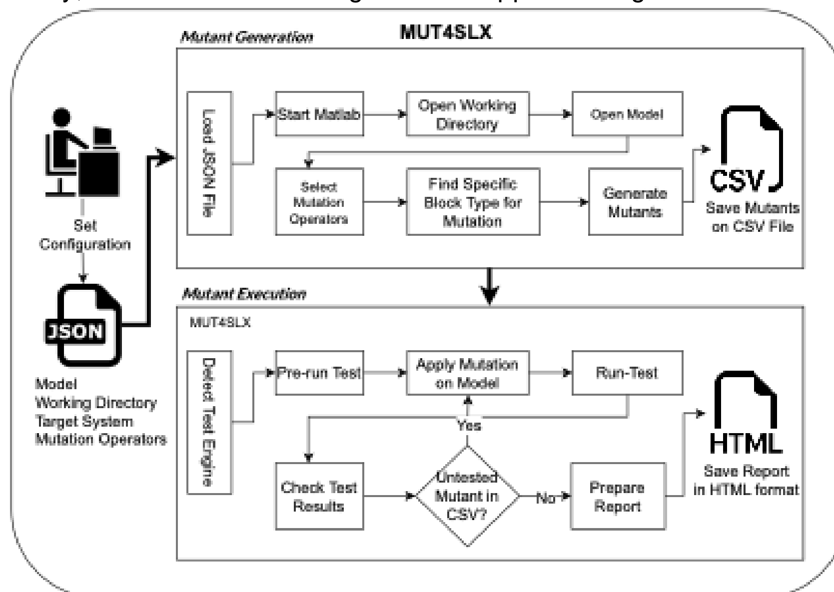


Figure 13 Overview of MUT4SLX.

The mutation operators are defined in close collaboration with an industrial partner. We first formulated mutation operators associated with the bugs in their database. Later, we added other mutation operators deemed relevant based on piloting MUT4SLX within the development toolchain. To prioritize these operators, we considered the frequency of the occurrences of the bugs associated with them. We then conducted several manual trials to determine a set of mutation operators (see Table 3) suitable for our needs.

Table 3. Simulink and Stateflow Mutation Operators

SIMULINK			STATEFLOW		
#	Name	Description	#	Name	Description
1	ROR	Relational Operator Replacement	1	ROM	Relational Operator Mutation
2	LOR	Logical Operator Replacement	2	LOM	Logical Operator Mutation
3	ASR	Arithmetic Sign Replacement	3	BOM	Bitwise Operator Mutation
4	MMR	Min-Max Replacement	4	MOM	Math Operator Mutation
5	ICR	If Condition Replacement	5	NOM	Non-bitwise Operator Mutation
6	TOR	Trigonometric Operator Replacement	7	CCM	Control Chart Mutation
7	MOR	Math Operator Replacement	8	CDR	Control chart Duration Mutation
8	PMR	Product Multiplication Replacement	9	TOM	Transition Order Mutation
9	POR	Product Operator Replacement			
10	FIR	For Index Replacement			
11	FLR	For Limit Replacement			
12	UDO	Unit Delay Operation			
13	STR	Switch Threshold Replacement			
14	SCR	Switch Criteria Replacement			
15	CR	Constant Replacement			

Usage Instructions

A proof-of-concept version is available under the GPL-3.0 license at https://github.com/haliliceylan/MUT4SLX. This version features the mutant injection for Simulink only. People can contact Prof. Serge Demeyer (serge.demeyer@uantwerpen.be) to negotiate access to the full version, which includes mutant injection for Stateflow and the test execution engine. We will set up a training session for teams interested in piloting the tool.

Case Studies

The Simulink component of MUT4SLX was validated on a Helicopter Control System, a representative example built following the guidelines set out in ARP4754A, DO-178C, and DO-331 certifications. The validation shows that MUT4SLX can inject 70 mutants in less than a second, resulting in a total analysis time of 8.14 hours. Only 40% of the mutants are killed, illustrating that the system is undertested by the automated tests in the project.

The Stateflow component of MUT4SLX was validated against two sets of Stateflow models available on the internet: a Microwave System and a Cruise Control System. The test suites for Microwave and Cruise Control need to be mutation-adequate. The first achieves a 70% mutation score by killing 21 out of 30 generated mutants. The latter manages to kill 33 of the 63 generated mutants. The mutant generation is very fast; it takes 20-30 ms per mutant. Mutant execution consumes more time than mutant generation, and simulation dynamics seem to affect the actual timings more than the number of test cases or the number of mutants in the project.

3.11. DRACONIS

Within SmartDelta, Mälardalen University has developed a static analysis framework targeting block-based programming languages for *delta-aware* linting and static verification of design rules. The prototype tool DRACONIS (**D**esign **R**ule **A**nalysis and **C**hecking **O**f **N**orms in **I**EC & **S**imulink) parses and provides analysis reports in multiple formats to developers either during *edit time* or in a format for reporting during assessment phases.

Method Description

DRACONIS, at its core, is an analysis framework. The overall toolchain can be seen in Figure 14. The current version supports models either directly through parsing or indirectly by reading a JSON representation. Given a model, the analyzer component will extract a set of metrics and perform a dataflow analysis. This will be considered a baseline version. Any changes to analyzed files will trigger a delta analysis, upon which the tool will suggest what checks need to be re-done.

Depending on the usage scenario, the analysis reports are reported in one of two ways. The prototype has functionality for appending the report in textual format to a file. Alternatively, the result can be displayed in a web application, where the model is also rendered for visual checking.

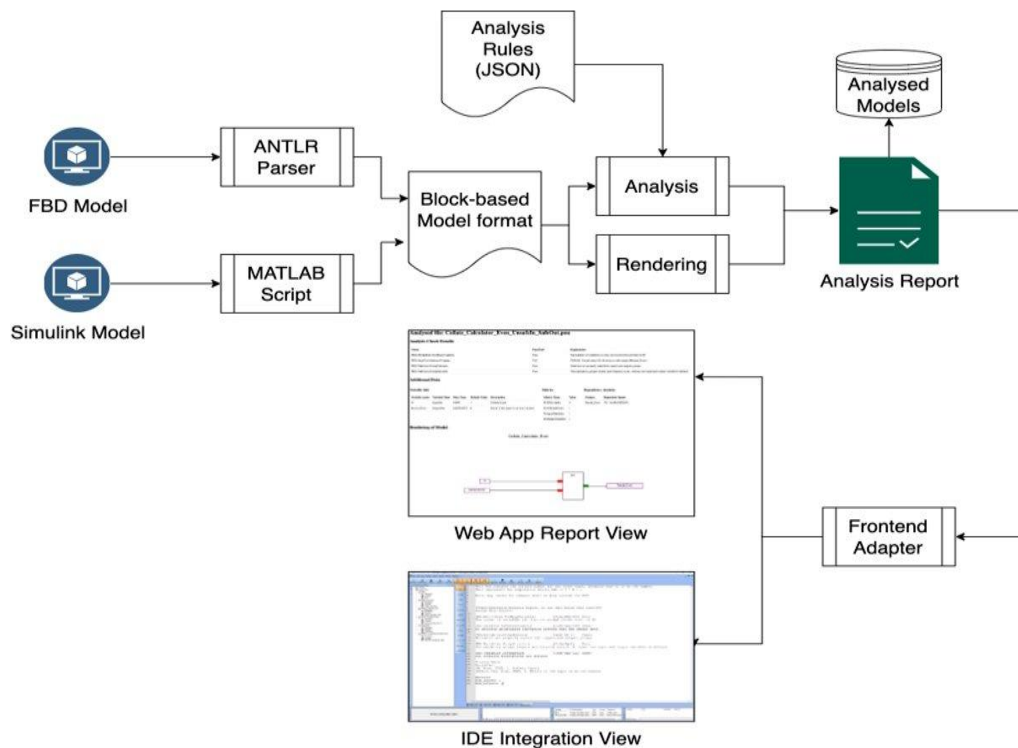


Figure 14 Overview of DRACONIS' toolchain. Block-based models are transformed into an abstract representation, then analyzed and stored in a database. The tool is usable either through its web application or directly connecting it to an IDE.

Improvement

DRACONIS has been initially validated on FBD models taken from the railway domain. The checks implemented have been shown to speed up validation during daily work. The work has focused on improving the usability for the tool as an addition to the daily work of end users.

Based on this, the following improvements have been made:

- A differentiation between *core* checks and user-configurable checks has been created. The core checks will continue to require some in-depth knowledge of how DRACONIS works but are also generating data that user-created checks can access. A context-specific Python programming language has been defined to create custom checks.
- To facilitate interoperability with other analyzers, the framework now allows connecting metrics files directly to analyzed models. With the current solution, the user can upload additional information as name-data pairs that are then attached to a given model. This allows the user to run heavy analyses offline or apply tools offering functionality not covered natively in DRACONIS and then upload the results for further review in the framework.

© 2024 SmartDelta Consortium

- The addition of programmatically uploading files to the framework allows for the scan of entire projects with minimal overhead for the user.

Purpose and Features

In addition to being a general analysis framework for block-based programs, DRACONIS is one of the tools in SmartDelta that automates the quality assurance process in an incremental development setting. Analyzing an intermediate representation allows the analysis of multiple types of models to be performed, and the analysis is presented in an accessible fashion. The tool has a dedicated mode for comparing variants, lists differences, and suggests follow-up quality assurance actions based on the changes.

Usage Instructions

1. Start the web server using the provided command. Use a suitable web browser to access the page.
2. From the start page, the user can either upload a new model or list all the uploaded models.
    - To add a new review model, it can either be uploaded manually through the GUI or programmatically as a POST request. A utility script has been prepared to automate this process, targeting a directory of files. Metrics-type data can be attached to models to facilitate interoperability with external tools. If uploaded manually, the model is analyzed, and the results are stored in the database and then presented to the user. To aid understanding, a rendering of the model is also presented.
3. In the list view, users may open any of the analyzed models to check the report in detail. Here, the user can also select a set of models to compare against each other. This will run the delta analysis between every pair of models selected and present the result in a report-friendly format.
4. In the report view, the user can review and comment on the reports, for instance re-classify the results of the different checks in the analysis core. An example can be seen in Figure 15.
5. Finally, the report, including metrics and review comments, can be exported and downloaded in Excel format.

Detailed usage instructions, including a setup guide and small test models, can be found at https://github.com/jean-malm-mdh/draconis.

Make and Download Report

Checks Failed

Check Name	Message	Review Status	Justification	Actions
FBD.DataFlow.SafenessProperty	Unsafe data ('IsOn_ST') flowing to safe output ('CanDoWork_ST') Unsafe data ('IsRunning_ST') flowing to safe output ('CanDoWork_ST') Unsafe data ('IsNotBusy_ST') flowing to safe output ('CanDoWork_ST')	Unviewed		Add Note · Justify Failure · Alert As False Positive
FBD.Variables.Initialization	The following variables are unnecessarily initialized to zero: On_ST	Unviewed		Add Note · Justify Failure · Alert As False Positive

Checks Passed

Check Name	Message	Review Status	Notes	Actions
FBD.MetricRule.TooManyVariables	The number of variables (4) does not exceed chosen limit of 40	Unviewed		Add Note · Alert As False Positive
FBD.Variables.GroupCohesion	Variables are properly sorted into inputs and outputs groups	Unviewed		Add Note · Alert As False Positive
FBD.Variables.GroupStructure	The mandatory groups (Inputs and Outputs) exists. At least one input and output variable is defined	Unviewed		Add Note · Alert As False Positive
FBD.Naming.Uniqueness	The variable names are suitably unique to be told apart by compilers	Unviewed		Add Note · Alert As False Positive

Figure 15. DRACONIS' report view. Failed and passed checks are grouped. For reviewing, some context-dependent actions are presented, where the user can review the result with an attached comment.

Case Studies

Within the project, DRACONIS has been applied to use cases provided by an industrial partner working in the railway domain. An internal version of DRACONIS was applied to 9 FBD models of varying complexity from a previously delivered project to study the speedup potential for checking design rules. Compared to the manual baseline captured by an experienced developer performing the same analysis, DRACONIS achieved a speedup of around 150 times when averaged over five executions.

A parallel check that encoded the detection of naming convention violations was developed using DRACONIS' features. This check was applied to interfaces of all models taken from the software baseline for a railway propulsion system developed using MATLAB Simulink. The resulting violations were presented on a signal level. Most of the violations represent some deviation between the official naming conventions and how interface signals are named in practice, mainly through unapproved abbreviations.

3.12. Test Effort Estimator

Method Description

The Test Effort Estimator (TEE) is a machine-learning-based tool designed to predict the amount of test effort required during incremental software development. It works by analyzing Java-based open-source repositories on GitHub and extracting features such as the number of changes in lines of code (CLOC), cyclomatic complexity, architectural smells, and more. The tool examines three levels of code change granularity—commits, pull requests, and releases—and applies classification and regression models to estimate the test effort regarding changes in the test code. The approach is validated on multiple Java repositories, and predictions are made for test effort, helping in resource allocation and improving the software development lifecycle.

Improvements

The primary improvement of this approach is its focus on incremental software development, as opposed to using decade-old datasets. This tool provides a more accurate and dynamic test effort estimation by utilizing modern repositories and real-time data extraction. Introducing various features such as CLOC, architectural metrics, and object-oriented smells increases the precision of estimates. Additionally, the multi-granularity approach at commit, pull request, and release levels enables a more refined understanding of testing effort needs, particularly when predicting at the release level where the tool achieves high accuracy.

Purpose and Features

The purpose of the Test Effort Estimator is to assist developers, project managers, and researchers in accurately predicting the test effort required for software projects. It aims to optimize the allocation of resources and reduce the risk of delays caused by underestimated testing efforts. It employs various regression and classification algorithms to estimate test effort; it estimates effort at the commit, pull request, and release levels, offering flexibility in project management; it uses up-to-date data from GitHub repositories instead of relying on outdated datasets and the tool generates a dataset specific to each project, improving estimation accuracy for that project.

© 2024 SmartDelta Consortium

Usage Instructions

- Select the Java-based repository from GitHub that you want to analyze. Ensure the repository contains sufficient commits, pull requests, and releases for meaningful analysis.
- Run the data extraction tool, which pulls relevant metrics from the repository, such as non-commenting source statements, cyclomatic complexity, and changes in lines of code.
- Once the data is extracted, the machine learning models are trained on this dataset to predict test effort. Both regression (for direct line estimation) and classification (for categorical effort levels) models are available.
- The models are validated by comparing predictions with actual changes in the test code. The results offer insights into resource allocation for future iterations.

Case Studies

The tool has been evaluated on three real-world open-source Java repositories:

- **Vaadin Platform**: A significant dataset with over 15,000 commits and thousands of pull requests and releases. Test effort estimation accuracy reached 97% for releases.
- **Netflix Conductor**: A moderate-sized repository used to demonstrate the tool's capability to estimate test effort at different levels of granularity.
- **Vaadin Flow**: The largest of the three repositories, where the tool achieved a high degree of accuracy in predicting test effort using machine learning models, particularly at the release level.

3.13. PyLC

As a contribution towards the WP3 goals of the SmartDelta project within MDU, an automated search-based PLC testing framework called PyLC has been developed. PyLC facilitates the unit testing of the PLC programs in Structured Text (ST) and Function Block Diagram (FBD) programming languages of the IEC61131-3 standard by automatically translating a PLC program to Python and generating test cases for it using the Pynguin[79] test generator tool in Python. PyLC has been introduced through two different academic papers, including [80] [81] and has been applied to more than 20 different real-world industrial use cases in industry.

Method Description

The proposed automated translation framework, PyLC, operates based on the automated parsing and analysis of a PLC program developed in the FBD language, represented as an PLCopen XML tree. Specifically, PyLC takes a POU as input, automatically extracts all the necessary information about the PLC program being translated, and stores it within a dictionary data structure in Python. This stored information is then utilized when PyLC automatically generates the executable Python code.

[79] https://pynguin.readthedocs.io/en/latest/index.html

[80] Ebrahimi Salari, Mikael, et al. "Pylc: A framework for transforming and validating plc software using python and pynguin test generator." Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing. 2023.

[81] Salari, Mikael Ebrahimi, et al. "Automating test generation of industrial control software through a plc-to-python translation framework and Pynguin." 2023 30th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2023.

© 2024 SmartDelta Consortium

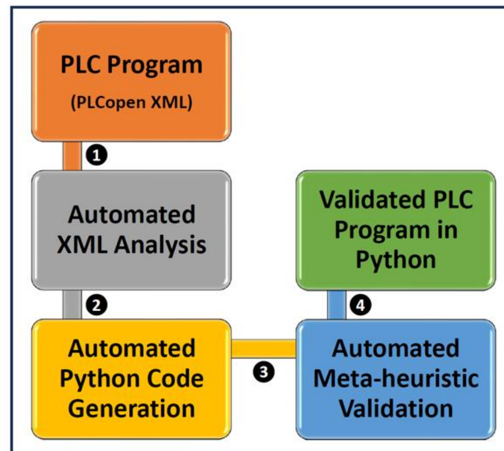


Figure 16 The overall translation process used in PyLC.

The overall translation process of PyLC is depicted in Figure 16. As illustrated, the initial step involves importing a PLC program developed in the FBD language as a PLCopen XML file. This is followed by automated parsing and analyzing of the XML tree to extract information about each POU and the blocks contained within (Step 1 in Figure 16). Subsequently, PyLC uses this extracted information to automatically generate executable Python code. This involves defining the required main and sub-functions and establishing the network between the existing Blocks from the imported PLC program (Step 2 in Figure 16). Next, PyLC employs the meta-heuristic automated unit testing techniques from Pyynguintool [8] to validate the translation (Step 3 in Figure 16). Lastly, the test cases generated by Pynguin are imported into the CODESYS IDE to be executed on the original PLC program using the CODESYS Test Manager tool. The PLC program's translation into Python is deemed valid if the generated test cases yield consistent results (Step 4 in Figure 16).

Improvements

To demonstrate the applicability and efficiency of the proposed translation framework, we translate ten different real-world PLC programs using the PyLC framework. Most of these PLC programs are used in the context of supervising industrial control systems developed by an automation company in Sweden. In contrast, the remaining ones are implemented in a nuclear plant.

All the PLC programs considered are developed in the FBD language and vary in size and complexity. We can draw several conclusions after applying the PyLC framework to these PLC programs and examining the information provided in Table 4. First, the FBD programs selected for translation encompass a variety of FBD block types, as detailed in Section II. This diversity highlights the extensive block support offered by PyLC. Second, the PyLC translation process is swift, with an average translation time of just 0.74 seconds. The size of the FBD program being translated, specifically the number of blocks can influence the translation efficiency. Larger PLC programs, like PRG4 and PRG7, tend to have marginally longer translation times. Overall, the collected results underline the potential and effectiveness of the PyLC translation framework in converting FBD-based PLC programs into executable Python code. This not only opens avenues for utilizing Python's capabilities within industrial automation but also offers a systematic approach to bridge the gap between PLC programming languages and general-purpose languages like Python. The PyLC framework demonstrates the capability for translating efficiently an array of industrial FBD programs, characterized by diverse block types, into Python code.

© 2024 SmartDelta Consortium

Table 4. Results of Test Generation using PyLC.

PRG Name	No. of Branches	No. of Blocks	Included Block Types	LOC in Python	Translation Time (s)
PRG1	12	4	LOG/TIM	80	0.7
PRG2	14	5	LOG/TIM/FB/SPEC	91	0.8
PRG3	6	3	LOG	50	0.5
PRG4	16	13	LOG/COMP	132	1.1
PRG5	3	1	MATH	22	0.4
PRG6	3	1	MATH	20	0.5
PRG7	16	13	LOG/COMP	100	1
PRG8	4	2	COMP	80	0.7
PRG9	8	7	LOG/COMP	77	0.6
PRG10	10	1	LOG	51	0.5

To assess the correctness and validity of the PyLC translation framework within an industrial setting, we translate ten real-world industrial PLC programs into Python, as detailed in the previous section. Subsequently, we utilize the Pynguin meta-heuristic test generator [8] to generate search-based test cases for the PLC programs translated using the PyLC framework. After collecting the test generation and execution results from Pynguin, we introduce the same test cases into the PLC environment for execution on the original PLC program within the CODESYS IDE. We then compare the test execution outcomes in both environments to determine the validity of the code translation from PLC to Python.

The results of the automated meta-heuristic testing for the included PLC programs using Pynguin are presented in Table 5. The evaluation of the translated Python code involved the instantiation of fitness functions, iteration counts, search time, mutant generation, and mutant survival rates. These metrics collectively provide insights into the efficiency, effectiveness, and coverage of the translation and testing processes. To ascertain the translation's accuracy, we test the generated Python code by utilizing meta-heuristic testing and record the test execution outcomes for each translated program using the Pynguin tool. Subsequently, we import these test cases into the PLC environment to execute them on the original PLC programs, aiming to discern congruence in their results. Upon automated execution of the acquired test cases on the original PLC programs (ranging from PRG1 to PRG10) via the CODESYS Test Manager, we observe that the test cases generated in the Python environment yield identical results when executed on the original PLC programs within the CODESYS IDE. This consistency shows the efficacy and correctness of the PLC-to-Python translations facilitated by our proposed PyLC framework. The PyLC translation framework, aided by Pynguin, generates test cases efficiently, attaining an average branch coverage of 98% across ten distinct real-world industrial PLC programs.

Table 5. Coverage and Mutation Results based on PyLC Test Generation.

PLC Program	Instantiated Fitness functions	Iterations	Search Time (s)	Generated Mutants	Surviving Mutants	Test cases	Verdict	Coverage	Covered Branches	Branchless code objects covered
PRG1	16	6042	1200	58	25	4	3/4	93.75	12	4/4
PRG2	19	5080	1200	43	25	4	4/4	94.74	13/14	5/5
PRG3	8	1	1	7	4	2	1/2	100	6/6	2/2
PRG4	24	1	4	23	15	9	5/9	100	16/16	8/8
PRG5	3	1	1	5	2	1	1/1	100	3/3	0/0
PRG6	3	1	1	5	5	1	1/1	100	3/3	0/0
PRG7	24	1	3	23	17	4	4/4	100	16/16	8/8
PRG8	6	1	1	6	3	2	2/2	100	4/4	2/2
PRG9	13	1	2	12	7	4	3/4	100	8/8	5/5
PRG10	12	1	1	5	2	6	6/6	100	10/10	2/2

In future work, we aim to conduct a more thorough examination of PyLC's scalability by applying it to even more sophisticated real-world PLC programs. Other future work directions include adding

support for the translation of PLC programs into the ST language and enhancing the translation validation mechanism of PyLC with a Python static verifier.

Purpose and Features

PyLC tool is developed to automatically translate a PLC program to Python to enable automated meta-heuristic testing for PLC programs. PyLC supports the PLC programs in both FBD and ST languages and can 1) import a PLC program as an OpenPLC XML file, 2) translate the imported PLC program to executable Python code automatically, and 3) validate the correctness of the translation through a search-based testing tool called Pynguin. Based on the results we gathered after applying PyLC to different real-world industrial use cases, we confirm that PyLC is fast in both automated translation and automated testing of PLC programs. PyLC enables testing PLC programs through five different search-based algorithms, including MIO, MOSA, DYNAMOSA, WHOLE SUITE, and RANDOM.

PyLC supports different FBD blocks in the PLC program under translation including Logic Blocks (LOG), Comparator Blocks (COMP), Timers and Counter Blocks (TIM), Mathematical Blocks (MATH), Function Blocks (FB), Special Blocks (SPC).

Usage Instructions

PyLC can be used to import a PLC program as a PLCOpen XML file which is supported by most of the PLC IDEs in the market. PyLC consists of four Python scripts that are chained to each other. One can use PyLC by importing a PLC program as an XML file. The PyLC source code is available in a GitHub repository[82] and can be downloaded and used by the user. The only consideration is the current version of the PyLC only supports the PLC programs in FBD language.

Case Studies

PyLC has been applied to different real-world industrial PLC programs of two big automation companies in Sweden and South Korea, which were developed in FBD and ST programming languages. The latest version of the PyLC tool, which is limited to being used with PLC programs in FBD language, has been applied to ten different FBD programs. Table 6 depicts the list of these ten FBD programs, as well as their abstract specifications and translation time.

Table 6. Details of PLC Programs Considered in the Case Study.

PRG Name	No. of Branches	No. of Blocks	Included Block Types	LOC in Python	Translation Time (s)
PRG1	12	4	LOG/TIM	80	0.7
PRG2	14	5	LOG/TIM/FB/SPEC	91	0.8
PRG3	6	3	LOG	50	0.5
PRG4	16	13	LOG/COMP	132	1.1
PRG5	3	1	MATH	22	0.4
PRG6	3	1	MATH	20	0.5
PRG7	16	13	LOG/COMP	100	1
PRG8	4	2	COMP	80	0.7
PRG9	8	7	LOG/COMP	77	0.6
PRG10	10	1	LOG	51	0.5

[82] https://github.com/VeriDevOps/PyLC

© 2024 SmartDelta Consortium

3.14. GW2UPPAAL

Method Description

The GW2UPPAAL tool integrates Model-Based Testing (MBT) and Model Checking to create a unified approach for verifying and testing system models. It transforms behavioral models created using GraphWalker (GW), a model-based testing tool, into UPPAAL-timed automata for formal analysis. This transformation automatically verifies properties such as reachability and deadlock freedom using UPPAAL's model checker. The tool automates the generation of UPPAAL models and queries, allowing users to validate the correctness of models before generating and executing test cases. One key feature is variant handling, where the tool supports multiple model variations by detecting deltas (i.e., differences between model versions) and applying these changes in the transformation process. This allows users to manage different versions or configurations of a system efficiently.

Improvements

GW2UPPAAL enhances the MBT process by adding automated formal verification and variant handling. The tool tracks delta changes made between model versions, allowing for efficient management of different system configurations and faster verification of updated models. This significantly reduces the time and effort needed for manual adjustments when multiple versions or configurations of a model need to be tested. By integrating formal verification through UPPAAL, GW2UPPAAL allows developers to check reachability, deadlock freedom, and other properties early, ensuring the test model's accuracy before generating test cases. Additionally, handling deltas provides better support for evolving systems, as changes can be isolated and verified without rechecking the entire model.

Purpose and Features

GW2UPPAAL is designed to streamline the MBT process by incorporating automated formal verification with enhanced variant handling. It helps developers and testers manage different versions or configurations of a system by detecting and applying deltas, and incremental changes between models, ensuring that each variant is accurately transformed and verified. The user can develop the system model using GraphWalker, including handling variants by specifying different configurations or model versions. Export the model as a JSON file. In addition, the user can use GW2UPPAAL to transform the GraphWalker model, and the tool will detect any deltas from previous versions, applying only the necessary updates in the transformation process and executing the UPPAAL model using the "verifyta" tool, automatically checking for reachability and deadlock properties while accounting for model changes or variants. Based on the verification results, make necessary adjustments to the model in GraphWalker, especially for deltas between versions, before generating new test cases. In the end, the user can create test cases for each model variant, ensuring that the changes between versions are accurately reflected in the test cases.

Usage Instructions

- Create a GraphWalker Model: Design the system model using GraphWalker and export it in JSON format.
- Run GW2UPPAAL: Provide the JSON file location and use the GW2UPPAAL tool to transform the GraphWalker model into UPPAAL's XML format.
- Verify the Model: Execute the generated UPPAAL model using UPPAAL's "verifyta" tool, which automatically checks for reachability and deadlock properties.
- Analyze Results: Use the verification results to make any necessary adjustments to the original model in GraphWalker.

- Generate Test Cases: Once the model is verified and refined, generate test cases based on the transformed and verified model.

Case Studies

GW2UPPAAL was tested on several models, demonstrating its capability to manage model variants and handle deltas. First, a model with multiple configurations can be handled, where deltas between variants were detected and handled seamlessly, reducing the time for transformation and verification to just 0.202 seconds. A larger model with evolving versions and multiple deltas can be used, where the tool successfully tracked and applied changes, significantly reducing manual effort. A complex system with multiple diagrams can be handled, where shared vertices across variants are flattened into a single UPPAAL model. The tool's delta handling ensured that updates to individual diagrams were applied only where needed. By efficiently handling variants and deltas, GW2UPPAAL reduces transformation time and ensures accurate verification of evolving systems, providing robust support for real-world model-based testing and formal analysis.

3.15. Testing Utilizing EARS Notation in PLC Programs

The Easy Approach to Requirements Syntax (EARS) addresses the inherent imprecision of natural language requirements concerning potential ambiguity and lack of accuracy. This paper investigates requirements specification using EARS and specification-based testing of embedded software written in the IEC 61131-3 language, a programming standard for developing Programmable Logic Controllers (PLC). Further, we study, utilizing an experiment, how human participants translate natural language requirements into EARS and how they use the latter to test PLC software. We report our observations during the experiments, including the type of EARS patterns participants uses to structure natural language requirements and challenges during the specification phase, and present the results of testing based on EARS-formalized requirements in real-world industrial settings.

Method Description

To evaluate the applicability of using EARS semi-structured syntax when creating test cases for PLC programs, we used three programs that implement the behavior stated in the three provided natural language requirements used in this experiment. All these three PLC programs are developed in CODESYS IDE using the ST programming language, whether by the authors or derived from existing industrial cases. This paper refers to these programs as PRG1, PRG2, and PRG3. We used the concretization steps of the EARS expressions. This happens by mapping the system response, condition, and events to the actual implementation in PLC. This contains information about the implementation elements of a system and its interfaces. An engineer needs to consider this information and identify the given signals and their characteristics. In this way, we define a set of signals related to the feature under test. In these cases, the next step for the selected requirements would be to design test cases to show that the requirement has been met.

The process of transforming system requirements into EARS requirements for PLC testing is a structured approach, as shown in Figure 17. It begins with the requirement analysis phase, where system requirements and design documentation are used. Next, the information collected is formalized into abstract EARS requirements in the construction phase. These requirements use logical names for entities, making communication with requirements engineers easier, though they are not directly evaluable at this stage. The implementation analysis phase then involves mapping these abstract entities to actual implementation elements by analyzing design documents. This includes identifying the signals or events corresponding to the abstract entities defined earlier. The process then proceeds to the EARS requirement concretization phase, where the abstract EARS requirements are transformed into concrete counterparts using actual signals and events from the

© 2024 SmartDelta Consortium

system. This results in a set of concrete EARS requirements ready for test creation and execution, as shown in Table 7.

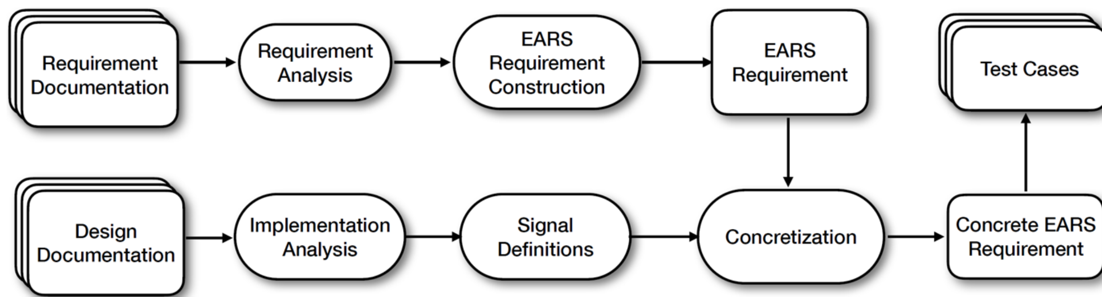


Figure 17. EARS Overall Process.

Table 7. Mapping between EARS Requirements and Test Cases.

Req.	**EARS Requirements**	**Concretized EARS Requirements**	**Example Test Cases**
RI1	The <user account system> shall <identify the user> If <the user is not identified> then <user account system> shall <alert>	if <uniqueID=FALSE> then <UniqueUserAccount> shall <Result_Unique=FALSE>	**Description:** Verify system alerts when a user is not identified. **Steps:** 1. Attempt to identify a user with invalid credentials. 2. Check uniqueID. 3. Verify alert is generated.
RI2	When <malware is detected> the <system> shall <warn the user>	When <NormalActivity ≠ MaliciousActivity> the <MalwareDetection> shall <MalwareDetected=TRUE>	**Description:** Verify system warns user when malware is detected. **Steps:** 1. Simulate malicious activity. 2. Check activity status. 3. Verify warning is issued.
RI3	When <the device is authorised> the <system> shall <grant access to the device>	When <found=TRUE> the <SearchID> shall <ConnectionAllowed=TRUE>	**Description:** Verify system grants access to an authorized device. **Steps:** 1. Simulate an authorized device. 2. Check device status. 3. Verify access is granted.

To create test cases for requirement RI1, which involves user identification and alerting, we set up a scenario where the system attempts to identify a user with invalid credentials, ensuring the *uniqueID* is FALSE. We then verify that the system generates an alert by checking if *UniqueUserAccount* results in *ResultUnique*=FALSE. This involves preparing the system to identify users, attempting identification with invalid data, and checking for the appropriate alert response. To create a test case for RI2, we simulate malicious activity in the system, ensuring that *NormalActivity* does not equal *MaliciousActivity*. We then verify that the *MalwareDetection* component sets *MalwareDetected* to TRUE and that the system issues a warning to the user. This involves running the system to monitor for malware, simulating malicious activity, and checking the detection and the warning response. For requirement RI3, to create a test case, we simulate the presence of an authorized device, ensuring that it is found (found=TRUE). We then verify that the *SearchID* component sets *ConnectionAllowed* to TRUE and that the system grants access to the device. This involves configuring the system to authorize devices, simulating the authorization of a device, and checking that access is granted correctly.

After generating the EARS-based test cases for each program, we execute them automatically using the CODESYS test automation framework, CODESYS Test Manager. The final step in this

© 2024 SmartDelta Consortium

methodology is to manually compare the actual output with the expected output to observe whether the program works as expected.

The methodology we propose for using EARS-based testing in real-world industrial settings consists of seven steps, as shown in Figure 18. The first step is to extract the functional requirements from the real-world PLC program (step 1). The purpose and process of functional requirements extraction in the context of this study were necessary for the experiment, as we lacked requirements at this level. The second step is to have a team of industrial PLC engineers evaluate the validity of the functional requirements (step 2). The next step is to transform the NL requirements into EARS requirements to mitigate the potential ambiguity and increase the clarity of the extracted requirements for the tester (step 3). As the next step, the EARS requirements must be concretized to facilitate the test generation by converting the Inputs/Outputs (I/O) into signals (step 4). After having the concretized test cases, it is time to manually generate test cases via the pre-defined Test Actions inside the CODESYS Test Manager tool (step 5). The next step is to automatically execute the test cases on the PLC program using the CODESYS Test Manager tool (step 6). The final step in this methodology is to enhance the generated test report of CODESYS Test Manager by measuring the code coverage automatically using the CODESYS Profiler tool and checking the results (step 7).

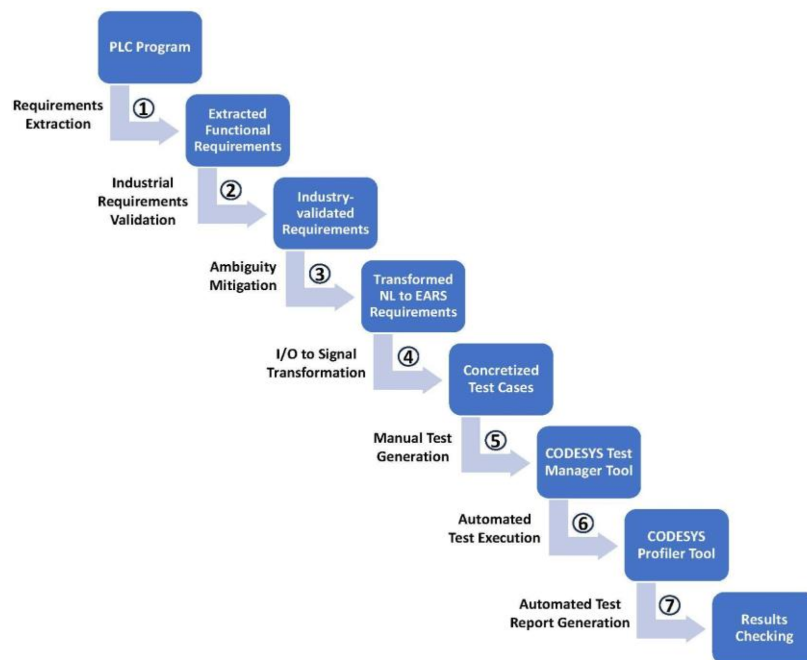


Figure 18. EARS Process From PLC Programs to Results Checking.

Improvements

We evaluated the application of EARS syntax in creating test cases for PLC programs, identifying its strengths, challenges, and potential advantages over traditional testing practices. EARS syntax was assessed in developing test cases for multiple PLC programs in CODESYS IDE. EARS-based test cases, executed through the CODESYS Test Manager, demonstrated good execution times, efficiently transforming natural language requirements into testable cases and ensuring compliance with specified requirements.

We also analyzed the types of EARS templates employed in writing requirements. Participants primarily used the ubiquitous template for core requirements, with additional use of event-driven, state-driven, and unwanted behavior templates. Key challenges included managing unclear initial requirements, selecting appropriate templates, and balancing positive and negative behaviors,

especially in “shall not” statements. The optional feature template was found to be unnecessary. Challenges in requirements specification and test creation using EARS were highlighted. Participants encountered difficulties in translating ambiguous requirements, selecting suitable templates, and defining adequate test coverage for positive and negative behaviors. These complexities underscore the intricacies of converting natural language requirements into structured EARS syntax for precise test case generation.

A comparison between EARS-based and traditional industrial testing for one program demonstrated that EARS-based testing improves efficiency, reducing complexity with automated, clear, and comprehensive test specifications. This approach enhances clarity in functional requirements, leading to more effective testing outcomes. In summary, these findings demonstrate EARS notation's applicability, efficiency, and advantages in requirements engineering and test generation for PLC programs while identifying areas for further improvement.

Purpose and Features

In this work, we have proposed using requirements engineering and testing using EARS notation for PLC systems. In requirements engineering, most participants preferred the EARS ubiquitous pattern for transforming some requirements from NL to the EARS syntax. In contrast, unwanted behavior and event-driven patterns were the most popular types for other requirement transformations. It was observed that different individuals used different EARS patterns to transform the same requirement based on their interpretation, showing an acceptable level of flexibility in EARS syntax. In the testing part, we assessed using EARS patterns for PLC testing in two phases. Initially, we executed EARS-based test cases on three PLC programs written in the ST language, which were developed based on the requirements included in our study. Subsequently, we introduced an EARS-based testing methodology to real-world industrial PLC programs. The results from these tests and the subsequent comparison with traditional PLC testing methods indicate that EARS-generated requirement-based test cases for PLC programs are effective and offer an accessible means for PLC testers to express test specifications. The proposed testing method covers all PLC programs in IEC 61131-3 standard using the help of CODESYS IDE and its testing tools, such as CODESYS Test Manager and CODESYS Profiler.

Usage Instructions

The user needs to follow the proposed step-by-step procedure to generate PLC unit-level test cases from requirements in natural language. The procedure is simple but effective and can be easily mimicked based on the example PLC programs we provided in the related publication. It is worth mentioning that most of these steps need to be done manually. However, some steps in this pipeline, such as test execution and measuring the code coverage, are automated using the CODESYS Test Manager and CODESYS Profiler tools.

Case Studies

The proposed methodologies in this work have been applied to four different real-world PLC programs in the context of supervising the port cranes. The results imply the applicability and efficiency of the proposed methodologies regarding real-world PLC testing.

© 2024 SmartDelta Consortium

3.16. Jazure

Jazure is a powerful tool designed to automate the selection and linkage of work item IDs with pull requests in environments using Azure DevOps and Jira. By seamlessly integrating these platforms, Jazure ensures that work items are consistently up-to-date and adequately associated with code changes.

The tool periodically syncs Jira work items to the Azure DevOps work item board, analyzes the code changes within pull requests, and intelligently identifies relevant work item IDs (as shown in Figure 19). It enforces branch policies that mandate work item linkage before any code can be merged into the master branch, thereby enhancing traceability and streamlining the software development process.

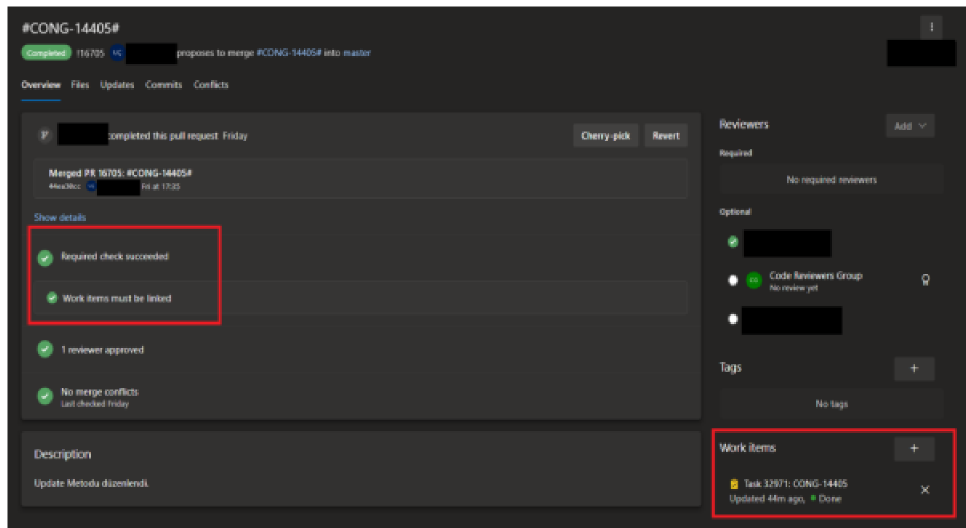


Figure 19. Overview of Jazure.

Improvements

Jazure significantly enhances the software development workflow by automating a traditionally manual and error-prone task. The tool can automate the linkage of work items to pull requests; Jazure removes the risk of human oversight, ensuring that every pull request is properly associated with its corresponding work items. The tool enforces branch policies that prevent merging without linked work items, achieving a 100% success rate in establishing relationships between work items and pull requests. Jazure bridges the gap between Jira and Azure DevOps, synchronizing work items across both platforms and fostering better collaboration and consistency. Developers can focus more on coding and less on administrative tasks, as Jazure automatically handles the tedious aspects of work item management.

Purpose and Features

Jazure is designed to improve efficiency and traceability in the software development lifecycle. It analyzes pull request code changes to select and link relevant work item IDs intelligently. Periodically syncs Jira work items to Azure DevOps, ensuring consistency and up-to-date information across both platforms. It implements mandatory work item linkage policies for merging into the master branch, enhancing code quality and traceability and operates on a scheduled basis to sync and update work items without manual intervention. Ultimately, it improves clarity in software evolution by ensuring all pull requests are properly linked to their corresponding work items.

© 2024 SmartDelta Consortium

Usage Instructions

- Configuration: Set up Jazure by providing the necessary authentication details for Azure DevOps and Jira. Then, configure the synchronization settings, including the frequency of syncing and specific projects or boards to monitor.

- Scheduled Syncing: Jazure will operate on a scheduled basis, periodically syncing Jira work items to the Azure DevOps work item board to ensure all information is current.

- Branch Policy Enforcement: Define and enable branch policies in Azure DevOps that require a linked work item before a pull request can be merged into the master branch.

- Automated Work Item Linking: As developers create pull requests, Jazure analyzes the code changes and automatically identifies and links the relevant work item IDs.

- Review and Merge: Review the pull requests in Azure DevOps. The linked work items will be displayed automatically. Proceed with code reviews and merge processes, confident that all work items are accurately associated.

Case Studies

Case Study 1: Enhancing Traceability for Team Connecta. Team Connecta faced challenges with developers needing to remember to link work items to pull requests, leading to gaps in traceability and project management. After implementing Jazure, They achieved a 100% success rate in linking work items to pull requests. Traceability improved significantly, allowing project managers to track progress more effectively. The team saved time previously spent on manual checks and corrections.

Case Study 2: Synchronizing Jira and Azure DevOps for Team Mars. Mars used Jira for issue tracking and Azure DevOps for code management but needed help with keeping work items consistent across platforms. With Jazure: Jira work items were automatically synced to Azure DevOps, ensuring consistency. Cross-functional teams collaborated more efficiently, with all members accessing up-to-date work item information. The synchronization reduced confusion and duplicate efforts caused by inconsistent data.

Case Study 3: Enforcing Quality Standards for Team ArGpt The ArGpt Team wanted to enforce stricter code quality and traceability standards but found it challenging to do so manually. By adopting Jazure, they implemented branch policies that prevented merges without linked work items. The quality of code integration improved, as all changes were properly reviewed and associated with documented work items. The team noticed a decrease in integration issues and an improvement in overall project documentation.

3.17. Smellyzer

Method Description

Smellyzer is a professional tool developed to detect software process smells related to Code Review (CR) and Bug Tracking (BT) processes in software projects. The tool leverages CR taxonomies and BT smells established in prior studies to evaluate the processes followed in software projects.

It analyzes project data from repositories and issue-tracking systems like GitHub, Jira, Azure DevOps, and GitLab. Smellyzer detects non-optimal practices (process smells) and presents the

© 2024 SmartDelta Consortium

results using multiple visualization methods. The tool is structured into several subsystems to maintain modularity and low coupling, allowing easy interaction and data processing for detailed process analysis.

Improvements

Participants of the case studies suggested several improvements to enhance Smellyzer's functionality:

- Filtering options: Users requested more filters, such as by branch or status, to allow focused analysis.
- Positive highlights: Suggestions included showing smelly pull requests and those without smells to encourage good practices.
- Group-level insights: Users proposed presenting cumulative smells for groups of developers to enable team-based process improvements.
- Customization: Expanding the detection mechanisms to account for specific company practices (e.g., new module development or team-based reviewing).
- Project-wide summaries: Users wanted reports on process smells across entire modules or applications rather than individual bug or pull request data.

Purpose and Features

Smellyzer is designed to identify and address process smells in software development to improve adherence to best practices in CR and BT processes. It helps detect non-ideal behaviors that can lead to delays, inefficiencies, and reduced code quality, providing actionable insights to rectify such practices.

Key Features:

- Detection of CR and BT smells: Smellyzer analyzes software repositories for process smells using predefined taxonomies.
- Visualization methods: The tool offers various visualization techniques to highlight smell occurrences and trends over time, such as graphs showing the number of smells per year or by the developer.
- Customization: Users can clean datasets, merge developer identities across platforms, and set thresholds for specific attributes.
- Data export: The tool allows users to download analysis results in formats like JSON for further processing or reporting.

Usage Instructions

- Project Creation: Users log into Smellyzer and create a new project by entering the project name and specifying the data sources (e.g., repositories, issue tracking systems).
- Customization: Users can configure the analysis by merging duplicate developer identities, excluding invalid names (e.g., bot accounts), and setting specific thresholds for smells.
- Running an Analysis: After setting up the project, users initiate the analysis by selecting the data sources to be evaluated. Smellyzer supports both one-time and continuous analyses.
- Viewing Results: Once the analysis is complete, users can view the results in multiple formats, including smell counts by year, developer, or per pull request/bug report. Visualization tools are available to help explore the results.
- Data Export: Users can export analysis results for offline review or reporting.

Case Studies

Smellyzer has been tested in three case studies involving small-scale, medium-scale, and large-scale proprietary companies:

- Pilot Study 1 (Small-Scale Company): A software analytics company used Smellyzer to analyze one of its projects hosted on Bitbucket and Jira. This helped assess how small teams manage CR and BT processes.
- Pilot Study 2 (Medium-Scale Company): A healthcare software company used Smellyzer on a GitHub-hosted project, which provided insights into process smells in a mid-size industrial environment.
- Main Study (Large-Scale Company): We analyzed one of Arcelik's projects, which involved six Agile teams using Azure DevOps for code reviews and Jira for issue tracking. The study showed varying levels of CR and BT smells and how they evolved, with higher occurrences in the early stages of development when the product was less mature.

3.18. Relink

Method Description

ReLink is a tool developed to enhance the traceability between pull requests (PRs) and issues by repairing historically missing links using data science algorithms. Unlike tools that enforce link creation during the PR process, such as Jazure, ReLink retrospectively predicts connections between PRs and issues. It analyzes textual similarities and applies heuristic guidelines to identify absent links, assigning a confidence score ranging from 0 to 100 to each potential connection. A higher score indicates a stronger likelihood of a valid association. ReLink's web-based interface provides visual aids to facilitate reviewing and confirming these predicted links, making it easier for practitioners to establish accurate traceability in their software projects.

Improvements

ReLink significantly improves the accuracy and effectiveness of software metrics by ensuring that all PRs are properly linked to their corresponding issues, even if the links were initially missing. Key improvements include:

- Enhanced Traceability: By recovering missing links between PRs and issues, ReLink ensures that historical data is complete, which is crucial for accurate software metrics and analytics.
- Data-Driven Predictions: Utilizes text similarity analysis and heuristic rules to predict connections, offering a more reliable method than manual reconstruction.
- High Accuracy: Achieved a Mean Reciprocal Rank (MRR) of 0.84 and high top-N accuracies (Top-1: 0.83, Top-3: 0.85, Top-5: 0.86) in the Connecta project, demonstrating its proficiency in correctly associating PRs with relevant issues.
- User-Friendly Interface: The web-based platform provides visual aids and confidence scores, making it easier for users to validate and confirm the predicted links.

Purpose and Features

ReLink is designed to improve the traceability and integrity of software development records by repairing missing links between pull requests and issues. Its key features include:

- Historical Link Recovery: Analyzes existing data to predict and recover missing PR-issue links in past records.
- Confidence Scoring: Based on text similarity and heuristics, it assigns a confidence score (0-100) to each predicted link, indicating the likelihood of a valid connection.
- Visual Aids: Offers a web-based interface with visual representations to help users easily review and confirm predicted links.

- Consistent Prediction Performance: Delivers high accuracy in link prediction across different projects, ensuring reliable results.
- Data Science Algorithms: Employs advanced algorithms to analyze code commits, PR descriptions, issue reports, and other textual data to find correlations.

Usage Instructions

For project management, we begin by setting up ReLink. This involves installing it on your server or accessing it through a web-based interface. Configure ReLink to connect with your project's repositories and issue tracking systems, ensuring access to all necessary project information. Next, import historical data to ReLink. This includes pulling in existing data on pull requests, issues, and any established links within the project. ReLink then uses its algorithms to identify and predict missing links between pull requests and issues, revealing previously overlooked connections.

Once the analysis is complete, head to the ReLink web interface to review the predicted links. Each prediction is accompanied by a confidence score and visual aids, helping you assess the relevance and accuracy of each suggested link. With this information at hand, we need to validate the results. Confirm or reject each suggested link, relying on the provided data and your expertise. When you confirm a link, ReLink will automatically update the project's records to include the new connection, ensuring an up-to-date linkage between project elements. Finally, utilize the enriched dataset to generate accurate software metrics, analytics, and reports.

3.19. Telemetry Anomaly Analyzer (TAA)

Purpose and Overview

One of the most significant challenges engineering and operations teams face in software-as-a-service environments is detecting and responding to anomalies in large volumes of telemetry data generated by complex distributed systems. The Telemetry Anomaly Analyzer is designed to address this challenge. This tool provides an automated, scalable way to detect and visualize anomalies in telemetry data collected from distributed systems. By utilizing OpenTelemetry-compatible telemetry data, the analyzer applies machine learning models to identify irregular patterns, facilitating rapid detection of performance bottlenecks, security threats, or potential failures. Additionally, the tool generates rich, interactive dashboards that help teams monitor, investigate, and respond to these anomalies in real time. As a holistic tools architecture developed on a mainstream cloud computing platform, the Telemetry Anomaly Analyzer integrates with several cloud services to handle data ingestion, processing, anomaly detection, and visualization.

Features

The tool inputs telemetry data, including traces, logs, and metrics, in formats compatible with the OpenTelemetry standard. This makes it well-suited for cloud-native architectures. It supports various telemetry sources, including distributed systems and microservices deployed across multiple cloud platforms.

At its core, the tool features an anomaly detection engine powered by interchangeable machine learning models trained to identify irregularities in system behavior. It supports unsupervised anomaly detection techniques, capable of learning patterns from historical data and identifying deviations in near real-time. Detected anomalies are presented through interactive dashboards that offer insights and visual representations such as heatmaps and timelines. These dashboards enable users to go down into specific events or system components to identify the root cause of anomalies.

The tool is designed for cloud-native environments, leveraging mainstream cloud platforms and utilizing managed services like data warehouses for data storage. It supports both custom and off-the-shelf options for model training. In addition to detecting anomalies, the tool can trigger alerts via multiple channels, such as email or Slack, based on predefined thresholds.

The architecture supports customizing machine learning models and thresholds, making it adaptable to diverse use cases. Users can extend the platform by adding more data sources, integrating custom anomaly detection models, and incorporating new visualization widgets for dashboards or using alternative visualization tools.

Usage

The tool is a set of cloud services and components and has a custom setup for each environment in which it would be used. The tool is a set of deployed services that work continuously and automatically to produce up-to-date data, visualizations and alerts in the following way:

1. Ingestion of Telemetry Data
    a. Once deployed, the tool begins collecting telemetry data from the configured sources and stores them in short-term telemetry data and a data warehouse for further analysis.
2. Machine Learning Pipeline Execution
    a. Based on the configured triggers, the tool's ML models analyze the incoming data, detecting any anomalies based on training data.
    b. Anomalies are classified, and possible alerts are generated based on severity levels defined in the configuration.
3. Visualizing Results
    a. Detected anomalies are displayed in the dashboards and are accessible via a web interface hosted on a visualization platform.
    b. Dashboards include various visual components, such as timelines, system heatmaps, and trace trees, that allow users to see how the anomaly propagates through the system.
4. Alerting
    a. When an anomaly is detected, an alert is sent to the configured communication channels.

4. A Methodology for Delta-Aware Quality Assurance

The methodology focuses on a holistic view of the entire development lifecycle, from initial planning to maintenance. It emphasizes the interconnection between various elements, including people, roles, skills, teams, tools, techniques, processes, activities, milestones, work products, standards, quality measures, and team values.

Here's a breakdown of the core components of this methodology:

- People and Roles: The methodology acknowledges that specific skills and personalities are vital in filling various project roles. This includes designers and programmers and all roles funded by the project, offering a broader view.
- Teams and Skills: The methodology involves working in different types of teams, where combining various skills leads to producing work products.
- Tools and Techniques: Specialized tools and techniques are employed to construct work products, following specific standards and quality criteria.
- Processes and Activities: The teams engage in various activities within the project. This structured approach ensures that all aspects of the project are considered and carried out systematically.

© 2024 SmartDelta Consortium

- Milestones: These checkpoints help measure progress and make necessary adjustments.
- Work Products, Standards, and Quality Measures: Work products must adhere to selected standards, including notations like drawing and programming languages, policies like incremental development, and team-determined conventions. Quality measures ensure that the output meets the intended criteria.
- Team Values: Team values are vital in shaping how the team communicates and operates. Aligning these values with the people's values and the process promotes a cohesive and productive environment. Different values influence different methodologies.

The different dimensions key to the SmartDelta software methodology shape the overall quality assurance framework. Requirements Management and Analysis are used in defining, analyzing, and managing requirements, ensuring that the software aligns with business goals and creates a solid foundation for development. Test Design and Selection focus on creating and choosing test cases for robust system inspection, while Test Amplification enhances existing test cases to increase coverage. Monitoring and Anomaly Detection are responsible for observing the system for abnormal behavior, and Static Analysis examines software without executing it to identify potential code issues. Integration of Functional and Extra-Functional Testing ensures that all aspects of the system's quality are examined, including often neglected areas like resource consumption and security. Delta-aware Testing focuses on fault detection and efficiency by allowing for adaptive testing strategies. Finally, Visual Representation and Mapping provide an accessible overview of the methodology's components, aiding communication and understanding between stakeholders. Together, these dimensions form a framework that addresses both functional and non-functional aspects of software quality. Together, these dimensions form a methodology where each aspect complements the others to ensure comprehensive quality assurance. The SmartDelta methodology uses a diverse array of practices, balancing the need for rigorous testing with efficiency and adaptability, making it suitable for complex and evolving software projects.

The overall methodology focuses on test design, automated test generation, and analysis of functional criteria and extra-functional properties (e.g., resource consumption, security, performance) based on the model specification of the corresponding requirements. We identify invalid states and conflicting requirements and use this type of information to create negative tests, which attempt to force the system to enter invalid states and can uncover bugs and vulnerabilities that positive test cases may not detect. One improvement of this methodology outlined in Figure 20 would be to define guidelines and a format for testers to specify test scenarios that evaluate combined functional and extra-functional properties, which are frequently neglected areas of testing.

© 2024 SmartDelta Consortium

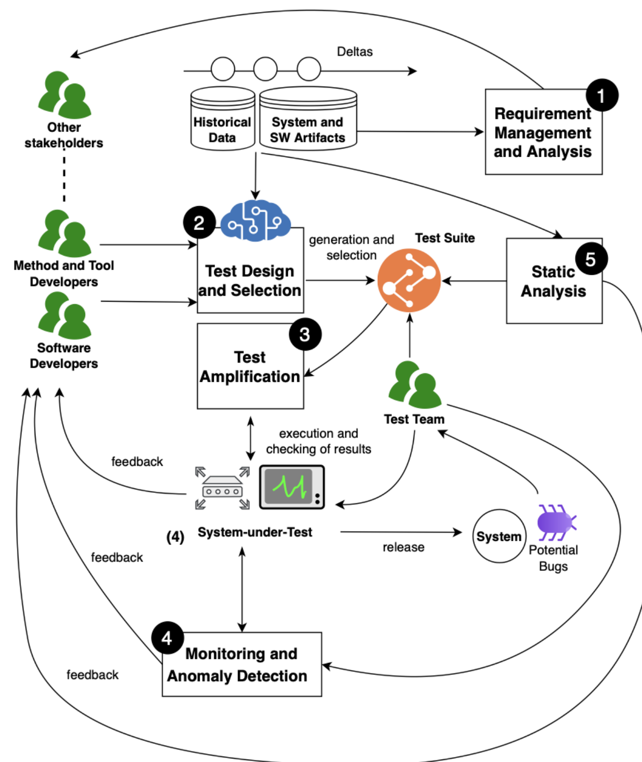


Figure 20 Overview of the SmartDelta Methodology for Quality Assurance.

Figure 20 depicts the overall concept of the methodology and Table 8 depicts the different dimensions of this methodology and the mapping to the different individual components. These can be seen as toolkits to enable users to interact with requirement management and analysis, the test generation and selection process, as well as static analysis and monitoring, as an enabler for delta-aware testing with better performance in terms of fault detection and efficiency.

Requirements Management and Analysis (1). The first dimension of the SmartDelta quality assurance methodology focuses on analyzing and managing requirements to identify conflicts, define system boundaries, and establish quality standards. According to SWEBOK [83], this includes validation (ensuring requirements meet goals) and verification (ensuring proper implementation). To clarify, requirements exist at different levels: business requirements define overarching goals, system requirements specify overall system needs and software requirements detail the behavior of software components. In industries like automotive, system requirements drive hardware and software specifications, which are further divided into component- and unit-level requirements. The SmartDelta methodology addresses system and software requirements, ensuring alignment with business objectives through robust validation and verification processes.

The following steps are used in the requirements management and analysis dimensions:

- Analyzing Requirements: This process involves a detailed examination of each requirement to ensure clarity, feasibility, and relevance. Analysts scrutinize the requirements to understand their implications on the system's design and functionality. This stage often involves stakeholders to ensure that the requirements truly reflect their needs and expectations.
- Managing Requirements Conflicts: Conflicts in requirements can arise due to various factors like stakeholder disagreements, technical constraints, or budget limitations. SmartDelta emphasizes identifying these conflicts early in the development process. This approach

[83] https://www.computer.org/education/bodies-of-knowledge/software-engineering

© 2024 SmartDelta Consortium

involves reconciling differing viewpoints, adjusting requirements, or prioritizing them to mitigate conflicts effectively.

- Defining System Boundaries: Understanding the bounds of the system is critical. It involves delineating what the system will and will not do. This clarity helps in preventing scope creep and ensures that the development efforts are focused and aligned with the intended purpose of the software.
- Creation of System Requirements: This step involves translating the gathered requirements into a structured set of system requirements. These requirements should be specific, measurable, achievable, relevant, and time-bound. They serve as a blueprint for the development team.
- Validation of Requirements: Validation focuses on ensuring that the requirements accurately reflect the needs and expectations of stakeholders and align with the business goals. This process involves reviewing the finalized requirements with stakeholders. Unlike managing requirement conflicts, which involve resolving disagreements or inconsistencies among stakeholders during the elicitation and negotiation phases, validation occurs after conflicts have been addressed and is a confirmation step to ensure the requirements are correct and agreed upon.
- Verification of Requirements: Verification is the process of ensuring that the system meets the specified requirements. This is often done through various testing methods, such as unit testing, integration testing, and system testing. Verification helps identify any deviations from the requirements early in the development cycle.
- Continuous Monitoring and Updating: Requirements management is not a one-time activity. It requires monitoring and updating throughout the software development lifecycle. This adaptability allows the team to respond to changes in the business environment, stakeholder needs, or technological advancements.
- Documentation and Traceability: Documenting the requirements and maintaining traceability is vital. It helps keep track of the requirements throughout the development process, making it easier to manage changes and ensure all requirements are addressed.

Test Design and Selection (2). The second dimension of the methodology covers the creation and selection of test cases. Test design results in creating or selecting test cases to execute the system and obtain a verdict [84]. Test design and selection have multiple purposes, e.g., covering certain branches and attempting to find failures. Test design in SmartDelta can also be used to measure the system's quality. Here is a detailed view:

- Test Case Creation: The process begins with the development of test cases. These are specific conditions under which the software is executed to check whether it operates as intended.
- Test Selection Criteria: It is crucial to select the right test cases. The methodology involves setting criteria based on the software's requirements, risk assessments, and the criticality of different features.
- Coverage Goals: Test design in SmartDelta aims to improve delta coverage, including code coverage (like branch coverage), functional coverage, and user scenario coverage. This ensures that all software parts are tested and different functionalities are thoroughly evaluated.
- Failure Detection: An essential purpose of test design is to detect potential software failures. This involves creating test cases that specifically target known vulnerabilities or complex areas of the system to uncover any issues that could lead to software failure.

[84] Eldh, S. (2011). *On test design* (Doctoral dissertation, Mälardalen University).

© 2024 SmartDelta Consortium

- Quality Measurement: Test design in SmartDelta is not only about finding faults but also about measuring various quality aspects of the system, such as performance, usability, security, and compatibility.
- Test Case Optimization: SmartDelta includes strategies for optimizing test cases to enhance efficiency. This might involve removing redundant tests and prioritizing tests.
- Continuous Improvement: Test design and selection in SmartDelta are iterative processes. As the software evolves, so do the test cases. Review and improvement of test strategies are necessary to adapt to changes in software requirements and technological advancements.

Test Amplification (3). Test amplification [85] is a method that automatically develops new test cases by adjusting existing written test cases. A typically used test goal is to enhance test coverage according to a coverage criterion, but this method can be used in different ways to aid both testers and developers. Here is a more comprehensive view:

- Improve Test Coverage: One of the main applications of test amplification is to increase test coverage. This includes code coverage (line, branch, and path coverage) and feature coverage.
- Automated Test Case Generation: Test amplification tools analyze existing test cases and automatically generate new ones by altering inputs, outputs, or internal parameters.
- Support for Testers and Developers: Beyond improving test coverage, test amplification aids testers and developers by reducing the manual workload of creating and maintaining test cases. It also provides insights into potential vulnerabilities and areas of improvement in the software.
- Dynamic Test Case Evolution: Unlike static test case creation, test amplification is dynamic. It continuously evolves test cases in response to changes in the codebase, ensuring that the tests remain relevant throughout the software development lifecycle.

Monitoring and Anomaly Detection (4). Monitoring and anomaly detection are activities in quality assurance that are used to identify abnormal behavior. In general, monitoring is observing or inspecting a system at different points to produce reports, notify an abnormal operation, draw operation baselines, provide inputs to the system or application management, etc. In addition, monitoring attempts to monitor a network or systems to detect abnormal activity or policy violations. Here is a detailed exploration:

- Tools and Techniques for Monitoring: This involves using various tools and techniques, from basic log file analysis to real-time monitoring solutions. These tools collect data on various aspects of the system, such as resource usage, response times, and transaction volumes, providing a view of the system's health over time.
- Anomaly detection refers to the process of identifying patterns in the data that deviate from the established normal behavior. These anomalies could indicate problems such as system failures, security breaches, or operational inefficiencies.
- Creating Operational Baselines: Establishing operational baselines as standard metrics for normal system behavior is an important monitoring aspect. These baselines are crucial for identifying anomalies, as they provide a point of comparison to determine what constitutes abnormal behavior.
- Proactive Alerting and Reporting: Monitoring systems often include mechanisms for alerting to potential issues. This can be real-time alerts, automated reports, or dashboards that provide an at-a-glance view of the system's status.

[85] Danglot B, Vera-Perez O, Yu Z, Zaidman A, Monperrus M, Baudry B (2019a) A snowballing literature study on test amplification. J Syst Softw 157:110398

© 2024 SmartDelta Consortium

- Network and System Monitoring: While monitoring can be applied to individual applications, it is also crucial at the network and system levels. This includes tracking network traffic to detect unusual patterns or volumes that could indicate security incidents like data breaches or DDoS attacks.
- Adaptive and Predictive Monitoring: Monitoring systems incorporate adaptive and predictive capabilities to anticipate potential issues based on historical data and trends.

Static Analysis (5). The static analysis evaluates the software without running it or evaluating specific inputs. Instead of trying to prove that the system fulfills its specification (e.g., formal verification), such techniques look for violations of valid or suggested programming and code review practices. Here is a more comprehensive view:

- Identification of Code Violations: A primary function of static analysis is to identify violations of coding standards and best practices. These could include potential bugs, security flaws, performance issues, and maintainability concerns.
- Code Quality and Maintainability: Static analysis tools assess the quality and maintainability of the code by checking for complex code structures, duplicated code, and adherence to coding conventions.
- Security Vulnerability Detection: Static analysis is particularly effective in identifying security vulnerabilities, such as SQL injection, cross-site scripting (XSS), and buffer overflows.
- Integration with Development Processes: Static analysis tools are often integrated into the development process, either as part of development environments or as part of continuous integration/deployment pipelines.
- Compliance with Standards and Regulations: Static analysis helps ensure compliance with industry standards and regulatory requirements (e.g., MISRA C).
- Formal Verification vs. Static Analysis: Unlike formal verification, which tries to prove that a system meets its specifications, static analysis is more about finding deviations from good programming practices.

Table 8. Mapping between the different dimensions of the SmartDelta Methodology for Quality Assurance and the individual tool components developed in WP3[86].

Methods/ Tool Name	Requirements Management and Analysis (1)	Test Design and Selection (2)	Test Amplification (3)	Monitoring and Anomaly Detection (4)	Static Analysis (5)
VARA+	x				
NALABS	x				x
SEAFOX		x	x		
FOKUS-CBTS		x	x		
IFAK-TCG		x			
Test Effort Estimator				x	x
SoHist				x	x
UI Test Generator		x			
Mut4SLX		x	±		
SONATA	x	x			
DRACONIS				x	x
PyLC		x			

[86] x stands for fully supported/available in this tool and ± for partially supported or experimental feature.

Methods/ Tool Name	Requirements Management and Analysis (1)	Test Design and Selection (2)	Test Amplification (3)	Monitoring and Anomaly Detection (4)	Static Analysis (5)
GW2UPPAAL		x			
EARS	x	x			
JAZURE	x				
Smellyzer				x	x
Relink	x				
TAA				x	

5. Broader Context of the SmartDelta Methodology

The SmartDelta project aims to optimize and automate quality assurance in incremental industrial software systems. WP3, in particular, focuses on creating and integrating automated tools and methodologies to assess the quality of "deltas," which are defined as incremental software changes across different product versions. This approach is crucial for ensuring that modifications in software systems, whether they pertain to functionality, performance, or other non-functional requirements, do not degrade overall system quality.

The tools and methodologies developed in WP3, as outlined in D3.4 Delta-oriented Quality Assurance Methodology, are designed to fit into the larger framework of SmartDelta by addressing key challenges of modern industrial software development, CI, CD and the evolution of software systems. For example, the methodology provides mechanisms for test case generation, regression testing, and static and dynamic analysis tailored to specific deltas. This ensures that the testing process is efficient, targeted, and capable of handling evolving requirements, which aligns with SmartDelta's broader goals of reducing manual effort, increasing system reliability, and enhancing the reuse of existing software artifacts.

For example, utilizing tools like SEAFOX for combinatorial test generation, DRACONIS for delta computation between models, and NALABS for requirements static checking, WP3 contributes to the overarching goal of automating quality assurance in ways that allow for faster, more reliable development cycles. Moreover, these tools are built to operate within continuous engineering environments, supporting ongoing development and adaptation. These are essential to SmartDelta’s vision of reducing time-to-market and maintaining high software quality standards across product versions. Finally, the tools from WP3 play a key role in delivering a comprehensive delta-aware quality assurance process that supports the long-term sustainability and scalability of industrial software systems, aligning with the core objectives of the SmartDelta project.

5.1. Selected Tools and Their Role in WP3

One of the tools developed in WP3 is DRACONIS, which computes deltas between different model versions and proposes which checks need to be re-run based on those changes. DRACONIS enables more efficient regression testing by ensuring that only relevant tests are executed, thus reducing redundant testing activities. This aligns with the project’s broader goal of optimizing testing cycles in CI/CD environments, where quick feedback is essential. DRACONIS plays a critical role in keeping regression tests focused and relevant as software systems evolve over time.

© 2024 SmartDelta Consortium

Another tool in WP3 is PyLC, which focuses on the incremental translation and logging of changes in programmable logic controllers (PLCs). As industrial systems often rely on PLCs for critical functionality, it is important to ensure that test cases are updated as changes occur in these systems. PyLC supports this need by tracking changes in PLC programs facilitating the integration of incremental testing and regression strategies. This tool not only supports the broader SmartDelta goal of improving automation but also ensures that test cases remain aligned with evolving system requirements, minimizing errors and ensuring software reliability.

SEAFOX is a combinatorial test generation tool developed to augment manual tests with automatically generated ones, specifically for IEC 61131-3 compliant PLC software. SEAFOX reduces the time needed for test case generation, which is especially useful in continuous integration environments where testing cycles are short. This tool helps by automatically generating test cases that achieve high code coverage, ensuring that all potential system behaviors are adequately tested. SEAFOX's role in WP3 is essential to increasing the efficiency of the testing process, reducing manual effort, and contributing to the SmartDelta project's objective of enhancing overall software quality through automated testing.

The GW2UPPAAL tool automates test suite generation when modifications in functional or non-functional requirements lead to product deltas. As software systems evolve, requirements often change, creating new deltas that must be tested to ensure continued system integrity. GW2UPPAAL supports delta-based testing by automating the generation of test suites that are aligned with these evolving requirements. This ensures that testing remains up-to-date with the latest system changes, providing a more agile response to development needs and contributing to the continuous assurance of software quality throughout the product lifecycle.

Another tool in WP3 is the Delta Test Coverage Evaluation (dTCE) tool, which evaluates test coverage across different software versions or deltas. The tool ensures that test suites maintain comprehensive coverage, even as the software evolves, by comparing multiple versions and assessing where tests may need to be added or adjusted. dTCE helps to ensure that critical aspects of the software are not overlooked during testing, supporting the larger SmartDelta objective of maintaining high-quality software through rigorous and efficient testing processes. This is particularly important in industrial settings where software reliability and safety are paramount.

Smellyzer. is designed to elevate software development quality by identifying and addressing process smells within code review and bug tracking. By pinpointing detrimental patterns, such as large changesets, sleeping reviews, and unassigned bugs, Smellyzer provides teams with actionable insights to refine their practices. This results in improved traceability, faster issue resolution, and a streamlined development workflow, ensuring a more cohesive and efficient software development lifecycle.

The Jazure is a tool developed within WP3 that automates the linkage of work item IDs between Azure DevOps and Jira, enhancing traceability and efficiency in the software development process. By synchronizing work items across these platforms, Jazure ensures that incremental changes—or deltas—are accurately tracked and associated with their corresponding pull requests. This automation is crucial for managing evolving software systems, as it reduces the manual effort required to maintain consistent and up-to-date work item associations. Jazure also enforces branch policies that mandate work item linkage before code can be merged into the master branch, preventing untracked changes from entering the production codebase. By facilitating better management of deltas and ensuring that all modifications are properly documented and reviewed, Jazure directly contributes to the SmartDelta project's goals of optimizing continuous integration and delivery pipelines, enhancing system reliability, and maintaining high software quality standards across different product versions.

© 2024 SmartDelta Consortium

Each of these tools contributes directly to the broader objectives of the SmartDelta project by automating critical aspects of software testing, reducing manual effort, and ensuring that testing remains relevant and effective as software systems evolve. Through tools like DRACONIS, PyLC, SEAFOX, GW2UPPAAL, Jazure, Smellyzer, and dTCE, WP3 supports the project's overall goal of improving quality assurance processes in continuous engineering environments. Together, these tools enable the SmartDelta methodology to optimize CI/CD workflows, enhance testing efficiency, and ensure that high-quality software is delivered across multiple product versions and deltas.

5.2. Example: How WP3 Tools Fit Into a Real Use Case

In a real-world use case, the tools developed within WP3 of SmartDelta can be applied to a complex industrial system, such as a train control management system. This system involves multiple software components that are regularly updated, creating new deltas that require rigorous testing to ensure continued functionality and safety.

In this context, DRACONIS would be used to compute deltas between different versions of the system models. By identifying the precise changes in each new version, DRACONIS helps to determine which tests need to be re-run, optimizing the regression testing process. This ensures that only the relevant tests are executed, significantly reducing testing time and resource usage while maintaining high testing accuracy. PyLC would track incremental changes in the system's programmable logic controllers (PLCs). As these controllers are often updated with new configurations or functionality, PyLC logs these changes incrementally, allowing for targeted regression testing of the PLC software. This minimizes disruptions in the development cycle and ensures that critical PLC functions are thoroughly tested after each update. SEAFOX plays a role by automatically generating test cases for the IEC 61131-3 compliant PLC software used in the system. SEAFOX reduces the manual effort required to create test cases in an environment where multiple system configurations must be tested. Its combinatorial test generation ensures high code coverage, improving the reliability of the testing process and detecting potential issues early in the development cycle. As new requirements emerge or existing ones are modified, GW2UPPAAL will automatically generate new test suites to reflect these changes. This is particularly important in a system like train control management, where safety and functionality are paramount, and any changes in requirements need to be immediately reflected in the testing process. GW2UPPAAL ensures that the system is always tested against the most current set of requirements, maintaining high-quality standards. Jazure automates linking work items between Azure DevOps and Jira during development. It synchronizes tasks across both platforms, properly tracking every code change. This enhances traceability and team coordination, preventing miscommunication. In complex systems like the train control management system, it ensures all changes are accounted for in both development and project management tools. In this case, the combined functionality of specific WP3 tools ensures that the train control management system undergoes efficient, targeted, and comprehensive testing throughout its lifecycle.

6. Selected Performance Metrics – An Overview

The performance metrics data provides valuable insights into the efficiency and effectiveness of the project's software tools. Examining these metrics allows us to evaluate current performance, set improvement targets, and track progress over time. Below, we provide detailed metrics and analysis for several WP3 tools.

NALABS is designed to identify and analyze requirement issues. The tool initially identified ten issues per 300 requirements, translating to a 3.3% problematic requirement detection rate. Although

© 2024 SmartDelta Consortium

the current values are yet to be measured, the target is to find ten issues per 100 requirements, aiming to significantly enhance the quality assurance process. The data for NALABS was collected from multiple resources, including companies developing safety and security-critical systems and various open-source repositories such as GitHub, Zendo, and the NLRP benchmark. The total analyzed requirements amounted to 7423 from different projects, providing a robust dataset for evaluation.

SEAFOX focuses on combinatorial test generation, initially taking 15 minutes on average to generate test cases per System Under Test (SUT) with an 81% branch coverage. The current metrics are still to be measured, but the target aims to reduce the generation time to 10 minutes per SUT while increasing branch coverage to over 90%. This reflects a strong focus on improving testing efficiency and comprehensiveness, which is crucial for faster and more effective testing processes.

Test Effort Estimator measures the accuracy of performance test analyses. While the initial accuracy values were not specified, the tool has achieved an approximate 95% accuracy for commits and releases in the largest repository and around 73% for pull requests. The target is set at 97%, indicating a high standard for performance testing accuracy. This improvement showcases the tool's progress and its commitment to maintaining high reliability in test analysis.

CBTS addresses the selection of regression tests from a huge set of regression tests based on the ability to detect unwanted side effects of each individual regression test. The selection algorithm considers the changes that have been applied to the source code as the connected source code areas, which the regression test covers or that influence the regression test. CBTS is evaluated by the following metrics: regression test selection time, number of selected regression tests, reduction in execution time of regression tests, and defect capability of selected regression tests. The Vaadin Flow Framework that was used as a pilot project for the CBTS tool has the following characteristics: 18 core modules, 34 test modules, a test suite comprised of over 6680 tests, and a total test execution time of 1,5h. For the validation of CBTS, it was only applied to a subset of the modules, namely: flow-dev, flow-polymer2lit, flow-pwa-disabled-offline and flowmisc (as shown in Table 9).

Table 9. The Evaluation of CBTS with Respect Test Components.

Test componentent	amount of tests	type of side effects	execution time	number of executions
test-pwa-disabled-offline	3	Changes in import Dependencies from external libaries Changes to Interfaces POM Dependencies	1.4s	9
flow-polymer2lit	23	Changes in import Dependencies from external libaries Whitespace and minor changes POM Dependencies	3.2s	12
test-misc	10	Changes in import Dependencies from external libaries Java Changes to Class Inheritance POM Dependencies	2.9s	12
test-dev-mode	27	Changes in import Dependencies from external libaries POM Dependencies	3.1s	11

© 2024 SmartDelta Consortium

CBTS achieved the following performance metrics:

- Regression test selection time: 30.23s (there is no comparison for this number because regression test selection has not been done manually before)
- The number of selected regression tests:
  - pwa-disabled-offline: 3/35 (~92% reduction)
  - Flow-polymer2lit: 23/75 (~70% reduction)
  - Flowmisc: 10/152 (~94% reduction)
  - Flow-dev: 27/265 (~90% reduction)
- Reduction in execution time:
  - pwa-disabled-offline: ~95% reduction
  - Flow-polymer2lit: ~80% reduction
  - Flowmisc: ~94% reduction
  - Flow-dev: ~96% reduction
- Defect detection capability:
  - None of the selected regression tests found a side-effect-caused defect.
  - The entire test suite detected only a few side-effect-caused defects; most defects found by the entire regression test suite were caused by direct changes applied to the code, not because of side-effects.
  - We cannot make a statement about CBTS's defect detection capability for now; further experiments are needed to evaluate the accuracy of the side-effects classification scheme and fine-tune the selection algorithm.

SoHist enables analysis of software quality metrics from SonarQube Analysis and offers custom views to track changes over time. It introduces the new *Weighted Code Evolution Significance metric*, a novel method for prioritizing SonarQube's main metrics (Reliability, Security, Maintainability, Test Coverage, and Duplicated Lines) based on user-defined weights. This allows users to focus on specific metrics, visually representing the project's lifecycle and highlighting significant changes according to the assigned weights. The greater the *Weighted Code Evolution Significance*, the more impactful the change in the chosen category. For details, please refer to https://zenodo.org/records/7713698.

PyLC evaluates mutation scores of PLC programs translated into Python. Although the initial values were not provided, the current average mutation score is 58.78%, with a target of 70%. This improvement indicates significant progress in test generation quality, demonstrating PyLC's ability to ensure higher software robustness through effective mutation testing.

GW2UPPAAL focuses on the average time it takes to generate models. Initially, it took 562.3 seconds, but this has now been drastically reduced to 0.1833 seconds, nearing the targeted 0.2 seconds. This showcases a substantial improvement in operational efficiency, which is crucial for timely and effective model generation.

ReLink determines missing links between PRs and issues based on a confidence score normalized between 0 and 100, calculated using text similarity and heuristic rules. The tool's effectiveness was evaluated in an open-source project and an industry-based case study. In the open-source project, ReLink achieved a top-5 accuracy of 0.80 and a precision of 0.77. In the industrial case study, ReLink's effectiveness was validated by practitioners selecting the correct link from five issue suggestions, resulting in a top-5 accuracy of 0.86 and a Mean Reciprocal Rank of 0.84.

Smellyzer was found to be 100% useful to detect CR smells and 93% useful to detect BT smells across three case studies. The tool's practicality is determined using the System Usability Scale (SUS), which is calculated as 77.5 for CR smell detection and 80.0 for BT smell detection.

© 2024 SmartDelta Consortium

Jazure focuses on automating the linkage of work items between Azure DevOps and Jira to enhance traceability in the development process. Initially, the success rate of linking work items to pull requests was approximately 70%, hindered by manual errors and oversight. Currently, Jazure has achieved a 100% success rate in establishing these links, with the target being to maintain this level consistently. This improvement underscores Jazure's effectiveness in eliminating manual errors, ensuring all code changes are properly tracked and associated with their respective work items, thus enhancing efficiency and reliability in software development.

7. Summary and Conclusions

The SmartDelta project focuses on advancing automated quality assurance and optimization for incremental software system development in industrial contexts. Within the scope of WP3, its main aim is to develop a methodology that integrates functional and extra-functional testing, regression testing, static and dynamic analysis, and anomaly detection. These efforts address iterative and incremental software development challenges, particularly in agile and CI/CD environments.

The deliverable introduces a delta-oriented quality assurance methodology that improves the efficiency and effectiveness of testing processes. This methodology is grounded in tools and techniques specifically designed to cater to incremental changes or deltas in software development. It employs automated test generation, regression testing strategies, and delta-specific analysis to ensure high levels of reliability and coverage. These tools, along with others, address specific aspects of delta-aware quality assurance, providing tailored solutions for challenges such as test selection, amplification, and regression test optimization. Performance metrics and case studies illustrate the real-world impact of these tools. The work establishes a path for future research and development in this area. Furthermore, the methodology can serve as an educational resource.

© 2024 SmartDelta Consortium