Leveraging Semi-formal Approaches for DepDevOps

Wanja Zaeske[1] and Umut Durak[1,2](✉)

[1] Department of Informatics, Clausthal University of Technology, Julius-Albert-Str. 4, 38678 Clausthal Zellerfeld, Germany
{wanja.zaeske,umut.durak}@tu-clausthal.de

[2] Institute of Flight Systems, German Aerospace Center (DLR), Lilienthalplatz 7, 38108 Braunschweig, Germany
umut.durak@dlr.de

Abstract. While formal methods have long been praised by the dependable Cyber-Physical System community, continuous software engineering practices are now employing or promoting semi-formal approaches for achieving lean and agile processes. This paper is a discussion about using Behaviour Driven Development, particularly Gherkin and RSpec for DepDevOps, DevOps for dependable Cyber-Physical Systems.

Keywords: Semi-formal approaches · Dependable systems · Agile

1 Introduction

Software engineering is evolving towards removing disconnects among its activities with employing continuous practices to achieve agile processes. First, Test-Driven Development (TDD) bridged the gap between implementation and testing. Then, Continuous Integration (CI) and Continuous Deployment (CD) attacked the disconnect between the development and deployment. Eventually Behaviour-Driven Development (BDD) enhanced TDD with specification and continuous acceptance testing. Now DevOps is connecting development and operations.

Continuity with streamlined and automated processes has long been studied in software engineering to achieve agility. Not only iterative and incremental development life-cycles but also inevitable software evolution during operation have been asking for rapid feedback cycles between the developer and the user. DevOps is defined as the set of practices for reducing the time between committing the code and using it in normal operation [3]. It connects two worlds: the development and the operation. Accordingly it consists of two integrated cycles; one for development and the other for operation. A *Release* starts an operation cycle that is composed of *Deploy*, *Operate* and *Monitor* steps. Feedback from monitoring starts the next development cycle that is composed of *Plan*, *Design*, *Build*, *Test* and *Release* steps.

DevOps harmonizes the agile software engineering practices, from TDD and BDD to CI and CD to realize a fast forward track. It further promotes monitoring and logging mechanism for feedback loops. As stated by Ebert et al. [8], "obviously, the achievable cycle time depends on the environmental constraints and deployment model". Inspired from the other application domains, Cyber-Physical Systems (CPS) users are now asking for on-the-fly software updates, easy problem reporting, and frequent feature enhancements. While the expectation is to have mobile-like driver-assistance or avionics applications, the dependability constraints and embedded software deployment models are preventing any cycle faster than years, deployed at the service centers by authorized personal.

This paper concentrates on dependability constraints, knowing that embedded software deployment is also an open research area for achieving full-fledged DevOps for CPS. Dependability is a system property that describes its ability to deliver services that can justifiably be trusted [1]. It is an integrated concept for availability, reliability, safety, integrity, and maintainability. Formal methods are mathematical techniques for specifying and verifying systems [7]. They have long been proposed and employed to tackle dependability challenges in general [17,19] and safety challenges in particular [2,5,12,13]. The dependable CPS community of the last decade has also praised formal methods as one of the key techniques [11,18]. While there are many research efforts that aim at integrating formal methods and agile practices, such as [4,9,21], this paper brings the semi-formal methods that are being practiced in DevOps world to the attention of dependable CPS community. The methods of interest are executable specification methods of BDD, Gherkin [22] and RSpec [6].

2 Behaviour Driven Development in DevOps

Chelimsky et al. [6] define BDD as "implementing an application by describing its behavior from the perspective of its stakeholders" It builds upon TDD, and promotes a semi-formal *ubiquitous language* for the specification of behaviours that is accessible to all the stakeholders of the system. The *ubiquitous language* idea is based on Evans [10], who stresses that the linguistic divide or the language fracture between the domain expert and the technical team leads only to vaguely described and vaguely understood requirements. The aim of BDD is to come up with executable as well as a human readable specification of the system, in a single representation [14].

BDD is structured around *features* which can be defined as the capabilities provided by the system that create a benefit to its users. A feature is usually described in BDD by a title, a brief narrative, and a number of scenarios that serve as acceptance criteria. Scenarios are concrete examples to describe the desired behaviours of the system. When the concrete examples are executable; they turn the criteria to an acceptance test. BDD calls this *automated acceptance testing.*

Gherkin is the common language to write features, particularly for the Cucumber test automation framework [22]. While it is not a Turing Complete language, it has a grammar enforced by a parser. It aims at human readability, while enabling execution in Cucumber using its grammar. The basic Gherkin keywords to specify a scenario are *Given*, *When* and *Then*. *Given* is used to describe the context of the system, the state of the system before an event. *When* is used to specify the event(s) and eventually *Then* is used to give the outcome(s).

Features that are written in Gherkin and executed in Cucumber are regarded as outer cycle. They define the behaviour of a system. RSpec is the name given to the language and the test automation framework that is used to specify the behaviour of objects [6]. It is regarded as the inner cycle. The test code is structured using *Describe*, *Context* and *It* keywords. *Describe* is used to define an example group. An example is a test case. *Context* is similar to *Describe*; it is used to group examples with a certain context. *It* is used to specify an example.

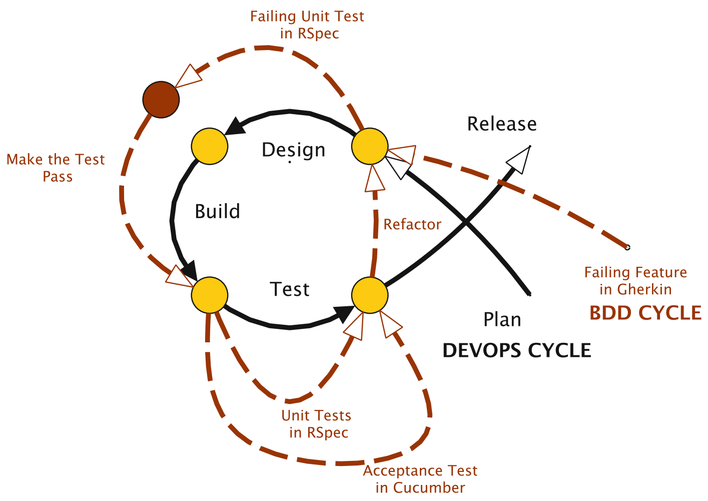

Fig. 1. Behaviour driven development in DevOps (Adapted from [23])

Figure 1 extends Yackel's ideas [23] about the integration of BDD in the development cycle of DevOps. The Plan, Design, Build, Test, and Release steps can be realized using a layered BDD approach with Cucumber and RSpec duo. Features are specified in Planning step using Gherkin. At the Design step, following a high level-design, required unit behaviour is specified using RSpec. Low-level design and implementation followed by Build step end up with a system to be tested. With every passing unit test in RSpec, system also undergoes acceptance tests in Cucumber against the feature specifications in Gherkin. The inner cycle ends when the outer cycle, namely the acceptance tests are successful and eventually leads to the Release step.

3 Gherkin and RSpec for Dependable CPS

If we take avionics as an example dependable CPS domain, the DO-178C Software Considerations in Airborne Systems and Equipment Certification [15] sets the baseline for process requirements. It necessitates high-level requirements specification that interprets the system requirements to the software item, and low-level requirements that can be directly implemented without further information. The authors would like to start a discussion about using Gherkin for the specification of high-level requirements and RSpec for low-level requirements with an example avionics application, Terrain Awareness and Warning System (TAWS).

TAWS is an airborne equipment introduced in 1990's for reducing the risk of the Controlled Flight Into Terrain (CFIT) accidents. It produces aural and visual warning for impending terrain with a forward looking capability and continued operation in landing configuration [16]. There are three classes of TAWS. Class A, being most stringent, are for large turbine powered aircraft with at least one radio altimeter; Class B for smaller turbine powered aircraft which may not have radio altimeter and Class C, being least stringent, for smaller general aviation aircraft.

Class C TAWS features include Forward Looking Terrain Avoidance (FLTA), Premature Descent Alerting (PDA), Excessive Rate of Descent (Mode 1), Negative Climb Rate or Altitude Loss After Take-Off or Go Around (Mode 3) and Five Hundred Foot Callout. The authors are prototyping a Class C TAWS, namely Open TAWS to demonstrate dependable DevOps concepts. Sample Gherkin and RSpec specifications that will be introduced in the following sections can be found at Open TAWS Git repository.[1]

DO-367 Minimum Operational Performance Standards (MOPS) for Terrain Awareness and Warning Systems (TAWS) Airborne Equipment states that during non-precision approaches Class C Equipment shall generate at least Five Hundred Foot Callout within 1.3 s of descending through 500 foot above terrain or the nearest runway, or the altitude Callout within 1.3 s of descending through the pilot selected altitude when the altitude callouts are not inhibited [16].

An excerpt from the Gherkin specification of the Five Hundred Foot Callout high-level requirement can be as follows:

```
Feature: Five Hundred Foot Callout
  DO-367 TAWS_MOPS_292
  Scenario: Aircraft less then 500 feet above the terrain
    Given Aircraft is in non-precision approach
        And Altitude callout is not inhibited
        When Aircraft descends under 500 feet above the terrain
        Then Within 1.3 seconds Five Hundred Foot Callout is given
```

Open TAWS is designed to have a terrain server, which needs to provide the terrain query interface that returns the altitude of a point at a given geographical position. Rust is selected as the programming language for Open TAWS due to its promises in safety, performance and zero-cost abstractions and growing embedded systems community [20]. An excerpt from the RSpec specification of

[1] https://gitlab.tu-clausthal.de/aeronautical-informatics/otaws.

the low-level requirement for the altitude query interface using Rust-RSpec[2] can be as follows:

```
rspec::describe("Altitude query", environment, |ctx| {
    ctx.specify("a position in geographical coordinates", |ctx| {
        ctx.it("should return the altitude of the terrain
        at that position", |env| {
            assert_eq!(env.sut.altitude(env.position),
            env.expected_altitude);
        });
    });
})
```

4 Outlook

The paper is a short discussion starter for employing semi-formal specification approaches of Behaviour Driven Development; namely Gherkin and RSpec for dependable CPS. The automated traceability and requirements-based test coverage analysis using test automation tools supporting Gherkin and RSpec is a promise of such an approach to support dependability and lean development. On the other side, ubiquitous specification languages, and specification-as-code approach enables both continuity and agility.

Both Cucumber and RSpec are written in the Ruby programming language, and were originally used for Ruby. While Cucumber (and Gherkin) could spread to various programming languages, RSpec is still almost exclusive to the Ruby community. There are both Cucumber and RSpec implementations for Rust. While Cucumber-Rust[3] is feature rich, Rust-RSpec is relatively limited and has not been maintained for a long time. Cucumber-Rust parses the human-readable semi-formal feature specifications and provides an API for developing test cases that implements them. However, limited API of Rust-RSpec almost fails to enable writing readable specification; they rather look like basic unit test code.

This paper reports the early experience from the example avionics application. Future work includes demonstration of a full-fledged DepDevOps with an extensive discussion about alternative tools and infrastructures.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. **1**(1), 11–33 (2004)
2. Barroca, L.M., McDermid, J.A.: Formal methods: use and relevance for the development of safety-critical systems. Comput. J. **35**(6), 579–599 (1992)

[2] https://github.com/rust-rspec/rspec.
[3] https://github.com/bbqsrc/cucumber-rust.

3. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Professional, Boston (2015)
4. Black, S., Boca, P.P., Bowen, J.P., Gorman, J., Hinchey, M.: Formal versus agile: survival of the fittest. Computer **42**(9), 37–45 (2009)
5. Bowen, J.P., Stavridou, V.: Formal methods and software safety. In: 1992 Safety of Computer Control Systems (SAFECOMP 1992), pp. 93–98. Elsevier (1992)
6. Chelimsky, D., Astels, D., Helmkamp, B., North, D., Dennis, Z., Hellesoy, A.: The RSpec Book: Behaviour Driven Development with Rspec. Cucumber, and Friends, Pragmatic Bookshelf **3**, 25 (2010)
7. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. (CSUR) **28**(4), 626–643 (1996)
8. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: DevOps. IEEE Softw. **33**(3), 94–100 (2016)
9. Eleftherakis, G., Cowling, A.J.: An agile formal development methodology. In: Proceedings of the 1st South-East European Workshop on Formal Methods, pp. 36–47 (2003)
10. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional, Boston (2004)
11. Fitzgerald, J., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-Physical Systems design: formal foundations, methods and integrated tool chains. In: 2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering, pp. 40–46. IEEE (2015)
12. Gerhart, S., Craigen, D., Ralston, T.: Experience with formal methods in critical systems. IEEE Softw. **11**(1), 21–28 (1994)
13. McDermid, J.A.: Formal methods: use and relevance for the development of safety-critical systems. In: Safety Aspects of Computer Control, pp. 96–153. Elsevier (1993)
14. Okolnychyi, A., Fögen, K.: A study of tools for behavior-driven development. In: Full-Scale Software Engineering/Current Trends in Release Engineering, p. 7 (2016)
15. RTCA: DO-178C software considerations in airborne systems and equipment certification. RTCA (2011)
16. RTCA: DO-367 minimum operational performance standards (MOPS) for terrain awareness and warning systems (TAWS) airborne equipment. RTCA (2017)
17. Rushby, J., Underst, F.B.S., Stankovic, J.A.: Formal methods for dependable real-time systems (1992)
18. Seshia, S.A.: New frontiers in formal methods: learning, cyber-physical systems, education, and beyond. CSI J. Comput. **2**(4), R1 (2015)
19. Thomas, M.: The role of formal methods in achieving dependable software. Reliab. Eng. Syst. Saf. **43**(2), 129–134 (1994)
20. Uzlu, T., Şaykol, E.: On utilizing rust programming language for Internet of Things. In: 2017 9th International Conference on Computational Intelligence and Communication Networks (CICN), pp. 93–96, September 2017. https://doi.org/10.1109/CICN.2017.8319363
21. Wolff, S.: Scrum goes formal: agile methods for safety-critical systems. In: 2012 First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), pp. 23–29. IEEE (2012)
22. Wynne, M., Hellesoy, A., Tooke, S.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Bookshelf (2017)
23. Yackel, R.: BDD in DevOps: an example of BDD in continuous integration. https://www.qasymphony.com/blog/bdd-devops-example-bdd-continuous-integration/. Accessed 20 May 2020