

Behavior Driven Development for Airborne Software Engineering

Wanja Zaeske* and Janick Beck†

Clausthal University of Technology, Institute of Informatics, 38678 Clausthal-Zellerfeld, Germany

Christoph Torrens‡ and Umut Durak §

German Aerospace Center (DLR), Institute of Flight Systems, 38108 Braunschweig, Germany

Behavior Driven Development (BDD) is a test driven development approach which aims at bridging the gap between the end-user and the software development. It promotes a semi-formal ubiquitous language for executable behavior specification and automated acceptance testing. The BDD infrastructure Cucumber and the language Gherkin have been established in many application domains. This paper discusses their utilization for airborne software engineering with an example application, Class C Terrain Awareness and Warning System, being developed using Rust language with Rust BDD infrastructure.

I. Introduction

The tendency in modern software engineering practices is towards removing disconnects among its activities by employing continuous practices to achieve agile processes. After Test-Driven Development (TDD) bridged the gap between implementation and testing, Continuous Integration (CI) and Continuous Deployment (CD) attacked the disconnect between development and deployment. Now DevOps is connecting development and operations. In this context, Behavior Driven Development (BDD) establishes a practice based on the behavior specifications from the end-user perspective. It builds upon TDD and promotes a semi-formal ubiquitous language for the specification of behaviors that is accessible to all the stakeholders of the system. BDD aims to come up with executable as well as a human-readable specification of the system.

Gherkin is the common language to write features, particularly for the Cucumber test automation framework [1]. 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. Gherkin even aims to be language independent, however we argue that sticking to the English language is a good measure to increase maintainability and improve collaboration efforts.

There is a growing interest in avionics domain towards the agile software engineering methods. The article recently published by the US Air Force Life Cycle Management Center introduces their new agile software engineering approach, DevSecOps, at F-16 Operational Flight Program M7.2+, as the sunset of a legacy software development approach “waterfall” [2]. With DevSecOps, they announce a release cadence reduction for new software to flight test from 18 weeks to 12 weeks and to warfighter from 3-4 years to two years. While these figures are still very high for many application domains, the progress is a big leap for airborne software engineering.

With the motivation to push agile practice further, the authors would like to start a discussion about using Gherkin for the specification of DO-178C high-level requirements [3] over an example avionics application. The paper extends Zaeske and Durak [4]. It investigates, reports a demonstrator experience and further elaborates the application of BDD for avionics systems under software considerations for airborne systems.

II. Background

A. Behavior Driven Development

Chelimsky et al. [5] 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

*Student Assistant, Aeronautical Informatics.

†Student Assistant, Aeronautical Informatics.

‡Research Scientist, Unmanned Aircraft, and AIAA Senior Member.

§Group Leader, Safety Critical Systems & Systems Engineering, and AIAA Associate Fellow.

behaviors that is accessible to all the stakeholders of the system. The *ubiquitous language* idea is based on Evans [6], 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 [7].

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 behaviors of the system. When the concrete examples are executable; they turn the criteria to an acceptance test. BDD calls this *automated acceptance testing*.

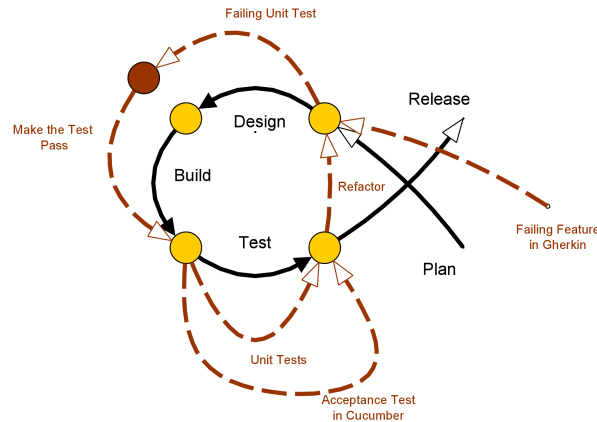


Fig. 1 BDD in Plan-Design-Test-Release Cycle (Adapted from [8])

Figure 1 presents how BDD accompanies the Plan-Design-Test-Release cycle. Features that are written in Gherkin and executed in Cucumber are usually regarded as outer cycle. They define the behavior of a system and are used for acceptance testing. For the inner circle, developers opt for unit tests. Further research needs to be done to choose a strategy for refining the low level specification. For this article however we will focus on the outer level, accordingly the DO-178C high level requirements and therefore Gherkin and Cucumber.

B. Gherkin

Each Gherkin feature is described in its respective feature file, a text file with the `.feature` file extension. It contains an arbitrary number of scenarios, which serve as acceptance criteria for the feature. A scenario typically consists of multiple steps, which describe the actions needed to stimulate the scenario and check the outcome. The steps of a scenario are distinguished into three different classes.

- *Given* prefaces a step which describes some initial state of the application or the world surrounding the application.
- *When* introduces a mutation of state on the application or the world. Often this can also be describe as the occurrence of an event.
- *Then* is used to describe an expected result, after the arrival of an event in a prior *When* step.

It is not necessary to have a step of every class inside every scenario. Therefore, it is possible to have a scenario which for example only contains a *When* and a *Then* step. Steps can be started with a couple of other keywords as well (*And*, *But* & ***), however all of them are reduced to one of the three-step classes introduced earlier. Following the class prefix of a step any prose text is allowed, but a brief and concise sentence is preferable. A scenario can as well contain a table with data, for example for the insertion of test data into a database. Furthermore, it is also possible to define a scenario outline which reads values from a table. These values than are put into the text to their respective placeholder, generating one scenario per row in the table. This is useful for repetitive scenarios where only a couple of words differ.

In order to help to organize the scenarios, it is possible to tag them with an arbitrary number of tags. Most Cucumber implementations support tag based scenario selection which can be used to only execute selected tests. Last but not least, it is possible as well to group scenarios using the `RuLe` keyword. The intended use case is to introduce a business rule for the application, so that all scenarios which prove the compliance with said business rule are grouped together.

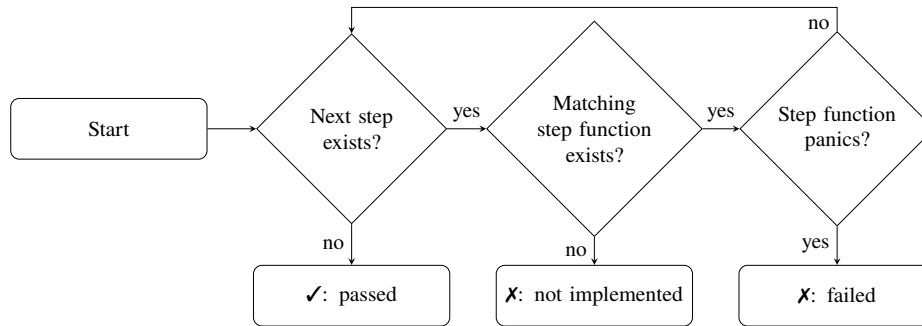


Fig. 2 Cucumber scenario test algorithm

C. Cucumber

The Cucumber framework offers an API to register one function per step definition. Each of these functions serves as the implementation of the changes to the world state described in the step sentence. Each step function receives a mutable reference to the `World`, a user defined type which holds all state during the cucumber testing. The execution of one scenario is described in Fig. 2. Each scenario is presented with a fresh (default initialized) world to avoid any side effects between different scenarios. For the testing the cucumber framework is given a path to a folder containing feature files. Each feature is parsed and for each scenario the algorithm described in Fig. 2 is applied. Once all scenarios where tested, a report is emitted. Figure 4 shows an example report from our demonstrator.

```

Using example: <rate of descent> = 1560, <height> = 100
features/mode_1.feature:22:5
✓ Given Mode 1 is armed                    features/mode_1.feature:23:7
✓ And Mode 1 is not inhibited              features/mode_1.feature:24:7
✓ And steep approach is not selected      features/mode_1.feature:25:7
When the rate of descent is at least 1560 feet per minute
✓ When the rate of descent is at least 1560 feet per minute
features/mode_1.feature:26:7
And the height above terrain is between 100 and 100 feet
✓ And the height above terrain is between 100 and 100 feet
features/mode_1.feature:27:7
Then a Mode 1 caution alert is emitted within 2 seconds
✗ Then a Mode 1 caution alert is emitted within 2 seconds
features/mode_1.feature:28:7
----- [!] Step failed: -----test/cucumber.rs:191:21
assertion failed: world.taws.push(&frame).alerts_count(alert) > 0

Scenario: Must Alert
Using example: <rate of descent> = 2200, <height> = 630
features/mode_1.feature:22:5
✓ Given Mode 1 is armed                    features/mode_1.feature:23:7
✓ And Mode 1 is not inhibited

```

Fig. 3 Partial Cucumber report including a failing scenario

A test succeeds if it does not panic, that is if it's step function does not terminate early with an error. Thus, *Then* steps usually contain `assert!(...)` statements which check for the expected outcome. It is not uncommon to have multiple such statements in one-step function - the Cucumber report will disclose which of them failed, as shown in Fig. 3.

Cucumber allows for regular expression (regex) `*` to be used to match against step sentences. This can be utilized to parse information out of a step definition - which increases the versatility of the Gherkin language significantly. The regex `^(.+)\s+is\s+(.*)inhibited$` matches both `Given Mode 1 is inhibited` and `Given FLTA is not inhibited`, enabling the user to parse steps efficiently. Further on, Cucumber automatically collects all groups and provides each step function with a vector containing them, easing the burden to utilize regex capture groups for data extraction. This combines the possibilities from Natural Language Processing with a customizable Domain Specific Language for test scenario declaration.

Cucumber reports present detailed information about the current state of the implementation. As shown in Fig. 4 we can directly estimate how far the implementation is completed.

*<https://docs.rs/regex/1.4.2/regex/>

```

- When the self-test is initiated                                     features/self_test.feature:16:3
⚡ Not yet implemented (skipped)                                   features/self_test.feature:17:5
Scenario: Initiated Self-Test
Using example: <capability> = fault reporting                       features/self_test.feature:16:3
- When the self-test is initiated                                     features/self_test.feature:17:5
⚡ Not yet implemented (skipped)                                   features/self_test.feature:29:3
Scenario: Initiated Self-Test                                     features/self_test.feature:29:3
- Given the self-test is initiated                                   features/self_test.feature:30:5
⚡ Not yet implemented (skipped)                                   features/self_test.feature:30:5
Scenario: Input Data Smoothing                                     features/self_test.feature:36:3
When the rate of input data reduces or stagnates
- When the rate of input data reduces or stagnates                 features/self_test.feature:37:5
⚡ Not yet implemented (skipped)                                   features/self_test.feature:37:5

[Summary]
4 features
64 scenarios (13 failed, 25 skipped, 26 passed)
400 steps (13 failed, 25 skipped, 162 passed)

Finished in 0.958 seconds.

```

Fig. 4 Partial Cucumber report with unimplemented steps and a summary

III. Behavior Driven Airborne Software Development

The DO-178C Software Considerations in Airborne Systems and Equipment Certification [3] sets the baseline for process requirements for airborne software engineering. It defines the software requirement as a “description of what is to be produced by the software given the inputs and constraints”. The standard 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 high-level requirements are directly produced from the system requirements through analysis. They include functional, performance, interface and safety-related requirements.

DO-178C endorses that the high-level requirements satisfy the following objectives:

- Comply with system requirements
- Accurate and consistent
- Compatible with the target computer
- Verifiable
- Conform to software requirements standards
- Traceable to system requirements
- Describes the behavior and accuracy of algorithms

The Executable Object Code is then needed to be tested for its compliance and robustness with the high-level requirements. The standard endorses a requirements-based test selection. Normal Range Test Cases are required to test the response to the normal inputs and conditions. Robustness Test Cases on the other hand are required for testing the response to abnormal input and conditions.

DO-178C describes two types of test coverage, one is structural and the other is requirements-based. Requirements-Based Test Coverage Analysis aims at confirming that there exist Normal Range Test Cases and Robustness Test Cases for every requirement.

Analyzing BDD against the above mentioned considerations of DO-178C, this paper investigates the ways to accurately specify high-level requirements using Gherkin that can then be used as test cases and executed in Cucumber. Executable high-level requirements inherently provide evidences for the Requirements-Based Coverage without further effort for Trace Data, which is meant to show the bi-directional relation between the requirements and test cases.

IV. A Demonstrator: Open Terrain Awareness and Warning System

TAWS is an airborne equipment introduced in the 1990s 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 [9]. DO-367 Minimum Operational Performance Standards (MOPS) for Terrain Awareness and Warning Systems (TAWS) Airborne Equipment specifies 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 openTAWS to demonstrate BDD concepts. Sample Gherkin specifications that will be introduced in the following sections can be found at openTAWS Git repository[†].

A. Specifying DO-367 MOPS using Gherkin

This section explains the application of the Gherkin language in depth and how it is used in relation to DO-367. The core of DO-367 is Minimum Operational Performance Standards (MOPS). A MOPS consists of one or multiple sentences containing the verb **shall**, following a description of the intended system behavior [9], Chapter 1.1.1. Therefore, a MOPS represents one or more high level requirement. MOPS are similar in semantics to scenarios in Gherkin. While most MOPS can be implemented by one scenario there are exceptions, which will be discussed later on.

To describe more complex circumstances, a scenario outline often seems a good fit - for example we opted to use it for the various envelopes described (e.g. in Mode 1). The corner points of the envelope are used as test cases to describe the area and test if an alert is emitted. For example the Listing 1 could be used with different height values using a Gherkins step like "When the height above terrain is at least <minimum_height> feet and at most <maximum_height> feet" to describe a simple envelope. The values in between the angled brackets are derived from a table following the scenario outline, which contains values for each placeholder per row. This method is easy to apply to data coming from tables or pictures.

As already hinted, there are MOPS which require more than one scenario/scenario outline to be implemented. The most frequent reason for this is the use of a logical "or" in a MOPS. Gherkin does not support an *Or* syntax, instead each combination of circumstances per "or" are phrased into their own scenario (or scenario outline). A good example for is MOPS_269, where multiple possible conditions are given (linked by "or") for an alert not to be emitted [9].

The *Rule* keyword is an addition to the Gherkin introduced in version 6 that is used to group together multiple scenarios within a feature file. While originally intended to be used to group business rules, we use it to group scenarios which come from the same chapter of the source material. This helps to maintain structure in the feature file. However, it is not implemented in all Cucumber versions yet. As the Cucumber Rust implementation for this is not yet done, this keyword was temporarily removed from openTAWS. Each scenario of openTAWS is tagged with the corresponding MOPS index number ("@MOPS_269"). This is done for easier referencing and editing of the feature file. Depending on the Cucumber implementation, an interactive HTML report can be generated which allows to search by tags. Also, it may be possible to run only scenarios which match a certain set of tags - useful if a specific features is worked on where the full test suite takes a long time to run.

The application of the *Given*, *When* and *Then* statement will be shown on a simplified MOPS:

The Equipment shall provide an alert when Mode is armed and Mode is not inhibited and Steep Approach is not selected and the height above terrain is at least 200 feet and at most 2000 feet, for a period of 1.0 seconds or more.

To split this statement up first the static conditions are extracted: "Mode is armed, Mode is not inhibited, Steep Approach is not selected". These conditions are immutable during the scenario, hence they are phrased as *Given* steps. Next conditions that change and trigger results are selected: "The height above terrain is at least 200 feet and at most 2000 feet". This parameter is subject to frequent change during the flight and therefore is selected for the *When* statement. Another hint identifying this parameter as suitable for *When* is that usually a change of this value precedes an alert event. At last the results are selected: An alert is provided within 1.0 seconds. This is an expected outcome and therefore belongs in a *Then* step. The resulting Gherkin scenario is shown in Listing 1.

B. Implementing Test Cases for DO-367 MOPS in Cucumber

While we established the methodology to translate the MOPS from natural language into Gherkin in the prior section, this section will be all about implementing the code to match (against) the scenarios. This will present us with some challenges due to the nature of the API of openTAWS; it is fed with the aircraft state like attitude, altitude, speed etc. and then returns the alerts (if any). While in an ideal world each step of a scenario would translate to one or multiple method calls on some object, multiple steps will only result in one method call in the openTAWS. That requires a particular implementation strategy.

[†]<https://github.com/aeronautical-informatics/openTAWS>

```

1 Given Mode is armed
2 Given Mode is not inhibited
3 Given Steep Approach is not selected
4 When the height above terrain is at least 200 feet and at most 2000 feet
5 Then an alert is emitted within 1.0 seconds

```

Listing 1 Partial Gherkin scenario

```

1 builder.given_regex("^(.+) is (.*)inhibited$", |mut world, mut matches, _step| {
2     matches[1].retain(|c| !c.is_whitespace());
3     let functionality = matches[1].parse().unwrap();
4     if matches[2].starts_with("not") {
5         world.taws.function_uninhibit(functionality);
6     } else {
7         world.taws.function_inhibit(functionality);
8     }
9     world
10 });

```

Listing 2 Rust implementation of a step function

1. Implementation of a Step Function

Consider the step **Given Mode 1 is inhibited** - it is possible to match against exactly this sentence, but by exploiting regular expressions, we can make a step function which matches similar sentences as well. What can these other, similar sentences be? An obvious abstraction is to capture the name of the alert which shall be inhibited. By doing so the sentence **Given FLTA is inhibited** matches as well. Furthermore, it is possible to match the contrary of the sentences, **Given Mode 1 is not inhibited**. Yet another possibility would be to capture the attribute (**inhibited**).

In our implementation, we opted against the last abstraction in order to keep our step functions organized. It is good to abstract up to a certain level, but matching almost any sentence with *is* in it bears the danger of reduced maintainability. That is if a step function matches too generic sentences, it may result in the need to circumvent certain sentence structures in the feature files. It should be the other way round, the feature file should dictate the implementation. Other than that it disturbs the readability of the implementation if the purpose of a step function is opaque.

Taking the above into consideration leads to the implementation which can be seen in Listing 2. Notice how registering a step function for a sentence is done by calling a method with the string (or regex) to match against and a closure[‡]. The closure's arguments are a mutable world, which can be any user defined type (usually a struct), a vector of strings and a reference to a step. The world contains all changes done by step functions (so far) and is to be modified according to the step sentence. The vector of strings contains the matches from the capture groups in the regex. There are functions for matching step sentences without regex as well, which omit the vector of strings in the closure's signature. The last argument of the closure is the step, a struct which contains additional information about the step. Among this information is the table data from the scenario (if any) and the line in the feature file where this sentence appeared.

2. Strategies for Test Generation

The state of an aircraft is normally quite complex, as it includes various information about the attitude, position, altitude, heading etc. All these measurements are recorded at a certain point in time. Therefore, the natural way of representing an aircraft state is a struct with fields for the measurements and a timestamp. This results in a lean API design as well. In fact for a TAWS library to be integrated it is enough to regularly pass a new aircraft state struct to it and retrieve emitted alerts. The signature of a function which does both could be `fn(state: &AircraftState) -> AlertState;`. This offers a well organized and clean API, however it presents us with a challenge: No matter how many steps of a scenario describe the state of the aircraft, only a single function call will yield the result. While this

[‡]an inline function which can also capture variables from the scope of its definition

might be an edge case in API design which can be circumvented in most cases, it still demonstrates a vulnerability in BDD and therefore will be discussed in greater detail. Furthermore, many scenarios do not give one concrete value for an attribute of the aircraft state, but for example a range (as can be seen in Listing 1). In consequence a strategy for the collection (and storage) of properties described in a scenario must be decided on. We came up with three strategies, which will be described in the following sections.

3. *Single State Strategy*

The most straightforward solution is to store a single aircraft state in the Cucumber world. This then is adjusted in the scenario steps until it is used as input for a single call to the openTAWS API. This strategy is compellingly simple, but fails to demonstrate a reasonable coverage: If a range is given for one measurement (as can be seen in 1 for the height), this method can only test one value. To claim a sufficient coverage at least the smallest and greatest value in a closed interval as well as some value in between should be tested.

4. *Generator Strategy*

To address the shortcomings of the aforementioned strategy in regard to coverage it might seem reasonable to implement an aircraft state generator. This generator is given more and more context by each step function. Once a *Then* step function is executed, it calls the generator which in return generates a set of aircraft state structs according to the scenario. These structs are then tested in the openTAWS while assert statements ensure, that the expected outcome of the scenario is fulfilled. Effectively this results in test procedures similar to what might be implemented without BDD.

While this strategy is certainly sufficient to fulfill the test requirements from Appendix A (Description of Test Cases for FLTA and PDA) of [9], it is not without flaw. For this approach to work, a lot of implicit knowledge about which steps are combined in a scenario is needed when implementing the aircraft state generator. This breaks some advantages of BDD: If a new scenario is constructed only of already existing sentences, it still is possible that the generator is not implemented for the very combination of constraints and scenario context which is created by said scenario. While the implementation of each step function for a scenario was necessary, now the implementation of each (used) step combination becomes necessary. On an abstract level this downgrades the role of features from flexible scenario descriptions to only test documentation.

5. *Press Mould Strategy*

Both of the strategies failed to fulfill the following three requirements:

- 1) Allowing for multiple aircraft states to be tested in a single scenario
- 2) Preserving the flexibility to remix feature files without having to adapt the generator implementation
- 3) Keeping the major part of the implementation in the step functions

To tick of the first point, a set of randomly generated aircraft states is stored in every freshly generated world. To make the process deterministic, a pseudo random number generator shall be used for the generation of these. A TAWS should validate the input signals ([9]), which may cause an issue if values between aircraft states which are close in time change by huge margins (e.g. the height changes by several thousand foot in less than a second). Therefore, the set of aircraft states shall be partitioned in groups. Each group consists of aircraft states which are close in time and the derivative of values between two chronologically following states shall be kept in plausible dimensions.

The set of aircraft states is pressed into form by each step function. That is, each step function which implements a constraint on a value of the aircraft state iterates over the set and modifies values according to the step sentence. To avoid issues with the validity of input signals, measurements must be taken to constrain the derivative of each value over time in plausible limits again. This solves the third item as well: now each step function is capable of preparing the world in accordance to its step sentence without knowing any bigger context.

This strategy however is not without problems either. In order for it to work, a good method for generating plausible test data which covers a significant volume of the input space has to be determined. For example the altitude from the test data has to cover heights from zero up to more than 35,000 feet. While it may be desirable to have a large count of aircraft test states, the size of the test suite must not hinder the testing itself. Our estimate is that a hundred test cases partitioned in groups of five might be enough.

```

1  /// # Example
2  ///
3  /// ```
4  /// use otaws::Envelope;
5  /// let points = vec![(1908, 150), (2050, 300), (10300, 1958), (10301, 1958)];
6  /// let envelope = Envelope::new(points).expect("invalid points given to envelope");
7  /// ```
8  pub fn new(points: Vec<(f64, f64)>) -> Option<Self>;

```

Listing 3 Rust Doc-Test example

V. Discussion

Overall we found the process reasonable straight forward. Deep knowledge of the standard is necessary in order to derive useful and sound Gherkin scenarios. However, this is to be expected - the feature files are to be written by the domain expert. Gherkin contributes a lot to the maintainability, as we found out reading feature files again after some weeks passed by. This might have been assisted by the another effect we observed: processing the standard to Gherkin is a good method to both memorize and understand the various details. The requirement to keep scenarios testable helps keeping the connection to the standard substantial. In some situations the BDD process guided us through the software design as well, as the scenarios provide a lot of clues about the data flow and API surface.

Some aspects did cost us considerable amounts of time nonetheless. Besides the issues which we've discussed already in greater detail, these include the various bugs in the Rust Cucumber implementation that we discovered [§]. Another danger in the process can be witnessed in this very paper: using BDD, one usually starts implementing before knowing the standard in total. This helps to generate results fast, but it might lead to design bugs which are expensive to fix later. Briefly summarized, both the advantages and disadvantages of evolutionary design are present in BDD. And then Rusts strive for correctness by design called for its penalty from time to time. This mostly manifested in lengthy discussions on practical vs bullet proof API design (where the latter refers to an API which can not be misused).

A. Why not RSpec

Earlier we've disclosed that we opt against using RSpec for the demonstrator. The main reason for this decision is that the rust implementation of RSpec [¶] is not well maintained. However, that is for a reason: the Rust ecosystem already comes with great testing capabilities out of the box. These include unit tests and the excellent `cargo test` command line interface to invoke them. The results of unit tests are prefixed by the relevant module path, and if a speaking name is given to the unit test, some of the RSpec grouping semantics are met. Further on Rusts offers yet another way of writing tests. Rust doc strings may contain code blocks. This code blocks are displayed with syntax highlighting in the documentation, and illustrate the (intended) usage of the API. However, they serve as tests as well - if a code block contains code which does not compile or panics, this results in a failing test. Listing 3 demonstrates the use of this. It is a powerful tool to validate the documentation as well: while BDD documents tests as an artifact, this tests documentation - every example code snippet in the documentation is **guaranteed to execute!** We argue that this combined is more than sufficient to substitute RSpec for our use case.

B. The Aircraft as one Integrated System

Many avionic standards tend to describe the aircraft as one integrated system, as does DO-367. Hence, the requirements range from describing inner logic of the TAWS all the way up to the color of visual indicators. Not all properties of the system *aircraft* can be checked against a software library, which by its very nature does not allow for any physical interaction. Examples for this can often be identified by the description of any interaction between the pilot and the TAWS.

[§]<https://github.com/bbqsrc/cucumber-rust/issues/80>

<https://github.com/bbqsrc/cucumber-rust/issues/82>

<https://github.com/bbqsrc/cucumber-rust/issues/86>

[¶]<https://github.com/rust-rspec/rspec>

C. API Surface: Lean vs Testable

Some MOPS, for example MOPS_270, require that inner state of the TAWS is mutated. This creates a tricky problem. Implementation wise the Cucumber framework serves as integration test. Therefore, it is presented the same API as any other consumer of the library. The standard proclaims no use case where the library consumer arms/disarms *Mode 1*, hence it seems to be a reasonable decision to not offer any capability to modify the armed state of *Mode 1*. If however such a capability is not present, MOPS_270 can hardly be tested (as it requires *Mode 1* to be disarmed, while *Mode 1* is to be armed at any time).

A workaround for this could be conditional compilation, where the public API of the TAWS library is extended by functions to modify the inner state as needed by the Cucumber implementation. This however bears a new danger: Now the tests are testing a different API than what is to be used in production.

D. Invariants

Some requirements in the DO-367 standard are invariant, like MOPS 268 (Arming/Disarming Mode 1). While it is possible to test that some input values do not result in a violation, it is not possible to effectively prove that using any flavour of integration testing. The derivation of such proofs is subject to the formal methods domain and falls out of scope for BDD. Fortunately the DO-367 Standard declares the testing of representative input data to be sufficient for testing the MOPS, as written in [9]. Therefore this issue is not acute.

VI. Conclusion

This work uses Behavior Driven Development (BDD) to show a path towards agile safety critical airborne software engineering. With this experience report from a demonstrator developed using BDD, the methodology is introduced in airborne software engineering context.

For the use case, it is shown how high-level requirements can be captured and formalized using Gherkin, and how test automation with Cucumber can support verification and consistency checks for high-level requirements. Specifically, the verification objectives from the DO-178C software standard can be supported to check that high-level requirements are accurate and consistent and that high-level requirements are verifiable.

Moreover, different strategies for test generation are discussed. In addition to this use case demonstration, further research and testing of this concept is necessary. Future work includes further demonstrators with different tooling, such as C/C++, ADA or Model-Based Design tool chains. There is also a need to look into robustness testing requirements of DO-178C in more detail, and develop techniques to address them with BDD. Further, it is interesting to check how using BDD can support DO-178C verification objectives more detailed.

References

- [1] Wynne, M., Hellesoy, A., and Tooke, S., *The cucumber book: behaviour-driven development for testers and developers*, Pragmatic Bookshelf, 2017.
- [2] Air Force Life Cycle Management Center, "Software innovations makes F-16 more capable," <https://www.afllcmc.af.mil/News/Article-Display/Article/2165904/software-innovations-makes-f-16-more-capable/>, 2020. Accessed: 2020-12-01.
- [3] RTCA, "DO-178C Software Considerations in Airborne Systems and Equipment Certification," *RTCA*, 2011.
- [4] Zaeske, W., and Durak, U., "Leveraging Semi-formal Approaches for DepDevOps," *International Conference on Computer Safety, Reliability, and Security*, Springer, 2020, pp. 217–222.
- [5] Chelimsky, D., Astels, D., Helmkamp, B., North, D., Dennis, Z., and Hellesoy, A., "The RSpec Book: Behaviour Driven Development with Rspec," *Cucumber, and Friends, Pragmatic Bookshelf*, Vol. 3, 2010, p. 25.
- [6] Evans, E., *Domain-driven design: tackling complexity in the heart of software*, Addison-Wesley Professional, 2004.
- [7] Okolnychyi, A., and Fögen, K., "A study of tools for behavior-driven development," *Full-scale Software Engineering/Current Trends in Release Engineering*, 2016, p. 7.
- [8] Yackel, R., "BDD in DevOps: An Example of BDD in Continuous Integration," <https://www.qasymphony.com/blog/bdd-devops-example-bdd-continuous-integration/>, 2018. Accessed: 2020-05-20.

[9] RTCA, "DO-367 Minimum Operational Performance Standards (MOPS) for Terrain Awareness and Warning Systems (TAWS) Airborne Equipment," *RTCA*, 2017.