Validation-Driven Development

Sebastian Stock(✉), Atif Mashkoor, and Alexander Egyed

Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria
{Sebastian.Stock,Atif.Mashkoor,Alexander.Egyed}@jku.at

Abstract. Formal methods play a fundamental role in asserting the correctness of requirements specifications. However, historically, formal method experts have primarily focused on verifying those specifications. Although equally important, validation of requirements specifications often takes the back seat. This paper introduces a validation-driven development (VDD) process that prioritizes validating requirements in formal development. The VDD process is built upon problem frames - a requirements analysis approach - and validation obligations (VOs) - the concept of breaking down the overall validation of a specification and linking it to refinement steps. The effectiveness of the VDD process is demonstrated through a case study in the aviation industry.

Keywords: Validation-driven development · validation obligations · formal methods · Event-B

1 Introduction

Formal methods play a crucial role when developing critical systems, allowing a correct specification of the system behavior. This specification can be checked for consistency via verification that often takes preeminence in formal development. Consequently, techniques like model checking, theorem proving, and associated toolsets such as SPIN [12] or Isabelle [20] are widely used in industry. On the other hand, the compliance of the specification with desired system behavior can be ensured via validation. Validation is supported by techniques like animation and simulation and associated toolsets like AsmetaA [5] or JeB [18]. Contrary to verification, using validation techniques and toolsets is less common, especially in state-based formal methods [16]. Even if used, they are considered a secondary activity towards the end of the development cycle.

A typical formal requirements specification process starts with a set of (natural language) requirements. Once specified, requirements undergo a stringent verification process for consistency checking. Then, the validation process follows. The whole development process is iterative. Verification is often given preeminence over validation because it does not make sense to validate something

The research presented in this paper has been conducted within the IVOIRE project, which is funded by "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF) grant # I 4744-N and has been partly financed by the LIT Secure and Correct Systems Lab sponsored by the province of Upper Austria.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2023
Y. Li and S. Tahar (Eds.): ICFEM 2023, LNCS 14308, pp. 191–207, 2023.
https://doi.org/10.1007/978-981-99-7584-6_12

inconsistent. However, prioritizing verification over validation may lead to crucial issues, such as keeping the end users out of the loop. While specifiers create, verify, and validate specifications, the end users only give inputs at the specification process's beginning or end. Consequently, late feedback means more changes, efforts, and costs.

Many techniques have been proposed to overcome this problem. For example, Baumeister [4] suggested using test-driven development (TDD) for writing formal specifications. The author proposes generating run-time assertions from the specification to check for compliance between specification, code, and tests. Later, Bonfanti et al. [6] proposed using behavior-driven development (BDD) for a similar cause. The advantage of BDD over TDD is that it supports early collaboration among stakeholders, such as specifiers, developers, quality assurance experts, and end users, by giving low-level tests a high-level meaning. However, this reliance on tests comes with a price, and while testing is a valid means of validation, it is not necessarily exhaustive enough to cover all validation challenges. BDD restricts itself to a scenario language translated to some test in the target language (e.g., natural language to LTL formula), which may not be extensive enough to validate all properties of interest. This translation is often limited depending on the expressiveness of the target language. Furthermore, BDD is usually applied to formal specifications late in the process.

This paper proposes a validation-driven development (VDD) process for writing formal specifications that puts validation at the center of formal development. The VDD process focuses on creating validatable specifications, allowing end users to subjugate the formal specification process. Furthermore, VDD suggests a highly expressive and systematic structuring, elicitation, documentation, tracing, and maintenance process for formal requirements specifications appealing to all stakeholders.

The VDD process is built upon two well-known concepts: problem frames [13] and validation obligations (VOs) [17]. Problem frames help analyze requirements in a structured and collaborative manner. On the other hand, VOs help check the compliance of a specification concerning the stakeholders' requirements and support incremental specification writing and evolution. Analogous to proof obligations, VOs break down the overall validation of a specification and associate it with the specification's refinement steps.

The rest of the paper is structured as follows: Sect. 2 provides the necessary background to understand the content of this paper by introducing Event-B and VOs. However, note that the findings of this paper are language-independent. Section 3 introduces and exemplifies the VDD process. Section 4 demonstrates the application of the VDD process through a case study from the aviation domain. Section 5 compares the VDD process to other similar approaches. Finally, Sect. 6 concludes the paper with some proposed future work.

2 Background

2.1 Event-B

The formal language Event-B [1] is based on first-order predicate logic and set theory and helps with specification writing, verification, and proving using the platform Rodin [2]. The behavior of a specification is defined using `machines` that contain a set of `variables`, which are described in the `invariant` section. `Events` are considered state transitions, with a guard marked with the `when` clause that must be true before enabling the event. `Context` defines the static part of a specification. The Event-B language supports both vertical and horizontal refinement styles. While vertical refinement is about concretizing the abstract data structure, horizontal refinement is about introducing additional features to the specification.

2.2 Validation Obligations

Validation obligations (VOs) are logical formulas associated with the correctness claims of given validation properties. Each VO represents a requirement showing evidence of its existence in the specification. Figure 1 shows the internal components of a VO and their interplay. A validation expression (VE) is run against the specification and can consist of one or more validation tasks (VT) connected by the logical operators $\vee, \wedge$, and ;. The semicolon operator represents a validation expression where the components before and after the semicolon share the same state space of the specification. Thus, this operator allows for complex validation expressions where steps depend logically on each other. Typical validation techniques are animation, simulation, testing, or model checking. VTs have parameters determined by the requirements and structures of the specification. VOs help us with traceability, documentation, and maintenance throughout the specification as they act as tokens documenting how a requirement is realized in a specification. Further, they indicate when a requirement is no longer satisfied.

Let us consider the following requirement in a lift example where the operator can choose between multiple floors from 0 to 2. `REQ0`: *The floor level will eventually equal 2.* This requirement is implemented in specification `M0`. Suppose we choose LTL model checking as a validation technique. In that case, we can encode the requirement into the following VO, where the parameter is an LTL formula:

$$\texttt{REQ0/M0} : \texttt{LTL1} := \texttt{FG}(\{x = 1\}) \tag{1}$$

3 Validation-Driven Development

VDD proposes a systematic process for requirements elicitation, documentation, tracing, and maintenance during formal developments. In the following, we discuss the workflow of the VDD process, the role of VOs in specification writing, and the structuring of the specification through problem frames.

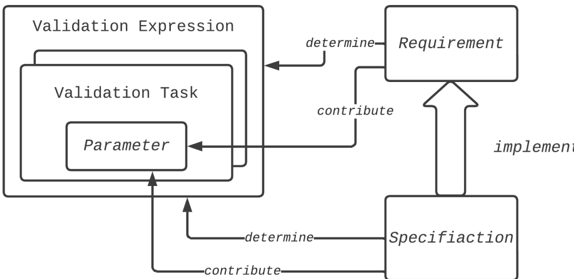


Fig. 1. Internal view of a VO

3.1 Workflow

Figure 2 shows the workflow of the VDD process, which is as follows:

1. Select a requirement.
2. Write a VO that, if successful, would give evidence for correctly implementing the requirement.
3. Implement the VO in the specification.
4. Verify the specification, e.g., check for internal consistency.
5. Run the VO, e.g., execute the associated validation task.

After satisfying a VO, the specifier can introduce a refactoring session to improve the existing specification. Overall, the approach is iterative for all requirements, and if we introduce additional VOs and change the specification, leading to other VOs failing, we have a hint of inconsistency. The VDD process also helps to keep things simple, i.e., if we need to introduce more than a handful of variables, state transitions, and invariants during the VO implementation, we most likely want too much at once. The VO makes this apparent. Checking multiple properties of one requirement hints that the requirement may be divided into sub-requirements.

Example. Let us specify `REQ0` from the previous lift example (step 1). For this, we first create the VO as shown in Eq. 1 (step 2). Now we need to implement the VO (step 3). We approach this as minimalist as possible. In the LTL formula, we need a variable `floor` which is some form of a number. Consequently, we start with `x` equaling 1. Then, we check the specification for internal consistency (step 4). If this is successful, we employ the LTL model checking to evaluate the VO (step 5).

3.2 Specification Structuring and Refinement

We now focus on problem structuring and refinement planning, two challenging tasks in formal developments [11]. The first challenge is to recognize what aspects of the problem are related to which other elements, i.e., eliciting the structure of

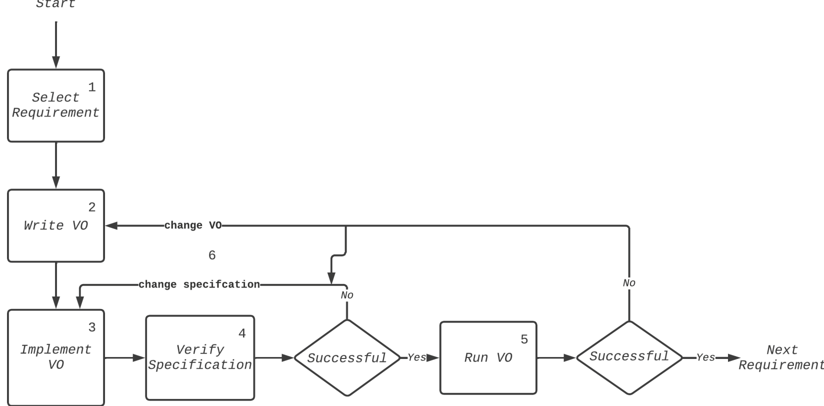
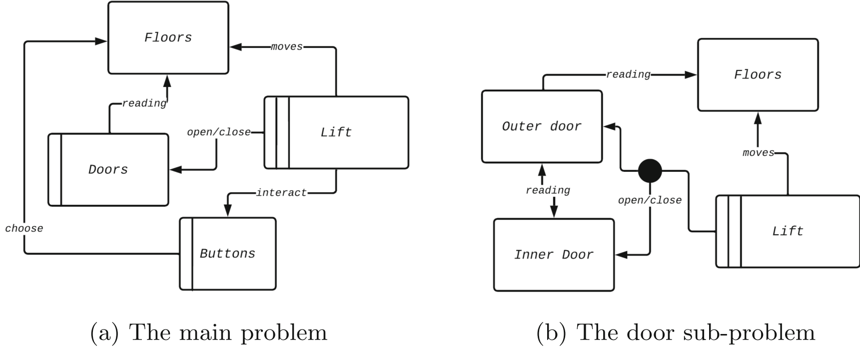


Fig. 2. VDD workflow

the specification. The second challenge is to derive a valid refinement structure from this, which supports the verification and validation process. However, both challenges require experience to master them.

Framing the Problem. We adapt the problem frame methodology [13] to structure specifications. Figure 3 shows the problem frame of the lift example. The rectangles represent the concerning domains. Each domain represents an aspect of the physical world that is observable to the stakeholders. Lines between the domains are interfaces and explain how the domains interact with each other. The rectangle with the doubled vertical stripe is the *machine domain*, the specification we want to write. Rectangles with one stripe are *designed domains* that represent the information we are free to express as we desire. This notation is for complex domains where the design is up to the specifier but where the details do not concern the global problem. Finally, rectangles with no stripe are *given domains*. Given domains are those we need to consider but cannot alter their appearance. They are usually very abstract for our specification purposes and require less attention. Our addition to the problem frames is the arrows on the interfaces indicating an information flow. Either they are uni- or bi-directional.

Example. In our running example in Fig. 3a, we want to specify a lift with three areas of concern. The `Floors` we want to navigate to are a given domain we cannot change. The lift `Doors` need to be detailed and marked as a designed domain. Finally, the `Buttons` is also a designed domain, as we have yet to get further instructions on how the buttons should look. Going into further detail, in Fig. 3b, we can see a sub-problem only concerning the lift's `Doors`. This sub-problem was separated as it would bloat Fig. 3 with information only specific to one domain. We can see that we replaced the `Doors` domain with two more specific domains related to each other. The `Outer Doors` are the doors on each floor. `Inner Door` is the lift's door and must read the outer door status to synchronize



(a) The main problem (b) The door sub-problem

Fig. 3. Problem frame of the lift problem and the sub-problem concerning `Doors`

accordingly. Furthermore, three domains share the `open/close` interface. The arrows at the interfaces show us the dependencies of their interaction, mainly the lift specification. In reality, domains are chosen and marked according to given and extracted information. This can lead to eliciting new requirements to fill gaps between the desired specification and reality. Moreover, we may involve non-technical stakeholders in the process due to a visual structure.

Structuring Specification. We use the following guidelines to structure the specification:

1. Domains sharing an interface will need to interact eventually. Therefore, they should refine each other horizontally (e.g., `Doors` is dependent on `Floors` in our problem frame and should refine it).
2. The first domain to be implemented is the one with the most connecting incoming interfaces. We then implement the domain with the second-highest incoming interfaces and proceed iteratively (e.g., specifying `Floors` before `Doors` as `Floors` has the most incoming interfaces).
3. Whenever we omit details in the main problem frame and create a sub-problem frame, we are confronted with a choice:
   (a) We can introduce the details immediately, substituting the domain with the domains introduced in the sub-problem (e.g., `Doors` is immediately specified as `Outer Doors` and `Inner Door`).
   (b) We can introduce the details later in a vertical refinement and keep the abstract domain around (e.g., `Doors` is refined to `Outer Doors` and `Inner Door`).
4. Whenever multiple domains share an interface without being connected otherwise, they may be related in a vertical refinement relationship.
5. Domains not directly connected to the machine domain are of secondary concern.

Structuring the specification further and fostering understanding for stakeholders involved, we can annotate domains with corresponding requirements

extracted from the requirements document. This helps later with the elicitation of VOs. We distinguish between two types of VOs: VOs focusing on the domain and VOs focusing on the interplay of two domains. Separating both concerns helps estimate the validation effort, as VOs focusing on the domain will likely still be valid if we change unrelated domains.

Example Continued. Applying these guidelines, we can derive a specification structure. For example, to specify the lift, we would start with the floors, as they are referenced most (Guideline 1). Next, we would specify the `Doors`. Here we are confronted with a choice. We can keep the `Doors` abstract for now and move on to the `Buttons` (Guideline 3b) or detail the `Doors` before moving on (Guideline 3a). The decision for either is dependent on the requirements we want feedback on. If we keep the `Doors` abstract (Guideline 3a), we can introduce the `Buttons` and gather early feedback on the whole system and the interaction between domains. On the other hand, if we choose to introduce the details of the `Doors` (Guideline 3b), we encounter a special case of two domains sharing an interface and being connected independently. The dependency structure is that `Outer Doors` and `Inner Door` complement each other as the bidirectional interface indicates. However, as a subproblem, they refine the `Doors` domain. Consequently, both domains are introduced at the same time. Therefore the problem frame helped us to evaluate the impact of possible specification structures.

Validation and Refinement. When introducing VOs early and then applying changes to the specification due to refinement or refactoring, we must tackle the (re)validation question. We can use the problem frame to indicate where revalidation might become necessary. For example, in horizontal refinement relationships, if we have an incoming interface, i.e., we consume information from another domain and change the producing domain, we must revalidate every VO consuming from this producing domain. Analogous is true for having producing domain. Adding to the insights proposed by Stock et al. [26] if a VO only concerns a single domain and is not dependent on others, outside changes do not invalidate it. For vertical refinement, rechecking VOs depends on the specification language. If the specification language has a strict notion of refinement, such as Event-B, where we can show the preservation of safety and liveness properties, our VOs will stay intact. For specification languages featuring a liberal notion of refinement, such as ASMs, we might recheck VOs. In some cases, the VO can be transferred, preserving its insights. For example, the works of Arcaini et al. [3] and Stock et al. [25] tackle the problem of information transfer, and the insights can be applied to VOs.

4 Case Study

4.1 System Description

We exemplify the VDD process on the Arrival Manager (AMAN) case study [19]. The AMAN system focuses on developing a human-machine interface for manag-

ing aircraft arriving at an airport. The particularity lies in continuously scheduling new aircraft to land at the airport while users can interact with the schedule on a screen in three different. The first interaction to consider is dragging the aircraft to another landing slot via the mouse. The second is blocking landing slots and disallowing the computer from scheduling aircraft in this slot. The third is to put the aircraft on *hold*, meaning that the countdown till landing is not reduced for these planes. Furthermore, the user can zoom in and out on the landing schedule, thus reducing or increasing the presented slots and aircraft, respectively. Figure 4 shows the working of the AMAN system. In the middle, one can see the remaining time till landing, and the boxes on the left and right are planes. Colors indicate different statuses, for example, *hold*.

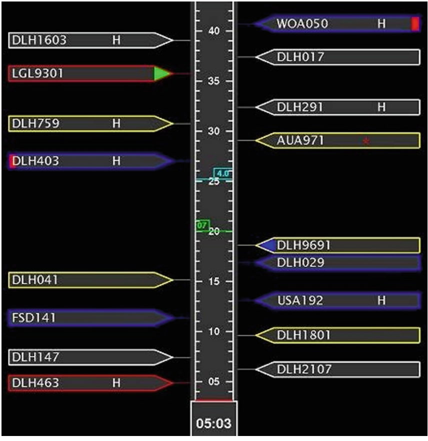


Fig. 4. Screenshot of the AMAN system [15]

4.2 Problem Structuring

This subsection demonstrates how the requirements of the AMAN system can be specified using the VDD process. We use the problem frames approach introduced in Sect. 3.2 to understand and define the problem. For brevity, only a portion of the case study and the validation process is shown here. For the complete specification and the VOs derived, please consider the work of Geleßus et al. [7].

Defining Domains of Interest. Consider Fig. 5a, the AMAN we want to specify is marked as the centerpiece by the two extra bars inside; this is the goal of the specification process. Next to the AMAN are designed domains partially mentioned in the system description. Here, we have the designed domain `User`, which encapsulates the user behavior. For example, the AMAN reacts to the

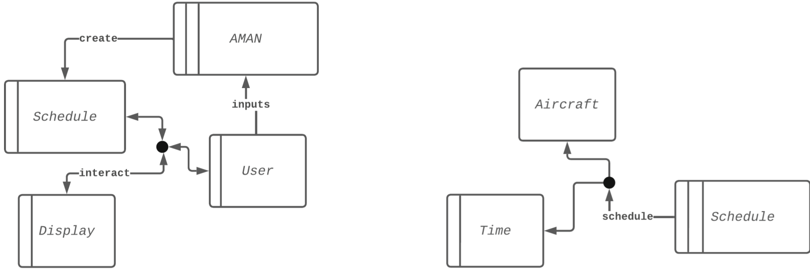
user input. We designated User as a designed domain because we know about some user behavior, but we are unaware of the details and might want to create a sub-problem frame. Then there is the designed domain of Schedule, which encapsulates the process of the AMAN creating a schedule from aircraft and time slots. We marked Schedule as a designed domain as we are not sure of the structure and behavior of the schedule and want to investigate further. Finally, we have the designed domain Display that works as a transmitter as a physical way of transmitting user inputs to the Schedule. However, the lack of an interface with the AMAN suggests its secondary role.

Sub-problem Structure. Diving deeper into the designed domains, we start with the sub-problem shown in Fig. 5b. Focusing on the Schedule itself, we now consider the Schedule's two components: Time, which is again a designed domain, and Aircraft, a given domain. We decided here that Aircraft is a given domain as no detail about Aircraft is available. Therefore, we consider it a rather primitive datatype. On the other hand, Time is complex and might require much consideration. Both tie into the Schedule domain, which, according to the proposed guidelines, indicates a refinement. Additionally, both have the same amount of incoming interfaces. Therefore, we can start specifying with any of them.

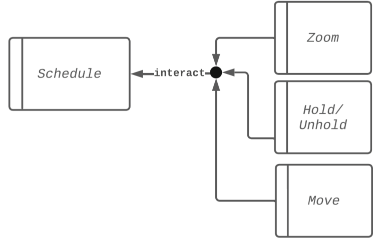
The second sub-problem in Fig. 5c covers the topic of user interaction. Here the domain structure is simple. However, all sub-domains need the Schedule, and additional domains share the interaction interface, which indicates some interference in the domains. Otherwise, the domains remain very loosely connected. What could be a consideration is that we define abstract User interaction that interacts with the Schedule and later refines the User interaction into the three subdomains. This, again, depends on how we define the scheduling.

Final Specification Structure. We can use the proposed guidelines discussed in Sect. 3.2 to derive a specification structure from these initial problem frames. Considering incoming interfaces, starting with the Schedule seems reasonable. We must decide if we detail the Schedule before implementing User interaction. An argument for this would be that we can validate the most basic function of the AMAN and get feedback on it. Further, we tackle the difficult representation of time early. Afterward, we may implement the User interaction. We subjugate the choice of what to implement first to what needs the most investigation and validation effort, as the individual User interactions only are loosely connected. Finally, we can conclude with the specification of the Display properties. The Display has no direct connection to the primary concern of the AMAN system. Therefore, its specification is a secondary concern.

The final specification structure is as follows:



(a) Problem frame of the `AMAN` (b) Sub-problem concerning the `Schedule`



(c) Sub-problem concerning the user interaction

Fig. 5. Problem frame of the AMAN and a sub-problem frame concerning the scheduling

1. Create the `Schedule` (Guideline 2):
   (a) Introduce the `Aircraft` domain (Guideline 3a)
   (b) Vertically refine the created specification by introducing `Time` (Guideline 3a & 4)
2. Horizontally refine the specification by introducing `User` interactions (Guideline 1 & 3a) and consequently `Zoom`, `Hold/Unhold`, and `Move` in any order (Guideline 1)
3. Horizontally refine the specification by introducing `Display` (Guideline 5)

4.3 Specification and Validation

We start the specification process with the `Schedule` sub-problem. In the following, we refer to requirements directly derived from the specification as a direct quote: `REQX` with `X` being a number. According to the tactics presented in Sect. 3.1, we start by selecting a requirement, creating a VO, and then specifying the requirement. For example, let's assume we select the requirement of `REQ1:` "Planes can be added to the flight sequence, e.g., planes arriving in close range of the airport." This requirement means: a) we have aircraft, b) we have something

```
1 machine M0_AMAN_Update sees M0_AMAN_Update_Ctx
2
3 variables scheduledAirplanes
4
5 invariants
6   @inv0,1 scheduledAirplanes ⊆ AIRPLANES
7
8 events
9   event INITIALISATION
10    then
11      @act0,1 scheduledAirplanes ≔ ∅
12  end
13
14  event AMAN_Update
15    any newScheduledAirplanes
16    where
17      @grd0,1 newScheduledAirplanes ⊆ AIRPLANES
18    then
19      @act0,1 scheduledAirplanes ≔ newScheduledAirplanes
20  end
```

Fig. 6. The schedule sub-problem with only aircraft

to store them, and c) we can manipulate this storage by adding planes. Let's formulate this as a VO:

$$\texttt{REQ1/MO}: \texttt{GF}(\texttt{BA}(\texttt{scheduledAirplanes} \neq \texttt{scheduledAirplanes\$0})) \implies \texttt{GF}(\texttt{BA}(\{\exists x.(x \in \texttt{scheduledAirplanes} \wedge x \notin \texttt{scheduledAirplanes\$0})\}))$$

The `GF` (Globally-Finally) operator indicates that the brackets' expression will eventually be true. The `BA` is the `before-after` operator, comparing the current version of a variable with the previous version marked with an `$0`, i.e., the difference between `scheduledAirplanes` in one step and the next step is observed. The LTL formula will ensure that our scheduled aircraft can contain an aircraft not previously in the set of scheduled aircraft. This, however, implies some state transition in our specification, going from an initial state to a state with one more aircraft that was not previously contained.

Figure 6 is an Event-B specification that attempts to satisfy the VO. We have a variable representing our `Schedule`, an `AIRPLANE` datatype, and an event creating a new schedule, eventually satisfying the VO. We could now generate more VOs to ensure soundness implementation regarding the amount of added planes. For now, we are satisfied and proceed.

Taking a look back at Fig. 5b, we need to implement the `Time` domain to cover the `Schedule` domain fully. The corresponding requirement we want to satisfy by introducing the time is `REQ5:` "The space between two aircraft is always $\geq$ 3, with 3 being the time in minutes." Following is the corresponding VO.

```
 1 machine M1_Landing_Sequence refines M0_AMAN_Update  sees M1_Landing_Sequence_Ctx
 2
 3 variables landing_sequence
 4
 5
 6 invariants
 7   @inv1,1 landing_sequence ∈ AIRPLANES ⇸ PLANNING_INTERVAL
 8   @inv13,2 ∀a1,a2 · a1 ∈ dom(landing_sequence) ∧
 9             a2 ∈ dom(landing_sequence) ∧ a1 ≠ a2 ⇒
10            (DIST(landing_sequence(a1) ↦ landing_sequence(a2))
11            ≥ AIRCRAFT_SEPARATION_MIN)
12   @glue1,1 scheduledAirplanes = dom(landing_sequence)
13
14 events
15   event INITIALISATION
16     then
17       @act1,1 landing_sequence ≔ ∅
18   end
19
20   event AMAN_Update refines AMAN_Update
21     any new_landing_sequence
22     where
23       @grd1,1 new_landing_sequence ∈ AIRPLANES ⇸ PLANNING_INTERVAL
24       @inv1,2 ∀a1,a2 · a1 ∈ dom(new_landing_sequence) ∧
25                a2 ∈ dom(new_landing_sequence) ∧ a1 ≠ a2 ⇒
26               (DIST(new_landing_sequence(a1) ↦ new_landing_sequence(a2))
27               ≥ AIRCRAFT_SEPARATION_MIN)
28     with
29       @wit1,1 newScheduledAirplanes = dom(new_landing_sequence)
30     then
31       @act1,1 landing_sequence ≔ new_landing_sequence
32   end
33 end
```

Fig. 7. The schedule sub-problem with added time

$$\begin{aligned}\texttt{REQ5/M1}:&\forall \texttt{a1},\texttt{a2}\cdot \texttt{a1} \in \texttt{dom}(\texttt{landing_sequence}) \land \\ &\texttt{a2} \in \texttt{dom}(\texttt{landing_sequence}) \land \texttt{a1} \neq \texttt{a2} \implies \\ &(\texttt{DIST}(\texttt{landing_sequence}(\texttt{a1}) \mapsto \texttt{landing_sequence}(\texttt{a2})) \\ &\geq \texttt{AIRCRAFT_SEPARATION_MIN})\end{aligned}$$

For this VO, we assumed that we upgraded our `scheduledAirplanes` from Fig. 6 to `landing_sequence` as shown in Fig. 7, which is a mapping from aircraft to time slots. Consequently, we demand that every aircraft contained in this mapping has a distance (`DIST`) to every other aircraft of `AIRCRAFT_SEPARATION_MIN`, which in our case is 3. Consequently, we must upgrade our `scheduledAirplanes` and take care of the proof.

Figure 7 shows the corresponding specification. We introduced the mentioned `landing_sequence` and further introduced `inv13,2` to establish proof. Furthermore, we refined our `event` to use the upgraded data structure. After discharging the proof, we establish that our requirement is truly represented in the specification.

As previously established, both domains `Aircrafts` and `Time` have a connection, and therefore when creating `M1`, we need to show that `REQ1` is still preserved in the specification. As Rodin only supports a safety-preserving notion of refinement, re-establishing the VO must happen by re-executing the LTL formula.

After completing the scheduling sub-problem, we move on to the `User` interaction part. Our VOs concerning the `Schedule` will not be revalidated when validating the interaction. We only consume the `Schedule`'s behavior as laid out at the end of Sect. 4.2.

5 Related Work

Several approaches have been proposed for the validation of requirements specifications. While some focus on the whole specification process, others focus only on certain aspects. We briefly introduce and compare some of them with our proposed process.

5.1 BDD Usage in Formal Requirement Specification

BDD [24] is a well-established technique in the area of software development. It is appealing due to its easy-to-follow procedure and its effectiveness in establishing that requirements are part of the code. First, a scenario is created and run (with intermediate steps) against the code. If this is successful, the next scenario is tackled. If it fails, either code or scenario has to be fixed. One strength is the imposed iterative nature, which comes naturally by adding more satisfied scenarios. Furthermore, tracing and maintaining requirements is massively simplified as every requirement has one scenario mapped to a group of tests. Naturally, attempts have been made to use BDD in formal developments.

There are many significant adaptations of the BDD approach for the formal specification community. For example, Snook et al. [23] proposed an Event-B targeting version of Cucumber [27] to describe scenarios in the Gherkin[1] language which is translated into a trace and executed against a specification. The scenario language FRETISH [9] goes in a similar direction as it can be used to express requirements which are then converted to an LTL formula with the help of the FRET [8] tool. This approach orients itself heavily on what Gehrkin does for programming examples. It provides a basic language to write scenarios, which can be (automatically) linked to LTL formulas.

While these approaches can be applied successfully, they suffer from two drawbacks. First, they consider validation after writing specifications, thus losing out on the advantages of validation-centered specifications. Doing validation last will compromise completeness due to time constraints or the complexity of the specification. Second, scenario language used in BDD causes problems of expressiveness and, therefore, suffers from a lack of completeness. Second, while these approaches work well, they only provide one solution to a validation problem. We must rely on the correct translation from the scenario language to the validation technique. Furthermore, there is no way to choose between different validation techniques to translate the scenario. This means a method like FRETISH can only react to a scenario by producing an LTL formula. However, model checking may not always be a good solution, e.g., in infinite state spaces.

[1] https://cucumber.io/docs/gherkin/.

VDD addresses both concerns while keeping the compact and easy-to-follow style of BDD. First, it puts validation at the center of the formal development process. Second, it offers a liberal syntax allowing for expressing and consequently validating different properties of interest with many techniques and tools.

Arcaini et al. [3] showed how BDD-like scenarios targeting ASMs can be transferred between refinement steps of abstract state machines. While the previously mentioned disadvantages to using BDD-like scenarios apply, this work highlights the importance of the transferability of validation results. In the context of VDD, with our approach, we know early when results are transferable or might be due to revalidation, as pointed out at the end of Sect. 4.2.

5.2 Bridging the Gap Between Natural Language Requirements and Formal Specification

Several efforts have been made to narrow the gap between natural language requirements and formal specifications, as it can reduce the mental load placed on the specifier, and it helps when attempting to involve non-technical stakeholders. The efforts are bidirectional: creating specifications from natural language requirements and validating natural language requirements in specifications. As discussed in Sect. 5.1, BDD for formal specifications caters to the latter concern.

Regarding creating specifications from natural language requirements, Golra et al. [10] focus on creating intermediate steps with meta-models for systematically translating requirements to formal specifications. A second work of Sayer et al. [21,22] uses translation patterns. However, as both approaches introduce intermediate layers of abstraction, they also introduce additional error sources where the translation could be wrong. Furthermore, they may suffer from the same problems discussed in Sect. 5.1, where the intermediate language might not be powerful enough to translate the constructs.

VDD does not introduce intermediate layers but changes the standard order from specification first to validation. Therefore no new error source was introduced. Furthermore, the mental load is reduced as the problem is tackled in smaller portions. Finally, with VOs, non-technical stakeholders can get a feeling for the progress the specification made and point to requirements that still need work.

5.3 Requirements Tracing

Another field of interest is systematically tracing the implementation status of requirements. Exculpatory for these efforts are, for example, the works [11,14], where a sophisticated set-theoretic representation for requirements is proposed, which is supposed to help with the tracing of requirements. Compared to our approach, the authors heavily focus on the properties of Event-B and proofing with proof obligations. Validation is a gap filler for everything that cannot be proven. While our work also contributes to traceability, it takes a more lightweight approach inspired by software development strategies and thus is more intuitive.

Furthermore, the focus is on validating and creating validatable specifications, not fitting a validation solution to an existing specification.

6 Conclusion and Future Work

This paper presents the validation-driven development process for writing formal specifications. It offers an iterative approach to formal specifications, strongly focusing on their validation. The aim is to provide a systematic process to structure, elicit, document, trace, and maintain formal requirements specifications. To this end, we employ an adapted version of problem frames complemented by validation obligations.

In the future, we want to provide tool support that helps automate the VDD process by keeping track of VOs, the specification structure, and changes. Especially the steps of VOs elicitation and creation could be fully automated.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010)
3. Arcaini, P., Riccobene, E.: Automatic refinement of ASM abstract test cases. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1–10 (2019)
4. Baumeister, H.: Combining formal specifications with test driven development. In: Zannier, C., Erdogmus, H., Lindstrom, L. (eds.) XP/Agile Universe 2004. LNCS, vol. 3134, pp. 1–12. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27777-4_1
5. Bonfanti, S., Gargantini, A., Mashkoor, A.: AsmetaA: animator for abstract state machines. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 369–373. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_25
6. Bonfanti, S., Gargantini, A., Mashkoor, A.: Generation of behavior-driven development C++ tests from abstract state machine scenarios. In: Abdelwahed, E.H., et al. (eds.) MEDI 2018. CCIS, vol. 929, pp. 146–152. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02852-7_13
7. Geleßus, D., Stock, S., Vu, F., Leuschel, M., Mashkoor, A.: Modeling and analysis of a safety-critical interactive system through validation obligations. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) ABZ 2023. LNCS, vol. 14010, pp. 284–302. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33163-3_22
8. Giannakopoulou, D., Mavridou, A., Rhein, J., Pressburger, T., Schumann, J., Shi, N.: Formal requirements elicitation with FRET. In: International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020) (2020)

9. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Generation of formal requirements from structured natural language. In: Madhavji, N., Pasquale, L., Ferrari, A., Gnesi, S. (eds.) REFSQ 2020. LNCS, vol. 12045, pp. 19–35. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-44429-7_2
10. Golra, F.R., Dagnat, F., Souquières, J., Sayar, I., Guerin, S.: Bridging the gap between informal requirements and formal specifications using model federation. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 54–69. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_4
11. Hallerstede, S., Jastram, M., Ladenberger, L.: A method and tool for tracing requirements into specifications. Sci. Comput. Program. **82**, 2–21 (2014). Special Issue on Automated Verification of Critical Systems (AVoCS'11)
12. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23**(5), 279–295 (1997)
13. Jackson, M.: Problem Frames: Analysing and Structuring Software Development Problems. Addison-Wesley, Boston (2001)
14. Jastram, M., Hallerstede, S., Leuschel, M., Russo, A.G.: An approach of requirements tracing in formal refinement. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 97–111. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15057-9_7
15. Martinie, C., Palanque, P., Pasquini, A., Ragosta, M., Rigaud, E., Silvagni, S.: Using complementary models-based approaches for representing and analysing ATM systems' variability. In: 2nd International Conference on Application and Theory of Automation in Command and Control Systems (ATACCS 2012), Toulouse, pp. 146–157. IRIT Press (2012)
16. Mashkoor, A., Kossak, F., Egyed, A.: Evaluating the suitability of state-based formal methods for industrial deployment. Softw. Pract. Exp. **48**(12), 2350–2379 (2018)
17. Mashkoor, A., Leuschel, M., Egyed, A.: Validation obligations: a novel approach to check compliance between requirements and their formal specification. In: ICSE 2021 NIER, pp. 1–5 (2021)
18. Mashkoor, A., Yang, F., Jacquot, J.: Refinement-based validation of Event-B specifications. Softw. Syst. Model. **16**(3), 789–808 (2017)
19. Palanque, P., Campos, J.C.: Aman case study. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) ABZ 2023. LNCS, vol. 14010, pp. 265–283. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33163-3_21
20. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer, Heidelberg (1994). https://doi.org/10.1007/BFb0030541
21. Sayar, I., Souquières, J.: Bridging the gap between requirements document and formal specifications using development patterns. In: 2019 IEEE 27th International Requirements Engineering Conference Workshops (REW), pp. 116–122. IEEE (2019)
22. Sayar, I., Souquières, J.: Formalization of requirements for correct systems. In: 2020 IEEE Workshop on Formal Requirements (FORMREQ), pp. 28–34. IEEE (2020)
23. Snook, C., et al.: Behaviour-driven formal model development. In: Sun, J., Sun, M. (eds.) ICFEM 2018. LNCS, vol. 11232, pp. 21–36. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-02450-5_2
24. Solis, C., Wang, X.: A study of the characteristics of behaviour driven development. In: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 383–387 (2011)

25. Stock, S., Mashkoor, A., Leuschel, M., Egyed, A.: Trace refinement in B and Event-B. In: Riesco, A., Zhang, M. (eds.) ICFEM 2022. LNCS, vol. 13478, pp. 316–333. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17244-1_19
26. Stock, S., Vu, F., Geleßus, D., Leuschel, M., Mashkoor, A., Egyed, A.: Validation by abstraction and refinement. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) ABZ 2023. LNCS, vol. 14010, pp. 160–178. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33163-3_12
27. Wynne, M., Hellesoy, A., Tooke, S.: The cucumber book: behaviour-driven development for testers and developers. Pragmatic Bookshelf (2017)