Modeling and Analysis of a Safety-Critical Interactive System Through Validation Obligations

David Geleßus[1](✉), Sebastian Stock[2], Fabian Vu[1], Michael Leuschel[1], and Atif Mashkoor[2]

[1] Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, 40225 Düsseldorf, Germany
{dagel101,fabian.vu,leuschel}@uni-duesseldorf.de

[2] Institute for Software Systems Engineering, Johannes Kepler University Linz, Altenbergerstr. 69, 4040 Linz, Austria
{sebastian.stock,atif.mashkoor}@jku.at

Abstract. This paper presents insights gained during modeling and analyzing the arrival manager (AMAN) case study in Event-B with validation obligations (VOs). AMAN is a safety-critical interactive system for air traffic controllers to organize the landing of airplanes at airports. The presented model consists of a human-machine interface comprising interactive and autonomous parts. We employ VOs to formalize requirements, uncover contradictions and ambiguities, and validate the model's compliance with the requirements. To capture the AMAN's human-machine interaction, we implement an interactive domain-specific visualization and an automatic simulation using the VisB and SimB components of ProB.

Keywords: Event-B · Refinement · Validation Obligations · Simulation · Visualization

1 Introduction

In this work, we model the arrival manager (AMAN) case study presented by Palanque and Campos [9]. AMAN is a semi-interactive tool consisting of interactive/human and autonomous parts. While AMAN automatically computes a landing sequence for the arriving airplanes, a human can manually intervene and change this sequence. An important aspect is that the user's inputs are prioritized over the system events.

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. The work of Sebastian Stock and Atif Mashkoor has been partly funded by the LIT Secure and Correct Systems Lab sponsored by the province of Upper Austria.

For our model, we use the Event-B [1] modeling language, which has been deemed effective in earlier works to model interactive safety-critical systems, including human-machine interfaces, e.g., by Singh et al. [3] and Ait-Ameur et al. [11].

The model itself was developed with the Rodin [2] platform. We provide rigorous evidence for the consistency of our model via model checking with ProB [7] and proof obligations. However, our primary focus is on systematically validating the requirements and appropriately presenting results to non-modelers to foster their understanding and contribution to the modeling effort.

To this end, we employ validation obligations (VOs) and use a management system and validation tools implemented in ProB2-UI [4]. For domain-specific views that foster stakeholders' understanding of the model, we use visualization via VisB [16] and simulation via SimB [15].

The rest of the paper is organized as follows: Sect. 2 presents the AMAN model in Event-B. Section 3 reports on the verification via model checking and POs. Section 4 describes the validation of the model via VOs. Section 5 reports our experiences using domain-specific views to tackle the interactive nature of AMAN. Section 6 highlights the lessons learned during this modeling and analysis exercise, showing parts of the specification where VOs helped to formulate questions for the stakeholders, make assumptions and uncover ambiguities. Finally, we conclude in Sect. 7.

2 AMAN Model

Our model[1] focuses on the software-related aspects of AMAN and, to some extent, the GUI itself. The specification [9] also describes autonomous, hardware, and human aspects, which we did not model in detail. Our model structure was guided by the HAMSTERS diagrams from the specification, and our refinement structure up to `M5` (cf. Figure 1) has a correspondence with Figs. 5 and 10 in the specification [9].

At the abstract levels, we model autonomous AMAN updates for the landing sequence (`M0` and `M1`). In the next steps, we introduce user inputs in an abstract manner (`M2` to `M4`). In `M5`, we model timeouts of AMAN updates. In `M6` to `M9`, we refine the abstract user events into *mouse movements*, *mouse clicks*, *mouse drags*, and *mouse releases*. The final refinement, `M10`, models a concrete pixel representation of all graphical UI elements.

AMAN Update and Landing Sequence (M0, M1). In `M0`, we introduce the event `AMAN_Update`, which manages the set of airplanes scheduled for landing. This event (and its refinements) encapsulate the autonomous part of the AMAN; all other events in our model are related to interactive user activities. In `M1`, the set of scheduled airplanes is refined to a landing *sequence* with associated landing times (relative to the current time; see discussion in Sect. 3). Furthermore, `M1` adds the

[1] The model and all other mentioned files are available at https://github.com/hhu-stups/AMAN-case-study/tree/bd044670a02092643230d6001cc2b355a2dc350a.

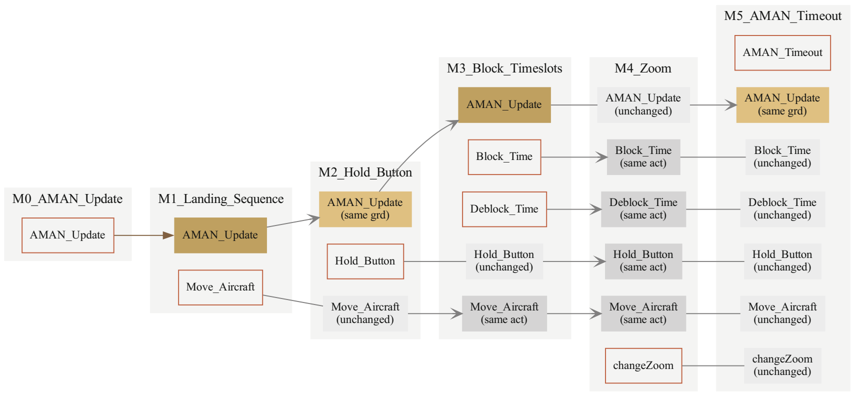


Fig. 1. Event Refinement Hierarchy until M5 (generated by ProB)

ability for the planning air traffic controller (PLAN ATCo) to move an airplane to another time slot via the `Move_Aircraft` event with respective parameters `aircraft` and `time`. M1 also introduces an important invariant stating that airplanes must be separated by at least three minutes. This invariant is preserved by both events `AMAN_Update` and `Move_Aircraft`.

Holding Airplanes (M2). `M2` introduces the *hold button*. First, we model the set of held airplanes (`held_airplanes`), a subset of airplanes in the landing sequence. The new `Hold_Button` event is introduced to add an individual plane to this set. A future `AMAN_Update` is expected to remove held airplanes from the landing sequence, which also removes them from `held_airplanes`. However, an airplane on hold can be rescheduled to another time slot.

Blocking Time Slots (M3). In the third refinement, `M3`, time slots can be blocked (stored in the variable `blockedTime`). The events `Block_Time`/`Deblock_Time` block/deblock an individual time slot. Regarding the events `AMAN_Update` and `Move_Aircraft`, we must ensure that neither AMAN nor the PLAN ATCo can move an airplane into a blocked time slot. However, we cannot posit $\texttt{ran}(\texttt{landing\_sequence}) \cap \texttt{blockedTime} = \emptyset$ as an invariant because the user could block a time slot still holding a scheduled plane (and thus violate the property). To overcome this, we introduced this conditional invariant: $\texttt{blockedTimesProcessed} = \texttt{TRUE} \Rightarrow \texttt{ran}(\texttt{landing\_sequence}) \cap \texttt{blockedTime} = \emptyset$ (see `Req6` and Eq. (2)). Here, `blockedTimesProcessed` is a helper variable that is set to TRUE by `AMAN_Update` and can be set to FALSE by `Block_Time`.

Zooming (M4). `M4` introduces the `changeZoom` event which updates the variable `zoomLevel`. Interactions with time slots and airplanes are restricted to the current zoom level. As shown in Fig. 1, this is encoded by adding guards to the interaction events but leaving the actions unchanged. Note that zoom does not affect AMAN's autonomous activities — AMAN can still schedule airplanes for a time slot that is not visible to the PLAN ATCo.

Timeout (M5). M5 introduces timeouts for AMAN updates (AMAN_Timeout event) which set a boolean variable timeout. The event is meant to occur when the AMAN does not respond within the expected deadline of 10 s. In this case, all user interactions are disabled, and the user interface provides feedback that the AMAN is no longer working.

Detailed User Interaction (M6, M7, M8, M9). M6 adds two events for the PLAN ATCo to select/deselect an airplane: selectAirplane and deselectAirplane. This is necessary to refine mouse events in the next steps. Furthermore, this also helps to set up a VisB visualization (see Sect. 5). The selected airplane is stored in the selectedAirplane variable. Holding and moving an airplane are refined to perform both events on selectedAirplane. When an AMAN update occurs, the selected airplane is cleared.

M7 implements the dragging of airplanes via a boolean variable dragging_airplane. Whenever an airplane label is selected, the dragging process starts. Furthermore, we introduce two events resume_dragging_airplane and stop_dragging_airplane for resuming/stopping dragging. As described in the specification, user interactions have priority over system events. Therefore, we ensure that AMAN updates do not occur while the user drags an airplane.

M8 refines M7 by adding more details to the dragging behavior. First, we replace dragging_airplane with dragged_airplane representing a specific airplane instead of a boolean variable. Second, we implement dragging behavior for the zoom slider by two new events: start_dragging_zoom_slider and drag_zoom_slider.

M9 implements mouse behavior including *mouse movement*, *mouse clicks*, *mouse drags*, *mouse releases*. These refinements were challenging because many variables were introduced, and some events were split into sub-events, as mentioned earlier. In particular, the mouse position and all allowed combinations with user interactions must be tracked.

Concrete Graphical Interface (M10). M10 models a raster-based UI rendered on a screen. Concrete pixel coordinates are set for all UI elements. Moreover, a variable mouse_pos tracks the pixel position of the mouse cursor. Events were added and extended to allow moving the mouse, and many user interaction events were restricted to execute only if the mouse is positioned appropriately. For example, a mouse click on the hold button is only registered if the mouse_pos is inside the button's pixel area. The modeled pixel coordinates for the UI elements match the design of our VisB visualization (see Sect. 5). Due to its complexity, we have not yet finished modeling this final refinement step — e.g. dragging of airplanes is not fully refined yet.

3 Verification

In this section, we evaluate the applicability of proving and model checking to verify the AMAN model.

Proving. When modeling within Rodin, proof obligations (POs) are automatically generated from the model. Afterward, provers in Rodin are applied to discharge them. This includes POs ensuring that the model's invariants are maintained (for more details, see Sect. 4), the absence of well-definedness errors, and the consistency between the refinement steps.

Table 1. Proof Statistics in Rodin

Machine	Total	Automatic	Manual	Undischarged
M0_ctx	0	0	0	0
M0	0	0	0	0
M1_ctx	3	2	1	0
M1	13	12	1	0
M2	4	4	0	0
M3	9	9	0	0
M4_ctx	0	0	0	0
M4	4	4	0	0
M5	0	0	0	0
M6	25	24	1	0
M7	10	10	0	0
M8	74	61	13	0
M9_ctx	0	0	0	0
M9	306	294	12	0
M10_ctx	54	17	37	0
M10	250	163	85	2
Total	752	600	150	2

Table 1 shows the number of POs in all refinement steps of our AMAN model (including automatic, manual, and unproven POs). Because `M10` is still in development, the total number of POs is not yet finalized. 600 out of 752 POs are proven automatically, while 150 POs are proven manually. As all POs from `M0` until `M9` are discharged, we achieved strong guarantees regarding the aforementioned properties covered by POs until `M9`.

Proving provides limited feedback when a PO cannot be discharged. Often, one must determine whether a PO cannot be discharged because the provers are too weak or whether the underlying proposition is false. As support, we use ProB [7], including its animation, disproving, and model-checking capabilities, to discover errors and inspect counter-examples. In particular, we can inspect concrete traces where, e.g., an invariant is violated. After discharging all POs, we proceeded to the validation part (see Sect. 4).

Table 2. Model Checking Statistics with ProB with Number of States, Transitions, Runtime (in Seconds), and Memory (in MB)

Machine	States	Transitions	Time [s]	Memory [MB]
M0_inst_1	9	66	0.30	158.87
M1_inst_1	1505	2 287 908	208.21	1424.12
M2_inst_1	9884	15 045 795	1468.97	8352.65
M3_inst_1 - M9_inst_1	–	–	> 3600.00	–
M0_inst_2	5	18	0.28	158.85
M1_inst_2	18	339	0.31	159.24
M2_inst_2	46	913	0.32	159.62
M3_inst_2	1953	49 154	1.80	186.81
M4_inst_2	1953	49 154	1.89	186.91
M5_inst_2	3905	102 210	2.87	211.05
M6_inst_2	9665	256 962	5.87	284.93
M7_inst_2	15 425	297 282	6.90	299.42
M8_inst_2	48 129	611 970	16.44	460.74
M9_inst_2	687 169	10 224 194	280.85	3994.74

Model Checking. As mentioned, we used model checking to complement proving, and to find definite errors before full proof was achieved. Timing aspects in AMAN could have been modeled by an increasing variable representing the current time. However, this would lead to infinite state space. Therefore, we model timing aspects as follows: the current time is always 0, and all times are relative to the current time point. This renders the state space finite concerning timing (cf. [5,10]), but other aspects still render exhaustive model checking intractable. We instantiated M0 to M9 with specific values for the constants (e.g., for the number of aircraft or the amount of zooming possible), to make exhaustive model checking feasible[2].

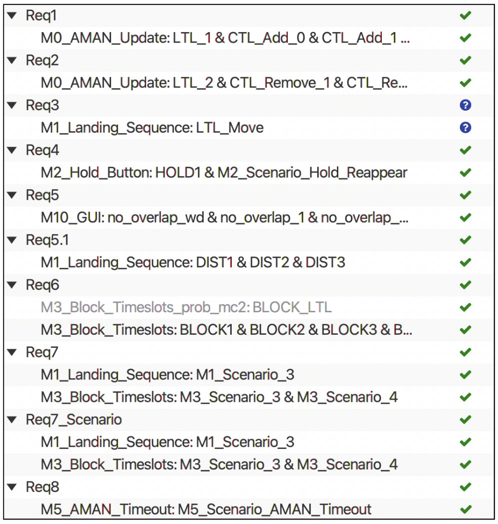
Table 2 shows the model checking results. The first configuration (*_inst_1) restricts the model to a single zoom level value of 15 (rather than allowing seven values from 15 to 45) and to only three different planes. In the second configuration (*_inst_2), we reduce the single zoom level to 5 and only two airplanes. We could not model check M10 in this way — the GUI model cannot be instantiated with these reduced configurations because it requires specific values for some constants. ProB was used to check all machines for invariant violations and deadlock-freedom[3]. Furthermore, we activated the new operation reuse feature [6] together with state compression to increase the performance (`-p OPERATION_REUSE full -p COMPRESSION TRUE`).

All experiments were run five times with ProB version 1.12.0-nightly[4], built with SICStus 4.7.1 (arm64-darwin-20.1.0) on a MacBook Pro (14", 2021) with

[2] Note that even on infinite state spaces, model checking can be useful in detecting errors. We did apply ProB also to the un-instantiated models.

[3] Note that deadlock-freedom is only verified by model checking.

[4] Revision `f41dfd4b29c7bd95583dffcb0adad44171f4f0c0` from 2023-01-10.



▼ Req1	✓
M0_AMAN_Update: LTL_1 & CTL_Add_0 & CTL_Add_1 ...	✓
▼ Req2	✓
M0_AMAN_Update: LTL_2 & CTL_Remove_1 & CTL_Re...	✓
▼ Req3	?
M1_Landing_Sequence: LTL_Move	?
▼ Req4	✓
M2_Hold_Button: HOLD1 & M2_Scenario_Hold_Reappear	✓
▼ Req5	✓
M10_GUI: no_overlap_wd & no_overlap_1 & no_overlap_...	✓
▼ Req5.1	✓
M1_Landing_Sequence: DIST1 & DIST2 & DIST3	✓
▼ Req6	✓
M3_Block_Timeslots_prob_mc2: BLOCK_LTL	✓
M3_Block_Timeslots: BLOCK1 & BLOCK2 & BLOCK3 & B...	✓
▼ Req7	✓
M1_Landing_Sequence: M1_Scenario_3	✓
M3_Block_Timeslots: M3_Scenario_3 & M3_Scenario_4	✓
▼ Req7_Scenario	✓
M1_Landing_Sequence: M1_Scenario_3	✓
M3_Block_Timeslots: M3_Scenario_3 & M3_Scenario_4	✓
▼ Req8	✓
M5_AMAN_Timeout: M5_Scenario_AMAN_Timeout	✓

Fig. 2. Requirement Overview in ProB2-UI's VO manager

an 8-core Apple M1 Pro processor and 16 GB of RAM. For the experiments, we set a timeout of one hour.

As shown in Table 2, the state space rapidly grows for the first configuration. The timeline for the planes and the blocking of time slots might cause this. In contrast, model checking can be applied efficiently for the second configuration. Here, we can model-check all AMAN behaviors with the given configuration. However, as soon as the GUI events are split into multiple ones in M9 (clicking, dragging, and releasing), the state space also grows rapidly. Thus, model checking is also feasible to verify the AMAN model, but only for configurations that limit the state space. This means that model checking does not achieve full coverage like proving.

4 Validation

In the following, we report on the validation of our model using validation obligations [8,14]. A *validation obligation* (VO) consists of one or multiple validation tasks. A VO is associated with a model to check its compliance with a requirement. The validation tasks inside a VO can be connected with logical operators like $\wedge$, $\vee$, and the sequential operator ;. In such a sequential operation, the result of the first validation is used for the second validation. An example of a VO is:

$$\texttt{Req1/M1} : \texttt{MC(GOAL, somepredicate); TR}$$

This VO expresses that `Req1` is validated on the model `M1` by running model checking to find a state satisfying the given predicate and then executing a trace from the found state.

VOs allow systematic tracking of requirements during the whole modeling process and checking for conflicts between the requirements. Moreover, VOs support different development styles, which are discussed in detail in Sect. 6.2. In this section, we report using VOs in an *a posteriori* manner, i.e., the model is validated after its development.

To create and manage the VOs for the model, we used the ProB2-UI VO manager, which is partially shown in Fig. 2. In the VO manager, VOs can be created and automatically validated against the model. Colored symbols indicate if the VO is successful (green check mark), not evaluated (blue question mark), or failed (red x mark, not shown here).

We used the validation tasks related to invariants, temporal properties, scenarios, and coverage criteria for the AMAN requirements. Below are a few detailed examples of VOs we developed for our AMAN model.

Invariant: Req5. The specification states that two airplane landing times must be at least three minutes apart. Furthermore, Req5 states that the aircraft labels must never overlap. We combine this into a requirement (called Req5.1 and also visible in Fig. 2) that there is always a minimum distance between two airplanes. As described in Sect. 2, this invariant is introduced in M1 along with guards of events for AMAN_Update and Move_Aircraft. The invariant to check this behavior is shown in Eq. (1).

$$\begin{aligned}&\forall a1, a2.\ a1 \in \mathtt{dom}(\mathtt{landing\_sequence})\\ &\quad \wedge a2 \in \mathtt{dom}(\mathtt{landing\_sequence}) \wedge a1 \neq a2 \Rightarrow\\ &\quad \mathtt{DIST}(\mathtt{landing\_sequence}(a1) \mapsto \mathtt{landing\_sequence}(a2)) \geq 3\end{aligned} \tag{1}$$

where we have

$$\mathtt{DIST} = (\lambda(x \mapsto y).x \in \mathbb{Z} \wedge y \in \mathbb{Z} | max(\{y - x, x - y\})$$

Rodin's PO generator generates three POs from this invariant, which we use as validation tasks annotated as $\mathtt{DIST}_1$ through $\mathtt{DIST}_3$ in ProB2-UI's VO manager. Those POs are then combined into a validation obligation:

$$\mathtt{Req5.1/M1} : \mathtt{DIST}_1 \wedge \mathtt{DIST}_2 \wedge \mathtt{DIST}_3$$

This means that for Req5.1 to be fulfilled on M1, the validation tasks $\mathtt{DIST}_1$ through $\mathtt{DIST}_3$ must be discharged.

The final refinement M10, which introduces concrete pixel placements for all UI elements, also includes new invariants (not shown here due to size) to ensure that the UI elements' pixels indeed do not overlap. Once again, we define validation tasks from the POs generated by Rodin for these invariants and construct another VO using these validation tasks to validate Req5:

$$\begin{aligned}\mathtt{Req5/M10} : &\ \mathtt{no\_overlap\_wd} \wedge \mathtt{no\_overlap\_1} \wedge \ldots \wedge \mathtt{no\_overlap\_6}\\ &\wedge \mathtt{no\_overlap\_airplanes\_wd} \wedge \ldots \wedge \mathtt{no\_overlap\_airplanes\_6}\\ &\wedge \mathtt{no\_overlap\_block\_slots\_wd} \wedge \ldots\end{aligned}$$

Invariant: Req6. `Req6` (also see Fig. 2) states that an aircraft label cannot be moved into a blocked time slot. Blocking time slots is introduced in `M3`. We formulate an invariant, shown in Eq. (2), to validate requirement `Req6` against the model. The invariant ensures that there are no airplanes scheduled in a blocked time slot, unless the PLAN ATCo has blocked new time slots and AMAN has not yet updated the landing sequence accordingly.

$$\begin{aligned}\texttt{blockedTimesProcessed} = \texttt{TRUE} \Rightarrow \\ \texttt{ran(landing\_sequence)} \cap \texttt{blockedTime} = \emptyset\end{aligned} \tag{2}$$

Based on this invariant, five POs ($\texttt{BLOCK}_1, \ldots, \texttt{BLOCK}_5$) are generated, which are composed as validation tasks into a VO, and assigned to the requirement:

$$\texttt{Req6/M3} : \texttt{BLOCK}_1 \wedge \texttt{BLOCK}_2 \wedge \texttt{BLOCK}_3 \wedge \texttt{BLOCK}_4 \wedge \texttt{BLOCK}_5$$

However, the invariant is not strong enough to ensure `Req6` for the PLAN ATCo. Especially when `blockedTimesProcessed` is equal to `FALSE`, the invariant on its own does not ensure that the PLAN ATCo cannot move an airplane into a blocked time slot. On the modeling side, we have ensured this with the guard $\texttt{time} \notin \texttt{blockedTime}$ in `Move_Aircraft`. Thus, the case study revealed the need for a new validation task type that checks for the presence of a guard, which we had not considered previously.

We also validated other requirements using invariants. Regarding the GUI, we formulate an invariant to check the zoom level (`Req16`). Furthermore, we formulate invariants to check that the user can only interact with a maximum of one GUI element simultaneously.

Temporal Property: Req1. We also validate some requirements by temporal model checking, e.g., `Req1` (also see Fig. 2):

> Planes can [be] added to the flight sequence e.g. planes arriving in close range of the airport.

First, we tried to validate this requirement by an LTL model checking task $\texttt{LTL}_1$ (see Eq. (3)) on `M0`:

$$\begin{aligned}&\texttt{LTL}_1 := \texttt{LTL(GF(BA(\{scheduledAirplanes} \neq \texttt{scheduledAirplanes\$0\})))} \Rightarrow \\ &\texttt{GF(BA(\{}\exists x.(x \in \texttt{scheduledAirplanes} \wedge x \notin \texttt{scheduledAirplanes\$0)\})))}\end{aligned} \tag{3}$$

`BA` is a new special LTL operator in ProB which allows the usage of a before-after predicate. In this example, `scheduledAirplanes$0` and `scheduledAir-planes` denote the airplanes before and after executing an event, respectively. Thus, the LTL formula expresses that new airplanes are scheduled to the landing sequence infinitely often, under the fairness condition that the scheduled airplanes change infinitely often.

However, this does not fully cover the requirement. For example, the fairness condition excludes traces where the scheduled airplanes never change. It should be possible to add airplanes to the landing sequence, assuming it is not fully occupied. We apply CTL model checking (see Eq. (4)). $\mathtt{CTL\_Add_i}$ checks that for all paths, there is always a next state where an airplane can be added to the landing sequence if it is not fully occupied.

$$\begin{aligned}&\mathtt{CTL\_Add_i} := \mathtt{CTL}(\mathtt{AG}(\{\mathtt{card}(\mathtt{scheduledAirplanes}) = i\} \Rightarrow \\ &\mathtt{EX}\{\mathtt{card}(\mathtt{scheduledAirplanes}) > i\}))\end{aligned} \tag{4}$$

$\forall i \in \{0, \ldots, n-1\}$ where n is the maximum number of airplanes in the landing sequence. The resulting VO on `M0` is as follows:

$$\mathtt{Req1/M0} : \mathtt{LTL_1} \wedge \mathtt{CTL\_Add_0} \wedge \ldots \wedge \mathtt{CTL\_Add_{n-1}}$$

Analogously, we validated `Req2` with a CTL model check. Here, we encountered the same problem with LTL model checking.

Scenario: Req7. Scenarios are sequences of actions leading to a goal formulated in natural language. A specification often provides a set of scenarios for validation. Scenarios are also important to demonstrate behaviors to domain experts. A scenario can be represented by one or multiple traces written in the form $\mathtt{T}_1, \ldots, \mathtt{T}_k$. This means those traces are executed as tests to show that a scenario is feasible and behaves as desired. Due to space concerns, we omit the parameters of the trace replay tasks, which contain the executed events and the event parameters. For example, we consider `Req7` (also see Fig. 2):

> Moving an aircraft label might not be accepted by AMAN if it would require a speed-up of the aircraft beyond the capacity of the aircraft;

Our model does not contain aircraft capabilities. However, we can validate an abstract version of the requirement in our model. We formulate `Req7` as a scenario and validate it with traces.

1. An airplane is scheduled to land for a specific time slot.
2. PLAN ATCo moves the airplane for landing to an earlier time slot.
3. AMAN detects that the airplane cannot land at the earlier time slot, thus processes the airplane again.

We can validate the scenario by a VO on `M1` with $\mathtt{T_{m1}}$ being the trace representing the scenario:

$$\mathtt{Req7/M1} : \mathtt{T_{m1}}$$

In `M3`, we added blocked time slots as a feature. For the VO to have full coverage, it must be extended to cater to blocked slots. This is achieved by running two traces with different blocked slots configurations $\mathtt{T_{m3.1}}$ and $\mathtt{T_{m3.2}}$:

$$\mathtt{Req7/M3} : \mathtt{T_{m3.1}} \wedge \mathtt{T_{m3.2}}$$

Coverage Criterion. In the following example, we evaluate the state space coverage of multiple traces representing scenarios. A stakeholder might want to ensure the employed scenarios and the associated traces are complete. Let $\mathtt{T}_1, \ldots, \mathtt{T}_k$ be the trace replay tasks used to validate all scenarios and let COV be the coverage evaluation task. Using the ; operator to pass the state space coverage information between the validation tasks, the coverage of all scenarios can be evaluated as follows:

Table 3. Coverage Results from Scenarios

Operation	Covered
AMAN_Update	yes
Move_Aircraft	yes
Hold_Button	uncovered
Block_Time	yes
Deblock_Time	yes

$$(\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_k); \mathtt{COV}$$

Practically, we have many scenarios in M3 which are validated by the traces $\mathtt{T}_{\mathtt{m3.1}}, \ldots, \mathtt{T}_{\mathtt{m3.4}}$. A VO to evaluate the coverage can be formulated as:

$$\mathtt{Coverage/M3} : (\mathtt{T}_{\mathtt{m3.1}} \wedge \mathtt{T}_{\mathtt{m3.2}} \wedge \mathtt{T}_{\mathtt{m3.3}} \wedge \mathtt{T}_{\mathtt{m3.4}}); \mathtt{COV}$$

The result of this VO can be seen in Table 3. It becomes apparent that Hold_Button is not covered, which—after a short investigation—leads to the conclusion that this feature was never tested when introduced in M2. This makes us introduce a new VO covering this case for M2 and refining it for M3.

5 Domain-Specfic Views

We have also created domain-specific views based on the model to help domain experts and users validate the model. The core idea is that non-modelers can participate in the validation process and give feedback without needing to know the implementation details of the model.

Visualization. As the modeled system is interactive, consisting partially of a GUI, a VisB visualization can be seen as a virtual AMAN prototype. VisB [16] is a tool in ProB2-UI to create interactive visualizations for formal models. VisB visualizations consist of an SVG image and a *glue file*, which links the SVG with the formal model. In particular, the glue file defines SVG objects' dynamic appearances and click actions. Thus, a user can interact with the graphical objects by clicking on them, which triggers events in the model, changing the state according to the events' actions.

We created two VisB visualisations: a high-level version at M6 where user behavior is implicit and a lower-level version for M9 with explicit user behavior, e.g., with a visualization of the mouse cursor along with events for mouse clicking and dragging. Figure 3 shows the visualisation for M6.[5] On the left-hand side of

[5] Our visualization shows the minutes relative to the current time, while the specification document shows the current minute in the current hour on the clock. Assuming that the current time is 9:03, then our visualization displays 9:05 as 2, while Fig. 6 in [9] would display 9:05 as 5.

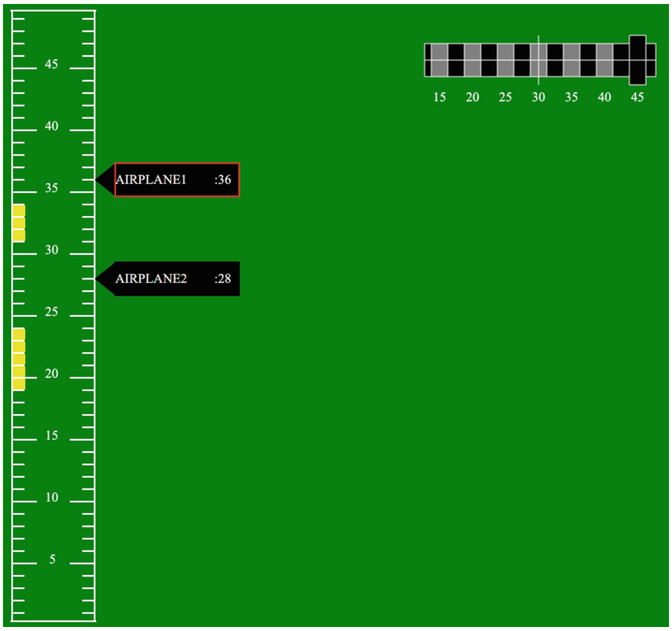


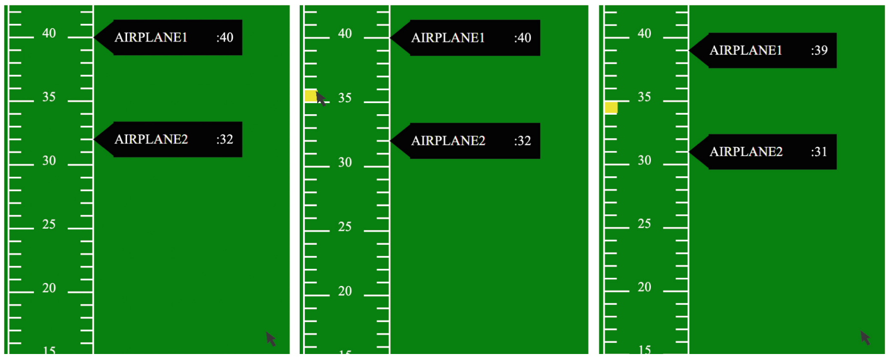
Fig. 3. Domain-Specific Visualization of AMAN in `M6`

Fig. 3, one can see the airplanes of the landing sequence and the blocked time slots (in yellow). The user can block or deblock time slots by clicking on the left-hand side. Similarly, an airplane can be selected by clicking on the label. It is then possible to hold the airplane (by clicking on the now visible `HOLD` button) or to change its landing time by clicking on the right-hand side of a time slot. Held airplanes are marked with a red frame (see `AIRPLANE1` in Fig. 3). The scale shown to the users and domain experts depends on the *zoom level*, which can be changed by clicking on the top right-hand side.

Simulation. SimB [15] aims to simulate a formal model in a realistic setting. It is a simulation tool built on top of the ProB animator, where one can associate timing and probability information with events.

Here, we combined SimB with VisB to obtain a "realistic" real-time prototype for users and domain experts to experiment with. Interactive events can be triggered by clicking within VisB, while SimB automatically executes autonomous background events. In particular, the `AMAN_Update` event is triggered every 10 s after initializing the AMAN model. AMAN updates are blocked while a user is interacting with AMAN; once the user interaction is completed, AMAN updates are activated again. An example of user interaction + simulation is shown in Fig. 4.

Abstractions. Validating some user actions is difficult due to the large state space size and the complex model we are confronted with. Therefore, we cre-



(a) *System Event:* AMAN Schedules Two Airplanes (b) *User Interaction:* Block Time Slot (c) *System Event:* Minute Passes

Fig. 4. Example: User Interaction + Simulation in SimB

ated a so-called abstraction to decrease the mental and computational load. This abstraction focuses on the user elements `M0` to `M9` without `M1`. Due to the contribution's size and content, the contribution [13] is available separately.

6 Lessons Learned

6.1 VOs for Validation

VOs provide a systematic approach to the requirements validation process. With the help of the VO manager integrated into ProB2-UI, we had a good overview of which requirements still had to be modeled, which requirements still had problems and which validations were successful (see Fig. 2). The VO manager also provided a good way to link the requirements in natural language (the "what" and possibly "why") to validate tasks that a machine can execute (the "how"). As modelers, we could focus on the how while directing questions about the what to the stakeholders.

Sometimes VOs helped us to identify conflicting requirements quickly. For example, one LTL formula introduced and validated for one requirement was later invalidated during the implementation of another requirement.

Unfortunately, much manual work is still required when dealing with VOs. While creating VOs is easy, maintaining them is hard. We are looking into improving the tool support in ProB2-UI in the future. Some VOs can already be automatically adapted for refinement, e.g., for trace refinement [12]. However, complete integration still needs to be accomplished.

6.2 VOs in Requirement Elicitation

We report our findings of employing VOs for requirements elicitation. We employed two approaches: the *a priori* approach, creating VOs before starting the modeling process, and the classical *a posteriori* approach creating the

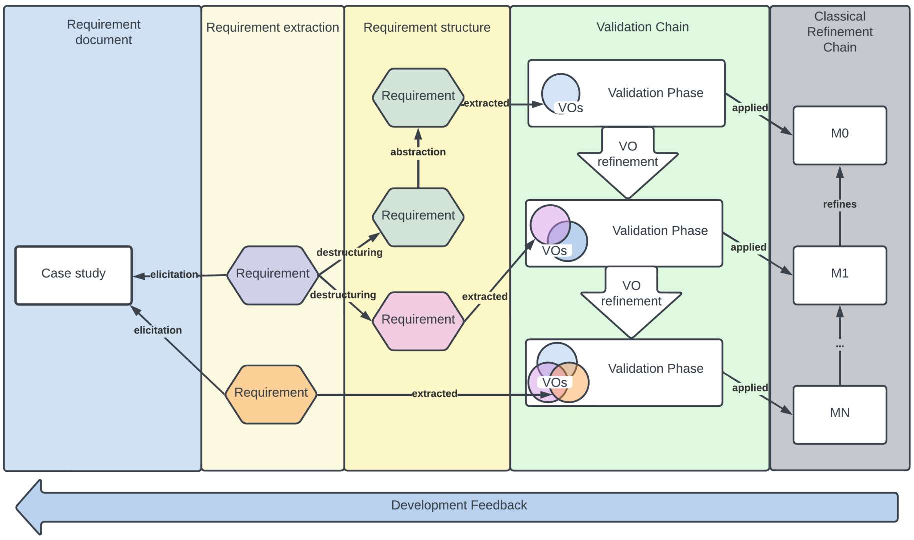


Fig. 5. A Priori Approach

VOs after or during the modeling. The resulting Event-B models were then combined and used as the baseline for the final model presented in Sect. 2. The a priori approach is new and orients itself towards test-driven and behavior-driven development schemes. We wanted to know whether such an approach is feasible for formal development.

A priori VO development. The idea of a priori development is shown in Fig. 5. The document is skimmed for the requirements and extracted and formulated as VOs. If it is impossible to write them as VO immediately, they are divided into more manageable pieces. Splitting the requirements also helps to find an initial structure in which the requirements should be implemented, as one becomes aware of the dependencies between them. After creating the VO, the model is written to satisfy the VOs. From here on, the process follows a feedback loop. When the model is refined, so are the VOs. This guarantees the presence of the requirements in the refined model.

We discovered two possible reasons for the difficulty of assigning a VO to a requirement. First, a requirement can be too complex and may consist of multiple sub-requirements, which was not obvious from the specification. In such a case, the requirement was split as shown in Fig. 5. Then, each sub-requirement was assigned a VO. For example, we wanted to implement the two requirements below into `M0`. The requirements are extracted from the explanatory text of the case study.

- $\texttt{Req}_{\texttt{Exp1}}$ An AMAN update can happen every time.
- $\texttt{Req}_{\texttt{Exp1.1}}$ Every 10 seconds, an AMAN update happens.

When creating a VO capturing this requirement, we discovered there are many assumptions behind the requirements:

1. There is a given number of updates per minute.
2. When the AMAN updates, the remaining updates are decreased by one.
3. A minute passes when the number of updates equals 0. The number of remaining updates is then reset.

Writing one VO that captures all these assumptions at once is possible but not advised as the corresponding expression would become too complex, reducing maintainability and traceability. Therefore, we decided to split the requirements and assumptions into multiple VOs. Each represents one assumption or explicit requirement. In this sense, VOs helped to structure and uncover the emerging requirements and their dependencies.

The second possible reason is when a requirement is too concrete for the current stage of the model. An example is shown in Sect. 4 when discussing scenarios. The requirement concerns concrete features of the model (e.g., the speed of aircraft), which were not implemented at this point. In such a situation, it is helpful to rephrase a requirement more abstractly, create a VO capturing the abstract requirement, and discharge it as shown in Fig. 5. Then, the corresponding VO is refined back to match the concrete version. This is useful to introduce validation for high-level requirements early on and make them part of the validation process.

A posteriori VO development. Within the a posteriori approach, a modeler first develops a model from the requirements and then validates it using VOs (see Fig. 6). Here, the modeler has to decide which requirements to choose and how they are encoded into the model for a development/refinement step. Once the development step is finished, the modeler creates VOs to fulfill the desired requirements. Furthermore, the modeler might discover new requirements, leading to a feedback loop similar to the a priori approach.

Discussion. The main advantage of the a priori approach is that the modeler has to reason about requirements and VOs in more detail before encoding them. The main disadvantage is the upfront cost of initially transforming all requirements into VOs and the time we invest in structuring them. Furthermore, the VOs cannot be checked on the model immediately.

In contrast, the main advantage of the a posteriori approach is that the VOs and the requirements can be checked against the model directly after their creation. Thus, the modeler receives feedback about errors and possible contradictions between requirements and VOs. As a result of the feedback, the modeler might also create new requirements.

Both approaches mainly differ in how they treat requirements. For example, the a priori approach focuses less on the implementation details. The a posteriori approach utilizes the experience of the modeler to avoid trial and error until a satisfying representation is found. From our current research, we argue that for a qualitative investigation of both approaches' usefulness, there needs to

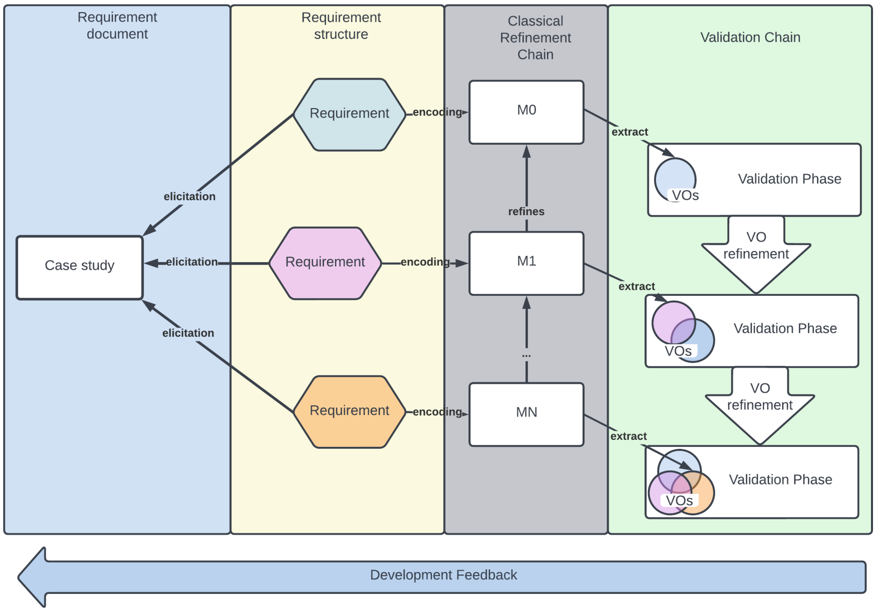


Fig. 6. A Posteriori Approach

be unified and broad tool support over multiple validation techniques. In the current unautomated state, the a priori approach required more effort than the a posteriori approach. This was due to the nature of the approach itself. Whenever the assumption about the model changed, i.e., an event was named differently, all VO that referenced this event had to be adapted, which required a lot of manual work.

6.3 VOs for Requirements Disambiguation

Requirements specifications often contain aspects that are not obvious to the modeler. In this regard, VOs uncover the unclear aspects rather quickly. Whenever we could not formulate a requirement as a VO, this triggered a deeper investigation, leading us to ask additional questions about the specification, make assumptions for the modeling process and uncover ambiguities. Below we summarise a few results of the investigations.

Questions triggered by VOs

1. For us, as non-experts, it needs to be clarified to which part of the system the term AMAN refers. Specifically: is the user-facing GUI *part of* AMAN, or is it a GUI *for* AMAN implemented as an independent component? This is relevant for `Req8`; if there are no AMAN updates for 10 s, does the GUI stop working entirely, or does it continue operating in a “manual only” mode without the autonomous part of AMAN? **Solution:** We assumed that when

a timeout occurs, the UI still functions but doesn't accept any input until the autonomous part of AMAN responds again.

2. Fig. 6 (in the specification [9]) shows an airplane on `HOLD` at 31 min, but the zoom level is at 30. However, the GUI should only show airplanes up to 30 min away. Is this an error in the example figure, or does this mean planes on `HOLD` are excluded from the zoom constraints? **Solution:** We assumed that airplanes outside the zoom are only relevant for the landing sequence but for nothing else. We later discovered that Fig. 6 in the specification displays the current minute within the current hour on the clock rather than the minutes relative to the current time.
3. It needs to be clarified what happens to airplanes after they are put on HOLD. Are they moved into a separate "HOLD sequence" and still shown to the ATCo? Alternatively, do they disappear entirely from the AMAN GUI? **Solution:** We assumed an airplane should stay indefinitely until it is explicitly removed from the landing sequence. Furthermore, a hold airplane might be rescheduled for a later time slot while not put off hold.
4. Based on Fig. 6 (in the specification), airplanes on HOLD still have an expected landing time. Does the 3-minute separation between landing times also apply to HOLD airplanes? **Solution:** We assumed this is the case.
5. When the user pushes and holds a button and a minute passes, what happens to the planes in the landing sequence? Could we enter an infinite loop where the AMAN never updates again (i.e., when we push and hold the left mouse button in a valid position)? **Solution:** We assumed that the user interaction does not take forever.

Assumptions. Furthermore, we made additional assumptions to model the AMAN in Event-B:

1. If the ATCo selects an airplane and zooms in so the airplane is no longer visible, is it still possible to press the hold button? We assumed this is *not* possible, as it would contradict `Req12`. Moving the zoom slider should deselect an airplane that leaves the zoom range, making this situation impossible.
2. Initially, we assumed that Fig. 6 in the case study specification shows the minutes relative to the current time. As a result, we lacked explanations about what happens with blocked time slots when time passes. Consequently, we model the time relative to the current time (also implemented in the VisB visualization, see Fig. 3). Thus, we assumed that once a minute passes, all blocked time slots are moved forward one minute.
3. We did not model the landing of airplanes, especially because the specification does not go into detail about this. Instead, we assume that landed planes are removed from the landing sequence, just like planes that disappear from the landing sequence for any other reason.
4. We assume that when the zoom slider is moved, the new zoom level is only applied once the mouse button is released.
5. We assume that there is no possibility to unhold a plane.

Ambiguities uncovered by VOs. Finally, we reported back to the case study providers on discovered ambiguities that were eventually resolved in the updated requirements specification due to our feedback:

1. We first assumed that the AMAN overrules the user. However, the user can overrule the AMAN according to the updated requirements specifications of AMAN. This means that user interaction has a higher priority than AMAN.
2. Inconsistency in requirements, e.g., landing sequence and arrival sequence, led us in the wrong direction by attempting to model airplanes approaching the airport separately from the landing sequence. This inconsistency was also removed in the updated requirements specification of AMAN.

6.4 Role of Verification

During the development of the AMAN model, It was better to verify the model before validating it (in each development step). The validation techniques quickly detected problems when changing the model, e.g., an LTL formula or a trace may no longer be valid. Using Event-B in Rodin, we receive fast feedback about whether a PO is discharged. In general, however, there might be properties that are difficult to prove (e.g., they might require finding inductive invariants). In those cases, it is probably best to interleave verification and validation and only tackle the proof of complex properties once validation is successful.

7 Conclusion and Future Work

In this work, we presented a formal AMAN model in Event-B, developed using Rodin. This case study is challenging from the modeling perspective as it combines an interactive part, including a GUI, with an autonomous part. In particular, the AMAN case study highlighted the importance of stable and flexible tools that can deal with changes in the model and encourage experimentation.

For verification, we noted that POs are well functioning and valuable. However, with the introduction of complex GUI behavior, discharging them became increasingly challenging. Model checking proved unsuitable as a fallback option for verifying the complete model, as it struggles with the state space explosion problem. However, it is usable with decent performance when instantiating the model with restrictions.

During the validation of AMAN, we experienced that VOs are particularly useful in structuring the validation process and linking validations and requirements. Here, we critically analyze two development approaches and the ambiguities we uncovered during the employed modeling process.

Furthermore, we often felt the need to show our model to a domain expert and ask for feedback, which means that the domain expert's feedback is a valuable source of information and should be treated as such. To tackle this, we created an interactive GUI in ProB via VisB together with a simulation of autonomous AMAN activities with SimB.

In conclusion, AMAN is an interesting case study for further investigations, especially since the interactive part was fruitful in giving inspiration for developing and improving new techniques.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., 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. Transfer **12**(6), 447–466 (2010)
3. Aït-Ameur, Y., Aït-Sadoune, I., Baron, M., Mota, J.M.: Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes Multi-Modaux. Journal d'Interaction Personne-Système **1** (2014). https://doi.org/10.46298/jips.59, https://jips.episciences.org/59
4. Bendisposto, J., et al.: ProB2-UI: a java-based user interface for ProB. In: Lluch Lafuente, A., Mavridou, A. (eds.) FMICS 2021. LNCS, vol. 12863, pp. 193–201. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85248-1_12
5. Borrione, D., Paul, W. (eds.): LNCS, vol. 3725. Springer, Heidelberg (2005). https://doi.org/10.1007/11560548
6. Leuschel, M.: Operation caching and state compression for model checking of high-level models - how to have your cake and eat it. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods, Proceedings IFM 2022, LNCS, vol. 13274, pp. 129–145 (2022). https://doi.org/10.1007/978-3-031-07727-2_8
7. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
8. Mashkoor, A., Leuschel, M., Egyed, A.: Validation obligations: a novel approach to check compliance between requirements and their formal specification. In: ICSE'21 NIER, pp. 1–5 (2021)
9. Palanque, P., Campos, J.C.: AMAN case study. https://drive.google.com/file/d/1IqftxQIvrWpX1lcRts3WJzrBH7a3dMln/view
10. Rehm, J., Cansell, D.: Proved development of the real-time properties of the IEEE 1394 root contention protocol with the Event B method. In: Proceedings ISoLA, pp. 179–190 (2007)
11. Singh, N.K., Aït-Ameur, Y., Geniet, R., Méry, D., Palanque, P.: On the benefits of using MVC pattern for structuring Event-B models of WIMP interactive applications. Interact. Comput. **33**(1), 92–114 (2021). https://doi.org/10.1093/iwcomp/iwab016
12. Stock, S., Mashkoor, A., Leuschel, M., Egyed, A.: Trace refinement in B and Event-B. In: Riesco, A., Zhang, M. (eds.) Formal Methods and Software Engineering, Proceedings ICFEM, LNCS, vol. 13478, pp. 316–333. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17244-1_19
13. Stock, S., Vu, F., Geleßus, D., Leuschel, M., Mashkoor, A., Egyed, A.: Validation by abstraction and refinement. In: Proceedings ABZ (2023)
14. Stock, S., Vu, F., Mashkoor, A., Leuschel, M., Egyed, A.: IVOIRE Deliverable 1.1: Classification of existing VOs & tools and Formalization of VOs semantics. CoRR abs/2205.06138 (2022). https://doi.org/10.48550/arXiv.2205.06138
15. Vu, F., Leuschel, M., Mashkoor, A.: Validation of formal models by timed probabilistic simulation. In: Raschke, A., Méry, D. (eds.) ABZ 2021. LNCS, vol. 12709, pp. 81–96. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77543-8_6
16. Werth, M., Leuschel, M.: VisB: a lightweight tool to visualize formal models with SVG graphics. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 260–265. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_21