










Validation by Abstraction and Refinement

Sebastian Stock¹  , Fabian Vu² , David Gelebus² , Michael Leuschel² ,
Atif Mashkoor¹ , and Alexander Egyed¹ 

¹ Institute for Software Systems Engineering, Johannes Kepler University Linz,
Altenbergerstr. 69, Linz 4040, Austria

{sebastian.stock,atif.mashkoor,alexander.egyed}@jku.at

² Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, 40225
Düsseldorf, Germany

{fabian.vu,dagel101,leuschel}@uni-duesseldorf.de

Abstract. While refinement can help structure the modeling and proving process, it also forces the modeler to introduce features in a particular order. This means that features deeper in the refinement chain cannot be validated in isolation, making some reasoning unnecessarily intricate. In this paper, we present the **AVoiR** (**A**bstraction-**V**alidation **O**bligation-**R**efinement) framework to ease validation of such complex refinement chains. The triptych **AVoiR** framework operates as follows: 1) We first simplify a complex model by abstracting away the *noise*, i.e., removing the information unrelated to properties under analysis. 2) Using the Validation Obligations (VOs) technique, we formalize the validation tasks of the desired property. 3) Finally, we trickle down the validation results by establishing the *noiseless* model as a parent of the initially investigated model through the standard refinement relationship. Furthermore, by using the technique of VO refinement, we establish the VOs of the abstract model on the initial model. We use a case study from the aviation domain to show the proposed framework’s effectiveness.

Keywords: Formal Methods · Validation Obligations · Abstraction · Refinement · Validation · Event-B

1 Introduction

Model verification [18] checks whether we are building the model right. It often takes center stage in state-based formal methods [22], and there is a large set of robust verification techniques (see, e.g., the survey of tools for verification

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 and Alexander Egyed has been partly funded by the LIT Secure and Correct Systems Lab sponsored by the province of Upper Austria.

by Punnoose et al. [25]). In contrast, model validation [18], i.e., do we build the right model, aims to ensure that the model does what stakeholders want. Validation requires a good understanding of the property under investigation and how the model represents it. An additional challenge is that a model can be vast and complex, and not every model property is equally interesting for every stakeholder. So, suppose a stakeholder wants to validate a single property of a complex model. In that case, the interactions of the property with other model elements render this goal challenging as *noise* is coming from unrelated properties. Unfortunately, the existing state-of-the-art techniques and tools for model validation offer little help in this regard.

Consider the AMAN case study [24] about an airplane scheduling system consisting of several refinement steps¹². The behavior of the automatic/mechanical part is modeled early (M0 and M1), while the manual/user behavior part is modeled later (M2 to M9). If we want to validate the user behavior without the interference of the mechanical part, we are out of luck and have to deal with the *noise* from M0 and M1. It would be beneficial if we could abstract away the properties producing *noise*, enabling validation of the user behavior of M2 to M9 without unnecessary details.

This paper proposes the triptych **AVoiR** (**A**bstraction **V**alidation- **O**bligation **R**efinement) framework to validate a property of interest in a formal model by reducing any *noise*. In the first step of the framework, one abstracts away parts producing *noise*, making the model easier to validate. The second step establishes whether a property of interest is valid on the abstraction using Validation Obligations (VOs) [23]. In the third step, one establishes the created abstraction as an additional parent of the initially investigated model and transfers the VOs established on the abstraction back to the initial model using the refinement relationship. Using the AMAN case study from the aviation domain, we showcase the efficacy of the **AVoiR** framework.

The rest of the paper is structured as follows: Sect. 2 introduces the Event-B method, which we use in the context of abstractions and as a carrier language to provide an illustrative example and the notion of VOs. In Sect. 3, we give an overview of the **AVoiR** framework and introduce abstractions for Event-B and VO refinement. We then demonstrate the usability of the **AVoiR** framework in Sect. 4 on the AMAN case study and show a complex property on the abstraction, formalize it as a VO, and transfer it back to the initial model. Last, we compare the proposed framework with related work in Sect. 5 and conclude the paper in Sect. 6.

2 Background

2.1 Event-B

Event-B [1] is a state-based formal method with refinement as a key mechanism. A modeler can create a so-called **machine**, which describes a state automaton.

¹ Original case study code: <https://github.com/hhu-stups/AMAN-case-study/>.

² Code for this paper: <https://github.com/hhu-stups/AMAN-abstraction-example>.

The state is represented by **variables**, defined and checked against **invariants**. State transitions are defined through **events**. Additionally, **contexts** define new data types that machines can see.

Refinement is an established technique for model enrichment. Refinement means step-wise, rigorous, and inductive enhancement until a satisfying level of detail is reached. However, there is a wide variety of methods implementing different styles of refinement. In Event-B, a refinement is established by conducting an inductive proof that the refining **machine** does not violate existing constraints. The goal of the refinement is to either add a property or bring the model closer to implementation. In general, for the rest of the paper, we specify two kinds of refinements: *vertical refinement* and *horizontal refinement* (see Yeganeh et al. [30] for more details). Vertical refinement is about the refinement of variables, i.e., abstract variables are replaced by more concrete ones. They are usually linked by so-called *gluing invariants* for proving purposes. In contrast, horizontal refinement means adding new behavior to the model. The Rodin platform [2] supports refining and proving models.

Abstraction (in the context of this work) can be seen as the opposite of refinement. In this technique, we take (abstract) away unnecessary details from a model in a controlled manner leaving behind only the properties of interest. The resulting model is crisp and *noiseless*.

We introduce abstractions as a part of the AVoIR framework tailored to Event-B. However, there are other state-based formal methods like ASM [9] or TLA+ [20], where the framework may also be applied.

2.2 Validation Obligations

VOs were introduced by Mashkoor et al. [23] and further defined by Stock et al. [28]. They aim to provide a systematic embedding for requirements assuring conflict freeness and completeness. We provide a quick recap of the notion of VOs to facilitate readers.

A validation obligation (VO) is a validation expression (VE) composed of (multiple) validation tasks (VTs) associated with a model to check its compliance with the requirement.

We can express a VO formally by:

$$\text{Req/Model} : \text{VE}$$

The VE consists of one or VTs combined using logical operators \wedge , \vee , and a special sequencing operator $;$. $A;B$ means that the end state of task A is used as the starting state for task B . Figure 1a shows the VO structure schematically. A *requirement* is realized in the model and ensured to be present by a VO. The VO contains the VE with the necessary parameters. A parameter requires the following three considerations: the VT the parameter is put into (e.g., LTL model checking needs an LTL formula), the properties the VT attempts to validate (e.g., a liveness property is validated with an LTL formula), and the implementation

chosen in the model (e.g., the names of variables and events). To talk about a single validation task, we use the following naming pattern: $VT(\text{parameters})$

VT is a placeholder for a specific task type. parameters are the parameters of the employed validation technique. An example of a task would be TR which is the trace replay task that executes an animation from a given point. The parameter for this task would be a trace. Another task example would be model checking MC . The parameter for the MC task would be calculating the coverage (COV), checking for invariant violation (INV), or searching for a goal ($GOAL$), i.e., a predicate to be satisfied. Multiple parameters must be provided depending on the specialization, i.e., one needs a predicate to be satisfied for the $GOAL$ specialization. An example of a VE that searches for a state and executes an animation from the found state on a fictive machine $M1$ can be written as:

$$MC(GOAL, \text{some_predicate}); TR(\text{some_trace})$$

3 AVoiR Framework

An overview of the $AVoiR$ framework is shown in Fig. 1b. The three steps of the framework are: 1) create an abstraction, 2) use VOs to formalize and validate properties of interest, and 3) establish the abstraction as an additional parent to the initial model and refine the VO to fit the initial model. In the following, we describe each step in detail.

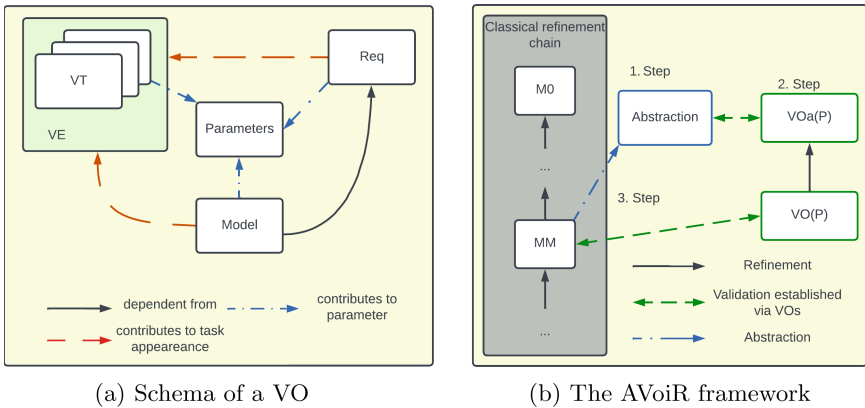


Fig. 1. Schematic view on a VO (left) and the $AVoiR$ framework (right)

3.1 Step 1 - Creating an Abstraction

The first step of the $AVoiR$ framework is to create an abstraction, reducing the *noise* and enabling a more accessible investigation of the model's properties. In

the context of Event-B, an abstraction is a recomposition of selected features already present in the refinement chain. For the transferability of findings, an abstraction acts like an additional parent of an existing machine without altering the refinement chain.

Consider Fig. 2a. There, we see a classic refinement chain $M0$ to $M2$, with $M0$ being the most abstract machine and $M2$ being the most concrete one. We can now create an abstraction from this refinement chain by selecting features (variables or events) we want to observe and creating a new machine from these features. In Fig. 2a, the features of $M0$ and $M2$ are used for the abstraction, and those features are recomposed in an abstraction $A1$. The red arrow between the two feature extraction arrows indicates that we can have side effects on variables and events. Indeed, $M1$ could do a vertical refinement (data refinement) on variables of $M0$. $M2$ relies on these refined variables and is incompatible with variables from $M0$. In this case, we need to *demote* the variables from $M2$ relying on $M1$ to instead rely on the variables of $M0$.

Example. Let us consider Abrial’s interlocking model [1]. The model aims to ensure collision freedom in a train yard. For demonstration purposes, we consider the refinement levels `train_0` to `train_4`³ from the abstract to the most concrete machine. `train_0` models routes over the tracks as a set of blocks that can be reserved. The variable `resrt` is vital for us, representing all reserved blocks. `train_1` builds a data structure that maps blocks to a tracking number. The variable `frm` is important because it represents all formed routes. The other refinements add more details to model trains and signals.

Let us consider a situation where we ask a railway domain expert to validate our assumptions made in the model. We especially want to know whether the reservation, forming, and freeing of routes are in the proper order. However, for the modeling, we choose an abstract representation of these three statuses for a route, which is difficult to comprehend for a non-specialist, who would need to learn the syntax. For modelers, the free routes would be $ROUTES \setminus (\text{resrt} \cup \text{frm})$, reserved routes would now be $\text{resrt} \setminus \text{frm}$, and formed routes would now be `frm`.

As this feature interplays with other features, it could be hard for a non-modeler to understand and give feedback. Therefore, we reduce the *noise* from this formulation by creating an abstraction $A1$ as shown in Fig. 2b. We *demote* the high-level constructs of `frm` and `resrt` to a simple representation we call `rs` (route status). $A1$ only contains `rs` and events that manipulate the route status, with the events adapted to the *demoted* variables. With the created abstraction, we can now do all sorts of validation, e.g., animation, tracing, and state space projections.

³ The whole example is available under https://figshare.com/articles/code/Abstraction_Examples/19786924/3.

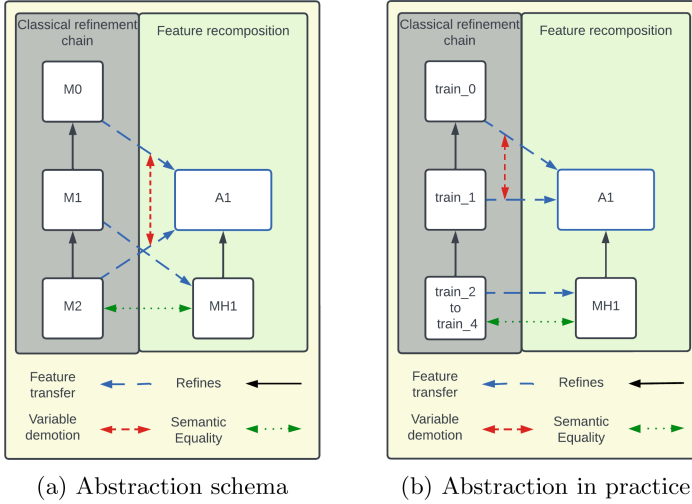


Fig. 2. Schematic abstraction (left) and abstraction from the example (right)

3.2 Step 2 - Creating VOs

With the abstraction in place and the domain experts' feedback, we proceed to the second step of the AVoIR framework to systematically validate properties under investigation in the abstract model. For this, we employ the notion of VOs as introduced in Sect. 2.2. An example requirement to be formulated as a VO would be REQ0: Reserving, forming, and freeing a route is possible in this particular order. A VO stating this would have the form:

REQ0/A1 : TR([route_reservation, route_formation, route_freeing])

After its creation, the VO can be successfully validated against the model to establish the property's presence.

3.3 Step 3 - Trickling Down Insights

Many techniques that transfer validation-sensitive results between an abstract and a concrete model rely on a formal refinement relationship established between both. An example is LTL refinement as presented by Schneider et al. [26]. For this reason, it is useful to establish the abstraction as an additional parent of the initial machine. Consider Fig. 2a, where we want to establish a refinement relationship between A1 and M2 to transfer insights. However, since M2 is already refining M1, it cannot have another parent as per Event-B laws. We, therefore, create a helper machine MH1, which contains all the missing features from M1 and refines A1. It might become necessary to create new gluing invariants to deal with the *demoted* variables. In the end, if MH1 is equal (same variables,

same events, same invariants, ...) to M2 (minus the added gluing invariants), we know that A1 is a parent of M2.

VO Refinement. VOs refinement now complements the abstraction by enabling the systematic transfer of validation results along a refinement chain. A VO can consist of multiple tasks, and to refine them, we need to know how they interfere with each other.

Together with the definition of a VO given in Sect. 2.2, the VO refinement is defined as follows:

A validation obligation (VO) which is established on an abstract model, is refined for the concrete model by applying the means of refinement to the parameter(s) of the included validation tasks (VTs).

The VTs are included in the VE of the VO, and each of their parameters needs to be refined. At first glance, it might seem more intuitive to refine the tasks. However, attempting this is challenging as we need to show the semantic equivalence of the two tasks. So instead, we focus on the parameters to preserve the encoded meaning, as already existing techniques show. We introduce the concept of the 'mean of refinement' to discuss the refinement of parameters. It will help us discuss what happens to a parameter during refinement.

The mean of refinement is the connection of abstract and concrete models in horizontal or vertical refinement. In the case of Event-B's vertical refinement, the mean of refinement is the gluing invariant, as this is the construct to connect both machines. We can apply this gluing invariant to transform an abstract variable into a concrete one. With the horizontal refinement, the mean of refinement would be the delta of abstract and concrete variables and events, i.e., which event is renamed or split into which other event(s) and which variables were added. For example, when splitting an abstract event into multiple concrete ones, the occurrence of the abstract event in a VT parameter can be replaced with a disjunction of all its concrete versions.

For each VT type, the refinement process is different as it must cater to the needs of the parameter and the means of refinement. For example, a trace is a parameter for the TR VT. Following our rule, we need to refine the trace, adapting it to the concrete machine or, in case of an abstraction to the initial machine. However, to achieve this, we must first detect the means of refinement, i.e., what changed between the abstract and concrete models. This process can be automated for trace replay, as shown by Stock et al. [27]. The final example is the LTL model checking VT LTL. For this VT, we need an algorithm to translate LTL formulas as, for example, laid out by Hoang et al. [16]. The translated formula would then have to be re-checked against the concrete version of the model. An alternative consists of proving the preservation of the property described also laid out by Hoang et al. [16] and later by Zhu et al. [31]. The impacting factor for VO refinement is the grade of available automation. For traces, the automation grade is high. For LTL, a semantic translation exists, laid out by

Hoang et al. An alternative would be to use proof obligations by Zhu et al. [31]; however, these are not automated yet. In the case of a visual state diagram, we would have to manually re-check as there is no refinement procedure for it yet.

VO refinement can be unsuccessful, i.e., the re-execution of the task fails. If this is the case in a regular refinement, we eradicated existing behavior with our refinement. If this is the case in an abstraction relationship, the abstraction over-approximated the reality of the initial model. When we try to re-validate the approximated property, it collides with the initial model, and the VO fails. From a failing VO, we can conclude that we chose the wrong abstract representation or our requirement might not be valid as the model might not satisfy it in general.

Refinement Syntax. VOs consist of validation expressions which again are composed of multiple tasks. Thus, we have to define how to refine these expressions. Therefore, we denote the refinement of `Model` with `Model'` and the refinement of a VO with:

$$\text{refine}(\text{Req/Model} : \text{VE}) = \text{Req/Model}' : \text{refine}(\text{VE})$$

The refinement of the expression is then achieved by refining the composition of the tasks making up the expression.

$$\begin{aligned} \text{refine}(A \vee B) &= \text{refine}(A) \vee \text{refine}(B) \\ \text{refine}(A \wedge B) &= \text{refine}(A) \wedge \text{refine}(B) \end{aligned}$$

And on the lowest level, a task is then refined with:

$$\text{refine}(\text{VT}(\text{parameters})) = \text{VT}(\text{refine}(\text{parameters})) \quad (1)$$

Refining the Sequential Operator. The refinement of the sequential operator is as follows:

$$\text{refine}(A;B) = \text{refine}(A); \text{refine}(B)$$

Refining the sequential operator has multiple side effects as the notion of state is involved. Figure 3 shows a trace representing the same VO on the abstract and the concrete model. The graphic has two main parts: the left-hand side represents the prefix (task A), and the right-hand side represents the sequential operation suffix (task B).

Consider a VE defined as follows: $\text{MC}(\text{GOAL}, \text{somepredicate}); \text{TR}(\text{trace})$ (with $\text{MC}(\text{GOAL}, \text{somepredicate}) = A$ and $\text{TR}(\text{trace}) = B$). Part A intends to reach a specific state, and part B executes a specific trace in the second step. Assuming this composed task holds in the abstract top part of Fig. 3, we refine the VE for the concrete model by applying the rules previously introduced. For the sequential operator, four cases might arise in Fig. 3:

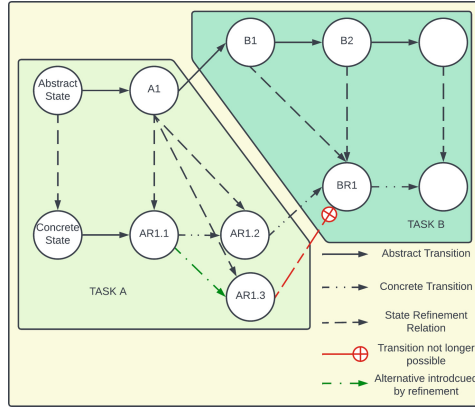


Fig. 3. Possible behaviors of states when refining the sequential operator

- Case 1: We refine and execute sub-task A and end up in AR1.2, from which we can execute task B.
- Case 2: We refine and execute sub-task A and end up in AR1.3 and assume it is the only refinement for A1. From this state, we cannot execute task B.
- Case 3: We refine and execute sub-task A and end up in AR1.3, from which we cannot execute task B. However, there might be other solutions where task B is feasible.
- Case 4: We refine and execute sub-task A and end up in state AR1.1. We would have to do an additional step to reach task B.

Case 1 is trivial as we can proceed with refining task B. Case 2 is also straightforward; the VO cannot be refined in this scenario. Case 3 requires us to search for other solutions. This can be challenging as we might not know whether other solutions exist or how to find them. However, this is a tool and modeling problem. Case 4 is more complicated. We successfully found a refinement for task A but need to reach task B. Therefore, we must pass through an additional state (AR1.2). State AR1.2 may be introduced by horizontal refinement, i.e., a new concrete behavior was introduced that forced state A1 into (two) different concrete sub-states. The challenge here is to recognize that these states are part of the same abstract parent and belong together. Suppose we can reliably recognize two concrete states that belong to the same abstract state. In that case, Case 4 poses no threat. We can refine each task individually concerning the sequential operator, and the task can be re-executed successfully. So, we can assume that the property from the abstract model is successfully transferred to the concrete model.

There is also a practical implication for Case 4. For example, we may want task B to have AR1.1 as a starting point instead of AR1.2. This is a valid demand, as both states represent the same abstract state but different concrete ones, and we may prefer the concrete state of AR1.1 over the one of AR1.2. There remains

the challenge of recognizing when a concrete state belongs to the same abstract one to select the right one. The best action in such a situation is to sharpen the VO. Sharpening the VO means creating a new VO on the concrete model. This VO has the same abstract behavior and explicitly rules out/demands concrete behavior we want/do not want to see. For instance, in our example, we would create a new validation expression and modify task A so that the goal rules out state AR1.2 while keeping task B intact. Of course, it might be the case that there is no solution.

Implications for Requirements. Until now, we only dealt with changes in the model. However, requirements might also change. A changing requirement will result in a changing model, task, and parameter (see Fig. 1a as a reference point). Tasks may become inappropriate for showing the presence of the requirement as a result of changing requirements. In this case, we must create a new VO to ensure the changed requirement's presence.

Example Continued. Now, we transfer the gained insights back to the initial model. For this, we refine A1 to MH1. In MH1, the features of `train_2`, `train_3`, and `train_4` are introduced. Further, we must refine the previously *demoted* variables. For this, we create additional gluing invariants:

$$rs^{-1}[\text{free}] = \text{ROUTES} \setminus (\text{resrt} \cup \text{frm}) \quad (2)$$

$$rs^{-1}[\text{reserved}] = \text{resrt} \setminus \text{frm} \quad (3)$$

$$rs^{-1}[\text{formed}] = \text{frm} \quad (4)$$

Equation (2) describes a free route as a route that is neither formed nor reserved. Equation (3) describes a reserved route as the reserved blocks minus the formed ones. Equation (4) describes the formed route as equal to the formed blocks. MH1 should now have the same content (events, variables, guards, invariants,...) as `train_4`, plus the added gluing invariants. We can therefore be sure that A1 is, so to speak, an additional parent of `train_4`, which allows us to transfer validation results like traces from the abstraction to `train_4`. Now, we also transfer the trace. $\text{refine}(\text{REQ0}/\text{A1}) = \text{REQ0}/\text{MH1}$. Refinement means the changed events and mapping Eqs. (2) to (4). However, as previously mentioned, tool support lets us successfully re-establish the trace for M4. The refined VO is of the form:

REQ0/MH1 : TR([route_reservation, point_positioning, route_formation,
FRONT_MOVE_1, FRONT_MOVE_2, BACK_MOVE_2, FRONT_MOVE_1, route_freeing])

With `FRONT_MOVE_1` and `BACK_MOVE_2` being the movement of the train and `point_positioning` the movement of switches. We could, therefore, transfer the previously gained insight back to the initial model.

Regarding proof obligations, the abstraction will create its own set of POs, many already encountered in the refinement chain M0 to M2. Moreover, additional POs will prove the relationship by gluing invariance between MH1 and A1.

Correctness. We can assure that the abstraction is an additional parent by discharging all POs. The correctness of the trickled-down validation results is completely up to the used techniques and tools. Therefore, correctly using and respecting their application conditions ensures the correctness of the transfer.

4 Case Study

To demonstrate the efficacy of the AVoiR framework, we apply it to the AMAN case study [24]. The case study focuses on modeling an Arrival Manager (AMAN). This semi-automatic, interactive system manages planes arriving at an airport by assigning them a landing timeslot, i.e., creating a landing order for the arriving planes. AMAN consists of two parts: a mechanical system that schedules the planes and a GUI from which a human can intervene, block timeslots for planes, and move planes around.

To evaluate the AVoiR framework, we use the implementation shown in Sect. 1. The model consists of nine refinement steps with M0 the abstract and M9 the concrete machine. The original implementation is described in detail in [15].

- M0 models an abstract set of planes (`scheduledAirplanes`) that the AMAN can manipulate.
- M1 replaces `scheduledAirplanes` with `landing_sequence` mapping planes to time slots.
- M2 adds the function for the human operator to set airplanes on ‘hold’.
- M3 adds the human operator’s function to block timeslots so that no plane can be scheduled there.
- M4 adds the function for the human operator to use a zoom that restricts the period currently worked on.
- M5 models the behavior when the mechanical part of the AMAN has a problem, i.e., a timeout.
- M6 models the user’s ability to select an airplane.
- M7 models the user’s ability to drag an airplane.
- M8 models the user’s ability to drag the zoom slider.
- M9 models the behavior of the user’s mouse cursor.

For demonstration, we create a *noiseless* view of the user behavior via an abstraction based on M9. Furthermore, we validate user behavior in a way especially tailored toward non-modeler domain experts on this abstraction via VOs. Finally, we transfer insights we gathered on the abstraction back to M9 via a VO refinement.

4.1 Abstraction

To create a *noiseless* version that only focuses on user interaction, we select all features from M0 and M2 to M9. We exclude the discrete representation of time and the explicit `landing_sequence`. As many variables introduced in M2 to M9 rely on time, we need to *demote* these variables to work without time; the events remain mostly untouched. Consequently, we get an abstraction MAbs.

Since abstraction removes details from the model, the state space is often reduced. Therefore, we can apply validation techniques relying on explicit-state model checking more easily. Table 1 shows the model checking times of both M9 and MAbs via ProB [21]. We used an Intel Core i7-10700 2.90GHz \times 8 CPU with 16GB RAM running Linux Mint for model-checking. We set a timeout of 10 min and use the same configurations for model checking. Furthermore, for both versions, we used the same amount of variable elements of the model, i.e., how many planes fly around and how far can be zoomed⁴. The experiment was repeated ten times, and the mean of the measured time and memory consumption was taken. For M9, the model checking process stopped unfinished after 10 min. In the unrestricted version of the experiment, we ran out of memory for M9; due to the computer crashing, no data was collected.

Table 1. Model Checking Results

Machine	Completion	States	Transitions	Time [s]	Memory [MB]
M9	Incomplete	> 8145	> 10 285 196	> 600	5565 ^a
MAbs	Complete	15 361	203 778	6	241.5

^a Memory usage at crash

4.2 Validating the Behavior

On the abstraction, we now validate a domain-specific requirement REQ1: When a click has been made and is ongoing, the only way to click something else is to release the click first.

Validating this with techniques like LTL model checking on M9 can be challenging due to finding the appropriate representation, having an acceptable runtime, and collecting the domain experts' feedback. However, it becomes simple with an abstraction focusing on the UI behavior. The solution is to create a state-space projection [19] that shows the click behavior and validates via projection inspection. A state space projection can be seen as a lens through which we look at the state space of a model. A state space projection needs a fully explored state space to work. We formalize REQ1 as a VO:

$$\text{REQ1/MAbs} : \text{MC}(\text{COV}); \text{VIS}(\text{PRJ}(\text{clickStartPosition}))$$

⁴ For full details, we refer to the files.

This VO states that we first model check the abstraction for full state space coverage and then apply a visualization task (VIS) to the uncovered state space to create the possibility of optical investigation. For this visualization, we use the state space projection (PRJ) with a formula on the uncovered state space. Our formula consists of one variable, `clickStartPosition`, that contains the GUI element on which the mouse click started. ProB creates a diagram, which we show in Fig. 4, consisting of five distinguishable entities represented by a box in the figure. Indeed, there are five different values. `clickStartPosition` is empty when the mouse clicks outside any GUI element. `clickStartPosition` contains `zoom_slider_pos` when the mouse was pressed on the zoom slider, `hold_button_pos` when it was pressed on the hold button, `airplane_pos` when it was pressed on an airplane, and `block_time_pos` when it was pressed on a time slot. Boxes represent the states, while arrows indicate state transitions in the form of events. So, for example, we cannot drag an airplane with the mouse and simultaneously change the zoom. Instead, we see that when something is clicked, we need to deselect it before we select something new, i.e., from a state where we choose something we cannot transition into another state before deselecting.

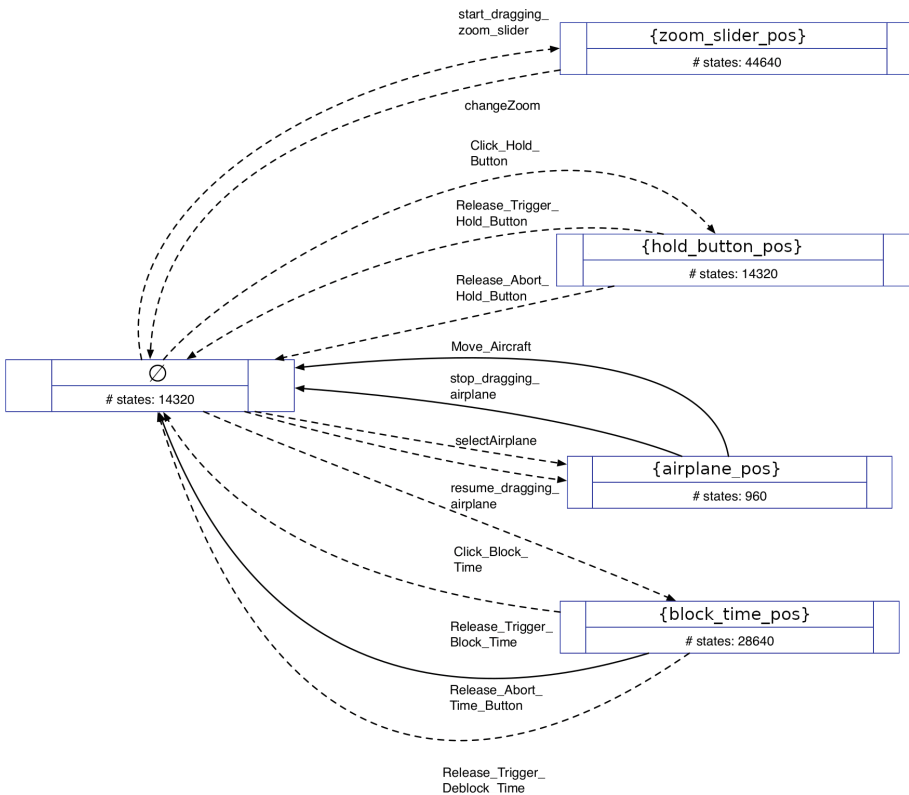


Fig. 4. State Space Projection, projecting on possible values for the mouse

Figure 4 is suitable for communication with a domain expert as it is relatively simple to understand. From the state space projection, we derive additional observations we want to hold on $M9$. For example, consider $REQ2$: When clicking on a time block happens, there are three ways to stop clicking at the block, as shown in Fig. 4. Namely, ($REQ2.1$) `Release_Trigger_Block_Time` (release the mouse on the block), ($REQ2.2$) `Release_Abort_Time_Button` (drag the mouse away and release elsewhere), ($REQ2.3$) `Release_Trigger_Deblock_Time` (a click on it deblocks a time slot that has been blocked). We create three traces that cover the desired events to validate this. We make a prefix for all three traces that execute `Click_Block_Time` and then three suffixes, one for each release action.

Let us formalize our intent as a VO. First, we create a VT that represents the prefix, which is reusable for all three traces, and the suffix that covers case ($REQ2.1$) with the other cases analogously. The VTs can then be assembled to a VE and assigned to the requirement. The following example covers the requirement's instance ($REQ2.1$).

```

pref := TR([SETUP_CONSTANTS,
            INITIALISATION, Move_Mouse_Block, Click_Block_Time])
suf1 := TR([Click_Block_Time,
            Move_Mouse_Nothing, Release_Abort_Time_Button])

REQ2.1/MAbs : pref; suf1

```

The VO can operate successfully at the abstraction, which establishes the requirement.

4.3 Refining VOs

For further development, it is helpful to re-establish $REQ2.1$ on $M9$ to know whether the properties of `MAbs` will hold and to bring it in line with existing VOs to ensure conflict freedom between them. To achieve this, we will refine the abstraction into a machine `MAbs_Helper`, which re-introduces the time property that was previously removed in the abstraction. We then apply $\text{refine}(REQ2.1/MAbs) = REQ2.1/MAbs_Helper$. Finally, we compare `MAbs_Helper` with $M9$. If both machines are equivalent regarding their events, invariants, and variables, we consider them equivalent.

During the creation of `MAbs_Helper`, some proof obligations must be manually discharged with the help of the Rodin tool. Once the refinement relationship between `M_Abs` and `MAbs_Helper` is established, it qualifies for trace refinement [27]. Now, we $\text{refine}(REQ2.1/MAbs)$. Even though the VE consists of two VTs, based on the tools we employ, we treat it as one trace and run it in the trace refinement tool, yielding a refined trace valid for `MAbs_Helper`. No new event was added (as the abstraction had the same events as its refinement). However, the abstract variables from the `M_Abs` were replaced by the concrete ones from `MAbs_Helper`. Because `MAbs_Helper` and $M9$ are semantically the same, as we previously established, we can also successfully replay the refined trace on $M9$, which was our goal. The process would then be analogous to the other suffixes.

4.4 Evaluation

We successfully applied the AVoiR framework to the AMAN case study. With the abstraction technique, we could provide an easily understandable domain-specific view of the possible user interactions to a domain expert. This would otherwise not be possible because the initial model's state space was too big and the model itself was too complicated, as shown in Table 1. Furthermore, we validated several requirements on our abstraction with the help of VOs and showed that these requirements are indeed implemented as part of M9 via VO refinement. However, the workflow could have been more convenient due to the lack of available tools. We plan to solve this issue in the future.

5 Related Work

In the context of state-based formal methods, (predicate) abstraction as a means of verification was previously applied to ASMs [8]. In contrast, we target validation, and our employed approach allows more flexibility for creating abstractions and reasoning over them. To our knowledge, refining formalized validation obligations is a novel idea. However, there is related work on abstractions, i.e., how to reduce details of models to better reason about them or different approaches for refinements.

CEGAR. The counterexample-guided abstraction refinement (CEGAR) method introduced by Clarke et al. [11] is a model-checking technique. With CEGAR, one takes an existing model and creates an abstraction with a smaller state space. Then, potential counterexamples found on the approximation are tested on the initial model. If they are false positives, the abstract model is corrected via refinement to no longer allow this counterexample. Our abstractions are not tailored towards model checking but can be helpful for any validation task, like animation, simulation [29] or enabledness analysis [13]. Still, CEGAR's idea of iteratively refining the abstraction until a property is satisfied could also be helpful for validation.

Alternative Abstractions. State space projections [7, 19] (which we used in Fig. 4) provide multiple abstract views on the state space of a model. To some extent, these are the precursor of our idea. However, they work at the level of explicit state space and not the model. To be fully precise, they need the entire state space (even though they can still be useful if only part of it is computed). AVoiR typically reduces the state space before applying projections and can be applied even if the concrete state space is large or infinite. Another related technique is GeneSyst [6], which provides an abstract view of the control flow graph of a classical B model.

Abstract interpretation [12] is an automatic abstraction technique mainly used for program analysis. It requires the development of an abstract domain and proving that the abstract operators are a sound approximation of the concrete ones. The abstract interpretation could be used to automate our approach if we identify a class of abstractions useful for a wider range of applications.

Decompositions. Abrial [3] and Butler [10] introduced the concept of decomposition for Event-B, i.e., decomposing a model into sub-models. These components can be further refined independently and, in the end, recomposed. As such, these approaches also tackle some of the issues our approach solves. The decomposition approach is motivated by the need to recombine the components, which imposes some restrictions. Our approach can, however, provide multiple non-disjoint abstract views on a system. Indeed, we do not need to partition the system into sub-components; we can focus in the abstractions on different features or aspects of the system which are relevant for validation. Thus both approaches are still complementary: our approach is useful for validation, while decomposition is helpful for code generation and compositional verification.

Retrenchment. Banach [4,5,14] introduced retrenchments, which can be imagined as a liberal version of refinement. As a result, the coupling between components is weaker than in a classical refinement relationship, allowing for higher modeling flexibility which is orthogonal to our concerns. It may be possible to combine the proposed abstraction approach with retrenchments.

CamilleX. As an extension to Event-B, CamilleX was proposed by Hoang et al. [17]. CamilleX features extensions that allow a more comprehensive and controlled refinement relationship between Event-B machines, thus helping in validation and verification effort. In contrast, our approach does not extend the existing language. Therefore it can be used without any new syntax or rules to learn. Furthermore, while creating an abstraction is cumbersome as it is done by hand, we look forward to providing tools for it in the future.

6 Conclusion and Future Work

This paper introduces the AVoIR framework for validating properties in complex models. The framework allows the creation of abstractions, a *reverse-like* operation to refinements, which works for complex models and helps validate desired properties using the VOs approach in simplified models. We then refine the VOs from the abstract model to re-establish the same properties in the initial complex model. The process helps domain experts quickly validate desired properties in complex models by tailoring a model for the task at hand, reducing

noise and the state space in the process. Finally, we demonstrate the efficacy of the proposed framework in a case study from the aviation domain.

In the future, we would like to test the AVoiR framework on further extensive case studies. Currently, the abstraction and the VO refinement process are manual. We also intend to develop tool support to automate this process.

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). <https://doi.org/10.1007/s10009-010-0145-y>
3. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to Event-B. *Fund. Inform.* **77**(1–2), 1–28 (2007)
4. Banach, R.: Graded refinement, retrenchment and simulation. *ACM Trans. Softw. Eng. Methodol.* (2022). <https://doi.org/10.1145/3534116>
5. Banach, R., Fraser, S.: Retrenchment and the B-Toolkit. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 203–221. Springer, Heidelberg (2005). https://doi.org/10.1007/11415787_13
6. Bert, D., Potet, M.-L., Stouls, N.: GeneSyst: a tool to reason about behavioral aspects of B event specifications. application to security properties. In: Treharne, H., King, S., Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 299–318. Springer, Heidelberg (2005). https://doi.org/10.1007/11415787_18
7. Bertolino, A., Inverardi, P., Muccini, H.: Formal methods in testing software architectures. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 122–147. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39800-4_7
8. Bianchi, A., Pizzutilo, S., Vessio, G.: Applying predicate abstraction to abstract state machines. In: Gaaloul, K., Schmidt, R., Nurcan, S., Guerreiro, S., Ma, Q. (eds.) CAISE 2015. LNBIP, vol. 214, pp. 283–292. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19237-6_18
9. Börger, E.: The abstract state machines method for high-level system design and analysis. In: Boca, P., Bowen, J., Siddiqi, J. (eds.) Formal Methods: State of the Art and New Directions, pp. 79–116. Springer, London (2010). https://doi.org/10.1007/978-1-84882-736-3_3
10. Butler, M.: Decomposition structures for Event-B. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 20–38. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00255-7_2
11. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM (JACM)* **50**(5), 752–794 (2003)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixed points. In: Proceedings POPL, pp. 238–252. ACM (1977)
13. Dobrikov, I., Leuschel, M.: Enabling analysis for Event-B. In: Science of Computer Programming, vol. 158, pp. 81–99. Elsevier (2018)

14. Fraser, S., Banach, R.: Configurable proof obligations in the frog toolkit. In: Proceedings SEFM, pp. 361–370. IEEE Computer Society (2007). <https://doi.org/10.1109/SEFM.2007.12>
15. Geleßus, D., Stock, S., Vu, F., Leuschel, M., Mashkoor, A.: Modeling and analysis of a safety-critical interactive system through validation obligations. In: Proceedings ABZ (2023)
16. Hoang, T.S., Schneider, S., Treharne, H., Williams, D.M.: Foundations for using linear temporal logic in Event-B refinement. *Formal Aspects Comput.* **28**(6), 909–935 (2016). <https://doi.org/10.1007/s00165-016-0376-0>
17. Hoang, T.S., Snook, C., Dghaym, D., Fathabadi, A.S., Butler, M.: Building an extensible textual framework for the Rodin platform. In: Masci, P., Bernardeschi, C., Graziani, P., Koddenbrock, M., Palmieri, M. (eds.) *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops. LNCS*, vol. 13765, pp. 132–147. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-26236-4_11
18. Institute of Electrical and Electronics Engineers: IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. IEEE (1991). <https://doi.org/10.1109/IEEESTD.1991.106963>
19. Ladenberger, L., Leuschel, M.: Mastering the visualization of larger state spaces with projection diagrams. In: Butler, M., Conchon, S., Zaïdi, F. (eds.) *ICFEM 2015. LNCS*, vol. 9407, pp. 153–169. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_10
20. Lamport, L.: *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston (2002)
21. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) *FME 2003. LNCS*, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
22. 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). <https://doi.org/10.1002/spe.2634>
23. Mashkoor, A., Leuschel, M., Egyed, A.: Validation obligations: a novel approach to check compliance between requirements and their formal specification. In: *ICSE2021 NIER*, pp. 1–5 (2021)
24. Palanque, P., Campos, J.C.: Aman case study (2022). <https://drive.google.com/file/d/1IqftxQIvrWpX1lRts3WJzrBH7a3dMln/view>
25. Punnoose, R.J., Armstrong, R.C., Wong, M.H., Jackson, M.: Survey of existing tools for formal verification. Technical report, Sandia National Lab. (SNL-CA), Livermore, CA (United States) (2014). <https://doi.org/10.2172/1166644>
26. Schneider, S., Treharne, H., Wehrheim, H., Williams, D.M.: Managing LTL properties in Event-B refinement. In: Albert, E., Sekerinski, E. (eds.) *IFM 2014. LNCS*, vol. 8739, pp. 221–237. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10181-1_14
27. 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. ICFEM 2022. Lecture Notes in Computer Science*, vol. 13478, pp. 316–333. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-17244-1_19
28. Stock, S., Vu, F., Mashkoor, A., Leuschel, M., Egyed, A.: IVOIRE Deliverable 1.1: Classification of existing VOs & tools and Formalization of VOs semantics. arXiv preprint: [arXiv:2205.06138](https://arxiv.org/abs/2205.06138) (2022)
29. 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

30. Yeganefard, S., Butler, M., Rezazadeh, A.: Evaluation of a guideline by formal modelling of cruise control system in Event-B. In: Proceedings NFM, pp. 182–191 (2010)
31. Zhu, C., Butler, M., Cirstea, C., Hoang, T.S.: A fairness-based refinement strategy to transform liveness properties in Event-B models. *Sci. Comput. Program.* **225**, 102907 (2023). <https://doi.org/10.1016/j.scico.2022.102907>, <https://www.sciencedirect.com/science/article/pii/S016764232200140X>