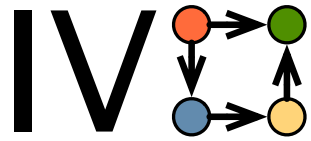

IVOIRE - Deliverable D 1.1

February 18, 2022

Version 1.1.0

Sebastian Stock, Fabian Vu, Atif Mashkoor, Michael Leuschel, Alexander Egyed

Contents

1 Introduction

This document presents the report for D1.1 ("Classification of existing Validation Obligations and Tools") of the IVOIRE project.
Within the context of software engineering, it is vital to check whether a model meets its specification (verification) and requirements (validation). Various formal techniques exist to verify a requirements model, such as theorem proving or model checking. However, to validate a requirements model, only a few formal techniques are at our disposal. Verification checks that a system or piece of software meets its specification. This means that the program is proven to be correct concerning its specification [98]. **Verification** answers the question: "Are we building the software correctly?" In contrast, **validation** checks whether a model met the stakeholder's requirements [97]. It answers the question: "Are we building the right software?".

In general, verification ensures the system's consistency with given safety properties (e.g., absence of infinite loops, absence of integer overflows). Successfully verified systems cannot violate those safety properties. Regarding verification, proof obligations (POs) have been introduced, e.g., for B [1] and Event-B [2]. Formally, POs are logical formulas extracted from the specification, which must be proven. A successfully discharged PO ensures the absence of a safety hazard in a model.

Validation ensures the presence of certain features. Therefore, a successfully validated system contains desired behaviors, e.g., the ability to perform actions in a specific order. Furthermore, verification and validation might overlap, e.g., establishing the absence of deadlocks can depending on the context be viewed as either as verification or as validation.

Compared to verification, validation has received less attention historically in the formal methods community. This is perhaps due to its unprovable nature, as suggested by Rushby: "By their very nature, the problems of validating top-level specifications or statements of assumptions do not lend themselves to definitive proof" [80]. Nonetheless, as outlined by Jacquot and Mashkoor [49] validation within a refinement-based development process is still a challenging task.

This report defines the term *validation obligation*, and present a classification of VOs and the underlying techniques, tackling the second question. Validation obligations (VOs) [69] can be used to check compliance of a model with its requirements in a refinement-based software development process. In this report, we provide a formal basis for this approach. VOs intend to achieve the following goals: (1) VOs should systematically validate certain given requirements, and (2) VOs should ease tracing the model to its requirements. Tracing requirements is important when developing the model which will be shown later in this report. VOs must therefore be resilient when more details are added to a model. As requirements are typically expressed in natural language, they can be ambiguous or imprecise. Although a VO aims to be a formal representation, some validation tasks may retain a manual and/or informal component. For example, a certain VO may produce a visualization that still has to be inspected

by a domain expert.

First, we will present the classification of requirements (see Section 2). In Section 3, we will discuss the terminologies *verification* and *validation*. Then, we define the term *validation obligation* along with how associated *validation tasks* are formalized and classified (see Section 4). Here, we also present an overview of existing validation tasks for the modeling languages Alloy, ASM, B, Event-B, VDM, TLA+, Z, CSP, and Circus. Furthermore, we will describe how VOs are used in a refinement-based software development process to validate requirements. Finally, we will demonstrate the VO approach on a Traffic Light model in Section 5.

A glossary containing the basic terms can be found in Appendix B. This report also includes a list of publications in the context of the IVOIRE project Appendix E.

2 Classification of Requirements

By definition according to the IEEE standard 729 [46], a requirement is defined as follows:

1. A condition or capability needed by a user to solve a problem or achieve an objective.
2. A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.
3. A documented representation of a condition or capability as in 1 or 2.

This section describes how requirements are classified. In general, they are separated into functional requirements, non-functional requirements, and domain requirements. Furthermore, requirements could also be distinguished between user requirements, and system requirements. [86]

2.1 Functional, Non-Functional, and Domain Requirements

Functional requirements describe how the system should behave. Regarding the general definition of a requirement, functional requirements match the definition of the first aspect. [86]

Thus, functional requirements include descriptions of safety properties, liveness properties, operational behaviors, scenarios, probabilistic behaviors, timing behaviors.

In contrast, non-functional requirements describe measurements or constraints for the quality of the software system such as performance, reliability, maintainability, testability, scalability, or security. Thus, non-functional requirements define criteria to evaluate those quality constraints or measurements. [86]

Taking a look at the definition of a requirement, non-functional requirements correspond to the second aspect.

Domain requirements are requirements that have been formulated from a domain expert's perspective. Therefore, domain requirements can either be functional or non-functional. [86]

Concerning the domain-specific aspect, it might be necessary to refine or abstract the model, projecting on the domain expert's perspective. Since the model is projected on a specific perspective, state space projection and refinement might play an important role during the validation.

2.2 User Requirements and System Requirements

Requirements could also be distinguished between user and system requirements. User requirements are written from a stakeholder's perspective, describing the expectation of how a system should behave. Therefore, user requirements usually describe how the user can interact with the model, and check the software's behavior. In contrast, system requirements describe how software components interact with each other. Thus, system requirements are rather architectural or structural. [86]

3 Verification and Validation

Sometimes the terms verification and validation are used interchangeably. In our paper, we use the following definitions. Validation checks whether a model meets the stakeholders' requirements. So, the main questions are: "Are we building the right software?" and "Are the desired features present?" In contrast, verification checks whether a model meets its specification. So, it tackles the questions: "Are we building the software correctly?", "Are all safety constraints be enforced?".

Validation can take on many forms. Some requirements can be validated by running the software with specific input, to observe the behavior, and check the correctness afterwards. For example, functional requirements described by scenarios can be validated by trace replay or simulation (see Section 4). Other requirements cannot be expressed as scenarios, e.g., requirements describing liveness properties or maybe even safety properties including invariants. For this purpose, model checking techniques, and proving (see Section 4) are also considered as validation techniques. There are also tools such as FRET [39], and SpeAR [30] where model checking is used as a validation technique to validate behavioral requirements. As an example, let us consider a requirement describing a safety property such as "The lift can only move when the door is closed". One might argue that checking these kinds of requirements belongs to verification rather than validation. However, the main goal here is also to check that the stakeholders' requirements are fulfilled. Note that we agree that model checkers and theorem provers are primarily verification tools, but we believe that they can also be used for validation purposes.

We have also encountered properties that clearly correspond to verification (and not validation), e.g., absence of well-definedness, absence of integer over-

flows, and absence of infinite loops. Those properties could be verified by model checking or proving as well. However, these techniques could still be important to ensure the model's and VOs' consistency. While verification and validation are different tasks, they sometimes complement each other. Consider that there are well-definedness violations in a model which are clearly classified as verification. In the case that such an error is found during the development process, one could store the trace as a test, desiring the absence of well-definedness errors.

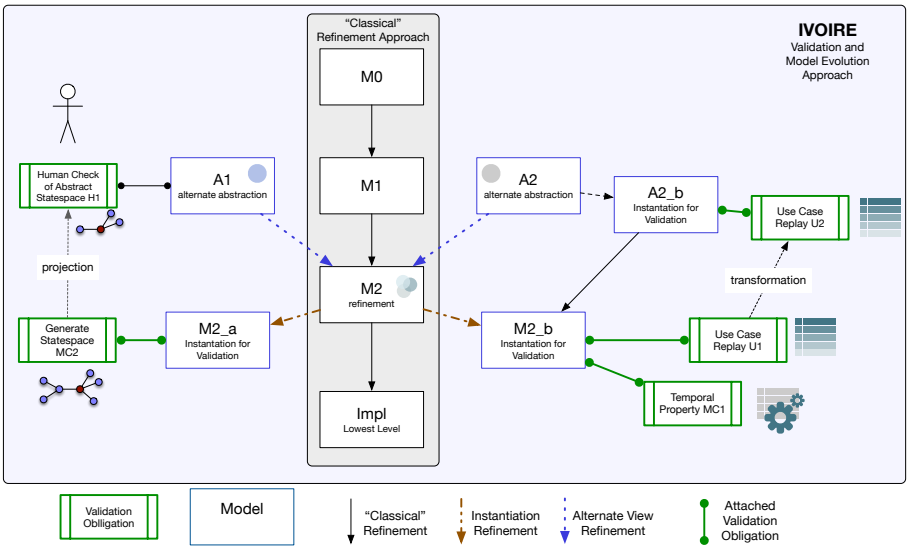

Figure 1: Refinement-based Software Development Process with VOs

4 Validation Obligations Approach

This section presents a formalization and classification of validation obligations (VO) and validation tasks (VT). As explained by Mashkoor et. al [69], refinement plays an important role in the VO approach. Therefore, the formal representation of VOs should also provide a basis to transform, refine or abstract VOs. Furthermore, this work discusses how VTs are created and how VOs are integrated into the software development process.

The idea of a refinement-based software development process assumes that a formal model is developed incrementally, i.e., step-by-step (see Figure 1). This means, that a model's refinement is created for each development step (black/solid-line arrows). Later down the refinement chain, more requirements are taken into account. VOs shall be used to ensure the presence of requirements in a refinement-based software development process.

To achieve this, newly introduced requirements must be validated incrementally at each model's refinement. Additionally, there might be the need for abstracting (illustrated by the blue/dotted arrows) or specializing/instantiating (illustrated by the red/dotted arrows) the model for a domain expert, only focusing on specific requirements. Note that the concept of refinement is already supported in some formalisms (e.g. B and Event-B), but the concept of multiple distinct abstractions is novel, as far as we are aware.

4.1 Refinement and Refactoring

Refinement is an essential technique to enrich models while ensuring their correctness. As shown in Figure 1, the refinement chain has a crucial role in the context of VOs, too. First, there is the classical refinement chain making up the middle of the figure. Here, the model is consecutively enriched with behavior and details. But there are also instantiation refinements and alternate view refinements going to the left and the right.

Instantiation Refinements are those that make a model particular for a use case, e.g., by providing an initialization for the variables.

Alternate View Refinements are those that allow a more abstract view onto the model, enabling easier reasoning, e.g., by ignoring behavior that is not relevant to validate a property, showing this property can become easier.
Multiple problems arise from that:

1. How should such refinements interact with each other? On the right-hand side of Figure 1, one can see that $A2\_b$ is refined by $M2\_b$, which is also an instance of $M2$. It is an ongoing question of how the relationship between these components should be allowed and formally defined in the first place. The problem of these multi-layer relationships is keeping track of the changes and dependencies. Furthermore, every abstraction has to be validated, and in the case of the multi-layer relationship, the abstraction $A2$ has to be verified as the refinement of the instance of $A2\_b$. Even with the minimal example, this would result in a set of new proof obligations that would have to be shown to ensure a correct refinement in the first place.

2. How should VOs be refined? When doing linear refinement, a VO that holds on $M1$ has to hold on $M2$. But what about nonlinear refinement? A VO on $A2\_b$ has to hold too when going down the refinement chain, but how is this shown? Imagine introducing $M3$ refining $M2$. Does one needs to create an $A3\_b$ to show the VO?. There could be a case where this abstraction is no longer feasible as new behavior entangles components that were only loosely connected before. An idea would be to reduce every VO from a nonlinear refinement back to the origin in the linear refinement chain, which will be researched and discussed in the future.

3. On the right-hand side of Figure 1 one can see a VO transformation. We do not know yet what this means in practice. And what the applications

and restrictions are.

Allowing combining abstraction, specializations, and refinements as shown in Figure 1 gives maximum freedom to the modeler for the price of simplicity. Creating a lot of instances and abstractions is easy but translating and tracking the VOs for each part of the model might get hard, possible losing sight of the original goal.

Besides, formal refinement models can be refactored e.g. changing the name of variables, or altering the state space by changing the behavior of operations. In this case, VOs are invalidated similar to POs. There might be tool support to adapt the VOs and especially their tasks to this in the future. For now, this is not an immediate concern as refactoring should not change the behavior of a model but the quality of life of the modeler.

4.2 Validation Techniques and Tasks

Before we define the term VO, we will first formalize the subsidiary concept of a *validation task* (VT). A validation task (VT) is identified with an identifier, and consists of a validation technique that is applied with the given validation parameters to the corresponding context. Executing a VT possibly modifies the internal state of the validation tool, e.g., consisting of the currently explored state space, and the current trace. The notation we will use for a VT is as follows:

$$\mathbf{VT_{id}}/\mathrm{VT}_{\mathrm{context}}/\mathrm{VT}_{\mathrm{technique}}\colon \mathrm{VT}_{\mathrm{parameters}}$$

For example, the following VT identified by $\mathbf{LTL_1}$ means that LTL model checking should be applied with the LTL formula `G{tl_peds = red` $\vee$ `tl_cars = red}` expecting a successful result in the Traffic Light model.

$\mathbf{LTL_1}$/TrafficLight/LTL: G{tl_peds = red $\vee$ tl_cars = red}, SUCCESS

Validation tasks can be combined into larger ones. For two validation tasks $\mathbf{VT_1}$ and $\mathbf{VT_2}$, we define two operators: $\mathbf{VT_1}; \mathbf{VT_2}$ (sequential composition) and $\mathbf{VT_1} \mid\mid \mathbf{VT_2}$ (parallel composition). Sequential composition means that $\mathbf{VT_2}$ is executed after $\mathbf{VT_1}$, based on the resulting state from $\mathbf{VT_1}$. For example, $\mathbf{VT_1}$ could apply model checking searching for a state, which is then used as the initial state of a trace replay task $\mathbf{VT_2}$. Parallel composition means that $\mathbf{VT_1}$ and $\mathbf{VT_2}$ are executed independently from each other.

Table 1: Overview of Validation Tasks, Auto = Automatically, Man = Manually, Part = Partially

Name	Task	Context	Parameters	Discharged
TR [4] [5]	Trace Replay/ Animation	Model	Trace T	Auto[1]
HT [4] [6]	Simulation + Hypothesis Testing	Model, Simulation	Number of Simulations N, Hypothesis H [3], Significance level α	Auto
EOP [4] [6]	Simulation + Estimation of Probability	Model, Simulation	Number of Simulations N, Property P [3], Delta value δ	Auto
OC [4] [6]	Operation Coverage Test Case Generation	Model	Operations O, Depth δ	Auto
MCDC [4] [6]	MC/DC Test Case Generation	Model	Level l, Depth δ	Auto
MC [4] [5] [6]	Explicit-state Model Checking	Model	Configuration $c \in Conf_{MC,F}$	Auto
SMC [6]	Symbolic Model Checking	Model	Configuration $c \in Conf_{MC,F}$	Auto
LTL [4] [5] [6]	LTL Model Checking	Model	LTL Formula ψ, $c \in \{\text{SUCCESS}, \text{FAIL}\}$	Auto[2]
CTL [4] [5] [6]	CTL Model Checking	Model	CTL Formula ψ, $c \in \{\text{SUCCESS}, \text{FAIL}\}$	Auto[2]
PSMC [4] [6]	Probabilistic/Statistical Model Checking	Model	Probabilistic Temporal Formula ψ [3], $c \in \{\text{SUCCESS}, \text{FAIL}\}$	Auto[2]
PO	Proving	Model	Formula ψ	Part
SVIS [7]	Inspection of State Space Visualization	Model	Formula ψ over state space $S_{\text{vis}} = (Z_{\text{vis}}, T_{\text{vis}})$	Man [9]
SPRJ [7]	Inspection of State Space Projection	Model	Expression ϕ, Formula ψ over projected state space on ϕ: $S_\phi = (Z_\phi, T_\phi)$	Man [9]
STAT [8]	Inspection of State Space Statistics	Model	Formula ψ over state space statistics $R_{\text{stat}} \subseteq P_{\text{stat}} \times \mathbb{R}_{\geq 0}$ [3] [10]	Man [9]
SISTAT [8]	Simulation + Inspection of Simulation Statistics	Model, Simulation	Number of Simulations N, Start Condition C_{st}, End Condition C_{end}, Formula ψ over simulation statistics $R_{\text{sistat}} \subseteq P_{\text{sistat}} \times \mathbb{R}_{\geq 0}$ [3] [10]	Man [9]
ED [7]	Inspection of Enabling Diagram	Model	Formula ψ over enabling diagram $R_{\text{ed}} \subseteq E \times E$ [11]	Man [9]
OCT [8]	Inspection of Operation Coverage Table	Model	Formula ψ over operation coverage table $R_{\text{oct}} \subseteq E \times \{\text{COV}, \text{UNCOV}\}$ [11]	Man [9]
RWM [8]	Inspection of Read/Write Matrix	Model	Formula ψ over read/write matrix $R_{\text{rwm}} \subseteq \{\text{READ}, \text{WRITE}\} \times (E \times V)$ [11]	Man [9]
VCT [8]	Inspection of Variable Coverage Table	Model	Formula ψ over variable coverage table $R_{\text{vct}} \subseteq V \times \mathbb{N}_0$ [11]	Man [9]
MMV [8]	Inspection of Min/Max Values	Model	Formula ψ over min/max values table $R_{\text{mmv}} \subseteq V \times (\bigcup_{t \in V} \tau(t) \times \bigcup_{t \in V} \tau(t))$ [11] [12]	Man [9]
VAP	Vacuous Parts Check	Model	Configuration $c \in \{\text{GRD}, \text{INV}\}$	Auto

[1] succeeds when scenario is replayable and all tests succeed
[2] succeeds when ψ is fulfilled for SUCCESS or ψ failed for FAIL
[3] depends on the tool that is used
[4] explores state space
[5] updates current trace (model checking only for FAIL and GOAL)
[6] generates (counter)-examples for a domain expert
[7] generates visualization for a domain expert
[8] generates table for a domain expert
[9] succeeds when formula is fulfilled on visualization, table, statistics, ...
[10] P_{stat} and P_{sistat} denote the respective set of (simulation) statistics properties
[11] E denotes the set of events/operations, V denotes the set of variables
[12] $\tau(v)$ denotes the set of all values which can be of v's type

Remark: $Conf_{MC,F}$ is currently definied as $\{< \text{FIN} >, < \text{DLF} >\} \cup \{< \text{INV}, \psi > | \ \psi \in F\} \cup \{< \text{GOAL}, \psi > | \ \psi \in F\}$, where F denotes the set of formulas in the supported formalism, FIN denotes checking for finite state-space, DLF denotes checking for deadlock-freedom, INV denotes invariant checking, and GOAL denotes searching for a goal.

In the following, we will formalize the validation tasks, and discuss the underlying validation techniques. In particular, we will also outline the strengths and weaknesses of each technique. There are various validation techniques one can use to validate a requirement. Table 1 contains an overview of VTs we have assembled while conducting or inspecting a variety of case studies. These case studies range from academic case studies (e.g., the ABZ landing gear case study [58], and the ABZ automotive case study [64]) to industrial applications in the railway industry (e.g., [42, 23]). Regarding the future, Table 1 and operators might gradually evolve for new case studies or applications.

Validation by Animation, Trace Replay, Testing Animation makes it possible for a human to execute the model interactively. Some animators explore all transitions from the current state to the succeeding states. This is done by interpreting the operational semantics of the used formalism on the model with all possible values for parameters and variables that are assigned non-deterministically. Regarding the notion for a transition, possible means that the corresponding guard is met. [49]

The main advantage of animation is that the user can interact with the model and view the model's state after executing an action. Thus, this validation technique makes it possible to reason about the model more easily. When an animator explores all succeeding transitions, the user also gets the information on which actions can be applied outgoing from the current state. This eases the interaction with the model in a way that the user does not need to think about which input parameters are required to constraint the guard. Nonetheless, it is then necessary to iterate over the possible values for parameters and non-deterministically assigned variables which leads to a combinatorial explosion of possible transitions. [49]

Outgoing from an animation process, the modeler could store the resulting trace representing a scenario with certain behaviors. Later on, the trace can be used to re-play the scenario, i.e., to check whether the scenario is still re-playable from the model (realized by **TR**). Trace replay is applied similar to animation, but with the main difference that it is done automatically.

A trace T consists of a list of the transitions $t_1, \ldots, t_n$. For each transition, the modeler could optionally add a predicate ψ to be checked after re-playing the transition. Here, we will use the notation $t <\psi>$ for a transition t and a predicate ψ. If there is no postcondition, we will use the notation t only.

Traces can then also be viewed as acceptance and unit tests which are well-known in traditional programming practice. Thus, they can then be used to ensure the presence of certain behaviors in the model.

Note that the form of a transition depends on the used formalism. E.g., in the B method, a transition consists of the operation's name, the values for parameters, and the values for non-deterministic assigned variables.

Validation by Simulation Simulation such as *co-simulation* [95] or *timed probabilistic simulation* [99] can be used to execute a model automatically. Here,

the modeler can define respective configurations or annotations to define simulation scenarios. Monte Carlo simulation [72] can be applied in the context of timed probabilistic simulation to generate a various number of simulations. Based on the resulting execution runs, statistical validation techniques such as *hypothesis testing* [52] (realized by **HT**), or *estimation of probability* [31] (realized by **EOP**) can be applied to show the presence of a behavior [99].

Sometimes, systems consist of several different components or subsystems that interact with each other. Each subsystem might be embedded into a different tool, or even modeled with a different formalism. Co-simulation implements the idea of combining the components into an overall system for simulation. In particular, the subsystems and their communication with each other are simulated in parallel. Regarding the communication itself, subsystems must exchange data with each other which again might trigger events. [95]

Regarding *timed probabilistic simulation*, the modeler can simulate the underlying model with timing and probabilistic behavior. Each simulation results in a trace where each executed event is annotated with a certain time, called *timed trace*. A timed trace can then be replayed in real-time, i.e., wall-clock time.

Validation by Test Case Generation Test case generation tries to satisfy a given coverage criterion by generating tests for a model. The desired coverage criterion is satisfied if each possible branch is covered by a test. Thus, each generated test is represented by a trace which can be seen as a scenario representing a certain property. Therefore, test case generation is a validation technique that can be used to generate new scenarios which again can be validated by animation, trace replay, and testing. Coverage criteria include operation coverage (realized by **OC**) and MC/DC coverage (realized by **MCDC**). While the goal of operation coverage is to cover each operation, MC/DC coverage is used to cover all possible outcomes of each operation. [82, 102]

Validation by Model Checking Explicit-state model checking checks state-based behaviors of a system by exploring its state space exhaustively (realized by **MC**). Exhaustive exploration leads to full coverage of the system's behavior when the model checking process terminates. Furthermore, it is then ensured whether the property is fulfilled or not. In the case that a property is violated, explicit-state model checking can return a counter-example. Again, when applying model checking to find a state satisfying a certain property, the technique can also provide a path leading to this state. Nevertheless, explicit-state model checking often struggles with the combinatorial explosion of the state space which is called the state space explosion problem. This is because the number of states in a state space grows exponentially wrt. the number of variables in a model. [7]

Temporal model checking includes LTL and CTL model checking. LTL model checking checks a temporal property (expressed as LTL formula) that is expected for the given system (realized by **LTL**). Using the transition system

and the Büchi automaton that is created from the LTL formula, LTL model checking checks temporal properties which are more complex than state-based properties. When negating the LTL formula, one is also able to find an example where the temporal property is true. [7]

To formulate more expressive temporal properties, the modeler could also write CTL formulas and apply CTL model checking (realized by **CTL**). Compared to LTL, CTL supports the operators $A\phi$ (ϕ is true for all paths), and $E\phi$ (it exists at least one path where ϕ is true). [7]

As the state space is also explored exhaustively, there are the same advantages and disadvantages as explicit-state model checking. [7]

Symbolic model checking (realized by **SMC**) bases on the idea of getting rid of the state-space explosion problem. To achieve this, the state space is not explored explicitly. Instead, logical formulae are derived from the model and then checked for solutions where properties are violated. Symbolic model checking makes use of techniques such as SMT solving and abstract interpretation which are realized in the algorithms for constraint-based model checking, bounded model checking, k-Induction and IC3 etc. As the symbolic evaluation of the model is an over-approximation, there might be some false positives. Furthermore, the counter-example might also be abstracted which leads to a loss of information. [55]

By assigning probabilities to events in a model, a state space could be generated on which transitions are labeled with probabilities. As result, the state space can be viewed as a Markov chain on which probabilistic model checking can be applied to validate probabilistic properties. It is also possible to validate probabilistic temporal properties, e.g., properties that are encoded with PLTL, PCTL, or PB-LTL formulas. Similar to probabilistic model checking, statistical model checking also aims to check probabilistic properties. The main difference is that statistical model checking applies Monte Carlo simulation, whereupon PB-LTL or BLTL formulas are checked with hypothesis testing or estimation. [60, 61] Both model checking techniques are realized by **PSMC**.

As mentioned before, timed probabilistic simulation also applies Monte Carlo simulation together with statistical validation techniques. However, timed probabilistic simulation does not check temporal formulas. Instead, the modeler can specify a property (with timing behavior if desired) along with a start and end condition which should be checked. [99]

Validation by Proving Proving is a technique that is used to ensure the model's consistency, i.e., to show the correctness of the program in certain aspects. To achieve this, proving is often applied to proof obligations which are formulas that are generated from the model (realized by **PO**). Relevant aspects could be e.g., the violation of invariants, deadlocks, well-definedness errors, or refinement errors. The process of proving itself is both automatic and interactive [2].

In practice, different solvers are applied to try to prove a formula. However, solvers are sometimes not strong enough to prove a formula. The proof must

then be done by the user interactively with additional effort.

The main purpose of proving is to ensure that the model does not contain any errors which seems to be rather verification than validation. As discussed in Section 3, we also see proving as validation.

Validation by Visualization, Statistics, and Metrics As already mentioned before, animation is particularly important for validation as the user might want to interact with the model to view the resulting state afterwards. Sometimes, the modeler even wants to inspect visualizations, statistics, or metrics, to reason about the model. The following VTs we consider are:

- *State Space Visualization:* One possibility consists of visualizing and inspecting the whole state space after applying certain steps. For a given state space consisting of reachable states, and possible transitions, one can formulate a predicate over the state space to be checked. This is realized in **SVIS**.
- *State Space Projection:* In practice, state spaces often become very large due to the state space explosion problem. As result, the visualization gets too complex to understand. To solve this problem, the modeler could provide an expression to create a state space projection onto this expression. This results in an abstract visualization of the state space which is easier to understand. [57] Similar to the state space inspection, one could also formulate a predicate to be checked over a projected state space (realized by **SPRJ**). The projected state space could also provide a base to apply other VTs, e.g., LTL model checking.
- *Enabling Diagram*: An enabling diagram is a diagram that describes for each operation which operations is enabled after executing this operation [28]. This helps to inspect how operations could depend on each other. The corresponding VT (see **ED**) expects a formula that is checked over the diagram.
- *Operation Coverage Table*: This table describes for each operation whether it is covered yet. Formally, one can provide a formula that is checked on the table as shown in **OCT**. Combining this task with other VTs, one can evaluate the coverage of the other VTs. For example, **OCT** could be applied after running a set of scenarios. Afterwards, one can then measure the coverage, and thus also the quality of the given set of scenarios.
- *Read/Write Matrix*: The read/write matrix is a table describing for each operation which variables are read and written. Similarly, one can inspect this matrix by checking a certain predicate (realized in **RWM**). This helps to inspect which parts of a state are influenced by an operation and vice versa.
- *Variable Coverage Table*: This table provides the number of values a variable has been assigned to. Similarly, one can also provide a predicate

that is checked over the table (realized in **VCT**). This task is usually also combined with other VTs, to evaluate their coverage wrt. to the model's variables.

- *Min/Max Values for Variables*: This table shows the minimum and maximum value each variable has been assigned to. Based on the table, one can also formulate a predicate which is checked on the table as shown in **MMV**. In combination with other VTs, this task also helps to inspect the coverage. For example, one can then inspect why the state space explodes.

- *State Space Statistics*: To validate a model, the modeler could also take state space statistics into account. Interesting statistics for the state space, could be, e.g., the number of states, or the number of transitions. One could also extract more complex statistics, e.g., the number of states with a certain property such as invariant violation, deadlock, or liveness. To determine which events are particularly important for the model, one could also take the number of transitions for each event into account. This task is also formalized with a predicate over the state space statistics as shown in **STAT**.

- *Simulation Statistics*: Based on a model and a simulation, one can also inspect simulation statistics. Interesting properties are, e.g., statistics about how frequently an operation is executed in the simulation. One could also inspect the percentage of how frequently an operation is executed when it is enabled. Similar to **STAT**, this is also formalized with a predicate over the statistics (see **SISTAT**).

Vacuous Guards/Invariants: There are also requirements made about the model's internal structure, rather than its functionality. For example, there could be a requirement, desiring that there are no vacuous parts in the invariant or an operation's guard. Therefore, we have introduced the corresponding VT which is formalized as **VAP**.

Code Generation for Validation In contrast to the aforementioned tasks, we do not see code generation as a validation task directly. Instead, it is rather a task that can be applied to enable other validation tasks afterwards.

During the software development process using formal methods, software is specified and refined step by step. Once a refinement level is reached which is close to implementation constructs, a code generator is applied. Regarding the B method, a code generator for embedded systems can be applied once the B0 language is reached, which is the implementable subset of B [22]. As an implementable subset of the specification language is required, memory usage of the final refinement can be verified. Thus, the generated code can be used for embedded systems. Additionally, the software engineer could also write or generate tests to validate the generated code. This also means that these validations are applied at the very end of the software development process.

Communication with the stakeholder and early-stage validation is particularly important in the context of VOs. To achieve this goal, our approach intends to take high-level code generators such as B2Program [100] into account. B2Program is suitable for application for early-stage validation of the software, but cannot be used to generate code for embedded systems. Based on the generated code, the model could then be animated, tested, and simulated.As the model is translated to a programming language, it could also be more familiar to the domain expert to work with it compared to working in the context of formal methods.

Languages and their tools For our research we have investigated nine major modeling languages regarding their tool support for different tasks. The results are shown in Table 2. Whenever something is marked with ✗, we did not find referable evidence for the existence of the respective tool support.
One can see that tool support is widely spread. As we use the ProB platform as starting point for further development, the B and Event-B languages are especially appealing as they are covered by most of the features we investigated.

Table 2: Specification Languages and Supported Validation Techniques

Tools	Alloy	ASM	B	Event-B	VDM	TLA+	Z	CSP	Circus
Animation	✗	✓([12])	✓([62, 101])	✓([11, 62, 101])	✓([74])	✓([41])	✓([11, 25, 78])	✓([11, 18])	✗
Trace Replay/Testing	✓([93, 24]) [1]	✓([19])	✓([10])	✓([85, 10])	✗	✓([41, 10])	✓([27])	✓([20])	✗
Test Case Generation	✓([88])	✓([35, 36, 37])	✓([94])	✓([82, 102, 94])	✓([29])	✓([41, 94])	✓([44, 94])	✓([73])	✗
Simulation	✓([16])	✓([38])	✓([99])	✓([99])	✓([95, 32])	✓([99])	✓([99])	✓([99])	?
Explicit-State MC	✓([17, 24])	✓([5])	✓([62])	✓([11, 62])	✓([66])	✓([104, 41])	✓([78, 11])	✓([40, 11])	✗
LTL MC	✓([17, 24])	✓([5])	✓([77])	✓([77])	✓([66])	✓([41, 77])	✓([77, 26])	✓([77, 63])	✗
CTL MC	✓([96])	✓([5])	✓([62])	✓([11, 62])	✓([66])	✓([11, 62, 41])	✓([26])	✓([62])	✗
Symbolic MC	✓([17, 24])	✓([5])	✓([55])	✓([55])	✗	✓([54, 53, 55])	✓([83])	✓([89])	✗
Probabilistic/Statistical MC	✗	✗	✗	✓([4, 92])	✗	✗	✗	✗	✗
Proving	✓([75])	✓([8])	✓([71])	✓([1, 2])	✓([3])	✓([21])	✓([15, 33])	✓([91, 47])	✓([34])
Refinement Checking	✗	✓([6, 14])	✓([71])	✓([1, 2])	✓([67])	✓([90])	✓([87])	✓([40])	✓([34])
State Space Visualization	?	?	✓([57])	✓([57])	✗	✓([56, 57])	✓([57])	✓([62, 11, 18])	✗
Code Generation [2]	✗	✓([13])	✓([100])	✓([79])	✓([43])	✗	✗	✓	✓([9])

4.3 Definition of Validation Obligation

We define the term *validation obligation* as follows:

> A *validation obligation* (VO) is composed of (multiple) validation tasks (VT) associated with a model to check its compliance with a particular requirement.

Thus, validating a requirement succeeds if all associated VOs yield successful results. Formally, a VO is annotated with an id, and consists of a *validation predicate* (VP). The VP consists of logical operations (with short-circuiting) on the associated VTs' validation results. Therefore, a VO succeeds, if the

[1] In Alloy, it seems that it is not possible to animate the model interactively. Nonetheless, it is still possible to test the feasibility and behavior of a scenario. Here, it seems that scenarios have to be encoded manually. Furthermore, note that Alloy only supports infinite traces

[2] High-Level Code Generation for (Early-Stage) Validation

corresponding $\text{VO}_{\text{predicate}}$ leads to a successful result. The notation we will use for a VO is as follows:

$\mathbf{VO_{id}}$: $\text{VO}_{\text{predicate}}$

Let V be the set of VTs, and let $\mathbb{B} = \{\text{TRUE}, \text{FALSE}\}$ be the set of booleans. Validating a VT is described as a function *validate* which executes a VT in the corresponding context. After executing the VT, the result is a boolean value describing whether the validation succeeds.

$$\text{validate} : V \to \mathbb{B}$$

An example for a VO that validates $\mathbf{LTL_1}$ is shown in $\mathbf{VO_1}$. $\mathbf{VO_1}$ succeeds if $\mathbf{LTL_1}$ discharges successfully.

$\mathbf{VO_1}$: validate(LTL_1)

As mentioned before, executing a VT modifies the animator's current state. For a VT T, *validate*(T) executes the task from the model's initial state. Again, a VO is executed from the animator's initial state, i.e., where no state in the state space has yet been explored. Another important aspect is the VO's consistency, i.e., that associated validation tasks do not contradict each other. For the validation of a task consisting of the form **VT1** || **VT2**, there is no problem as **VT1** and **VT2** do not depend on each other. Still, *validate*(**VT1** || **VT2**) only succeeds if *validate*(**VT1**) succeeds and *validate*(**VT2**) succeeds. For **VT1**; **VT2**, there is a dependency of **VT2** from the result of validating **VT1**. Thus, *validate*(**VT1**; **VT2**) requires *validate*(**VT1**) to succeed before the whole task can be applied.

4.4 Creating Validation Obligations

Currently, a VO is created by the modeler manually. There are also tools like UML-B [84], which attempt an automatic translation from the specification to a model. Regarding the future, one could explore whether and how VOs can be extracted from the requirements automatically. Here, we could take FRETIish or SpeAR into account to write behavioral requirements in natural language. For now, we have decided not to use requirement languages as this would add more complexity. As explained before, our work covers a wide range of validation techniques. Therefore, introducing a requirements language for each kind of requirement increases the complexity and makes this approach hard to use.

Later in Section 5, one can see a requirement where trace replay is applied after model checking (searching for a state). Furthermore, there is also a requirement describing a coverage criterion based on resulting traces from other

tasks. When creating a VO, the modeler needs significant knowledge about the modeling language, and about the environment and the techniques to create suitable tasks. For example, the preservation of an invariant can be shown by model checking or proving. Proving and symbolic model checking are therefore more suitable than explicit-state model checking to check an invariant in an infinite-state system. Another aspect is to check whether the property formulated in the VO actually captures the stakeholders' needs. As natural language is ambiguous, communication and feedback from the stakeholders are important.

4.5 VO-guided Workflow

Requirements engineering and software development are highly entangled processes. During the software development process, requirements are encoded into the model incrementally. When validating those requirements, stakeholders and developers get a better understanding of the system which might lead to new requirements being discovered, or existing requirements evolving or being changed. In the case that a VO fails, the modeler needs to re-consider the VO, the requirement, or even the model. Possible questions that could be asked are:

- Did we translate the requirement into a VO, a VT task or the model poorly?
- Does the requirement collide with other requirements? As a result, we may need to weaken or strengthen this or other requirements.

Another important aspect is requirements engineering, to structure requirements systematically, and to define dependencies between them. For example, there are languages such as KAOS [59], DOORS [45], or Problem Frames [48]. Furthermore, there is also ProR which defines an approach to structure requirements [50]. Concerning refinement and traceability, it will also be important to define dependencies between requirements and thus also VOs. Therefore, we will also take those aforementioned works into account.

5 Demonstration of Validation Obligations

In this section, we will demonstrate how VOs can be used to validate requirements. Let us consider a small traffic light example, modeling the cars' traffic light and the pedestrians' traffic light at a crossing in Germany, with the following requirements:

FUN1: There are two traffic lights: the cars' traffic light and the pedestrians' traffic light. Initially, both traffic lights are red.

FUN2: Cars' traffic light can switch to red and yellow, if it is red and the pedestrians' traffic light is red.

FUN3: Cars' traffic light can switch to green, if it is red and yellow and the pedestrians' traffic light is red.

FUN4: Cars' traffic light can switch to yellow, if it is green and the pedestrians' traffic light is red.

FUN5: Cars' traffic light can switch to red, if it is yellow and the pedestrians' traffic light is red.

FUN6: Pedestrians' traffic light can switch to green, if it is red and the cars' traffic light is red.

FUN7: Pedestrians' traffic light can switch to red, if it is green and the cars' traffic light is red.

SAF1: One of both traffic lights is red at any moment.

SAF2: Cars' traffic light can either be red, red and yellow, yellow, or green.

SAF3: Pedestrians' traffic light can either be red, or green.

LIV1: The situation that both traffic lights are red occurs infinitely often.

SCENARIO1: Running Cycle for Cars' Traffic Light:
In the beginning, the cars' and the pedestrians' traffic light are both red.
The cars' traffic light then switches from red to red and yellow.
Afterwards, it switches from red and yellow to green.
Now, it switches back to yellow, and then to red.
The pedestrians' traffic light stays red during the scenario.

SCENARIO2: Running Cycle for Pedestrians' Traffic Light:
In the beginning, the cars' and the pedestrians' traffic light are both red.
The pedestrians' traffic light switches from red to green.
Afterwards, it switches back from green to red.
The cars' traffic light stays red during the scenario.

Furthermore, this report also considers additional functional requirements **FUN8**, **FUN9**, and **FUN10** in a refinement. These requirements will not be validated in this report.

FUN8: A controller sending a command to switch a traffic light to a specific color, if there are no other commands queued.

FUN9: A traffic light can only switch its color if there is a corresponding command queued.

FUN10: A command can be rejected after it has been sent by the controller.

Encoding those functional requirements leads to the B model described in Listing 1.

```
MACHINE TrafficLight
SETS colors = {red, redyellow, yellow, green}
VARIABLES tl_cars, tl_peds

INVARIANT tl_cars : colors & tl_peds : {red, green} &
             (tl_peds = red or tl_cars = red)

INITIALISATION  tl_cars := red || tl_peds := red

OPERATIONS
cars_ry = SELECT tl_cars = red & tl_peds = red THEN tl_cars := redyellow END;
cars_y = SELECT tl_cars = green THEN tl_cars := yellow END;
cars_g = SELECT tl_cars = redyellow THEN tl_cars := green END;
cars_r = SELECT tl_cars = yellow THEN tl_cars := red END;
peds_r = SELECT tl_peds = green THEN tl_peds := red END;
peds_g = SELECT tl_peds = red & tl_cars = red THEN tl_peds := green END

END
```

Listing 1: Traffic Light Example

To demonstrate state space projection, Listing 1 will be refined. Regarding Classical B, it is not only necessary to add new events in the refinement, but also to add them in the abstract machine refining `skip`. The resulting machines are shown in Listing 4 and Listing 5.

After encoding commands to switch the traffic lights' colors (**FUN8** - **FUN10**), a domain expert might be interested in sending commands without considering the traffic light's color only. Here, the domain expert could define a diagram

describing how the logic for sending commands has to work. This is shown in **PRC1** in Figure 2.

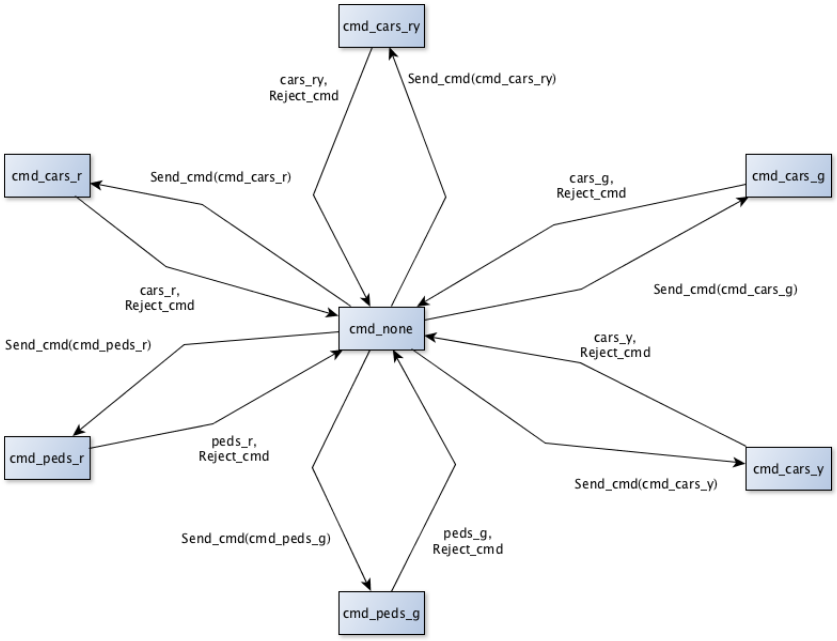

Figure 2: **PRC1** as Diagram

Based on the model, the designer could also run different simulations.

```
{
 "activations": [
  {"id":"$initialise_machine", "execute":"$initialise_machine",
   "activating":"choose"},
  {"id":"choose", "chooseActivation":{"cars_ry": "0.5", "peds_g": "0.5"}},
  {"id":"cars_ry", "execute":"cars_ry", "after":5000, "activating":"cars_g"},
  {"id":"cars_g", "execute":"cars_g", "after":500, "activating":"cars_y"},
  {"id":"cars_y", "execute":"cars_y", "after":5000, "activating":"cars_r"},
  {"id":"cars_r", "execute":"cars_r", "after":500, "activating":"choose"},
  {"id":"peds_g", "execute":"peds_g", "after":5000, "activating":"peds_r"},
  {"id":"peds_r", "execute":"peds_r", "after":5000, "activating":"choose"}
 ]
}
```

Listing 2: Traffic Light Simulation (TrafficLight_Sim)

Listing 2 shows a SimB file [99] that annotates operations with times and probabilities. Within the first simulation shown in Listing 2, the controller chooses between the cars' traffic light's cycle and the pedestrians' traffic light's cycle with a probability of 50% for each. Whenever a traffic light turns green

or red, it will not switch the color for 5 seconds. Switching the cars' traffic light from red and yellow to green, and yellow to red always takes 500 ms.

Based on this simulation, the modeler could then validate the probabilistic timing requirements **PROB-TIM1** and **PROB-TIM2**.

PROB-TIM1: Whenever both traffic lights are red, the cars' traffic light will turn green with a probability of at least 80% within the next 30 seconds.

PROB-TIM2: Whenever both traffic lights are red, the pedestrians' traffic light will turn green with a probability of at least 90% within the next 30 seconds.

Based on the encoded model, the non-functional requirements are as follows:

After validating **SCENARIO1** and **SCENARIO2**, the following coverage criteria are expected to hold: **COV1**, **COV2**, and **COV3**.

COV1: Validating **SCENARIO1** and **SCENARIO2** covers the model such that the cars' traffic light switches between four colors, while the pedestrians' traffic light switches between two colors.

COV2: Validating **SCENARIO1** and **SCENARIO2** covers all operations in the model.

COV3: Validating **SCENARIO1** and **SCENARIO2** covers the whole state space consisting of six possible states (including root) and seven possible transitions.

STRUC1: tl_cars is written by cars_ry, cars_y, cars_g, cars_g only.

STRUC2: tl_peds is written by peds_r and peds_g only.

STRUC3: There are no vacuous parts in the invariant and guards of the model.

STRUC4: The operations enable each other as follows: cars_ry enables cars_g, cars_g enables cars_y, cars_y enables cars_r, cars_r enables cars_ry and peds_g, peds_g enables peds_r, peds_r enables peds_g and cars_ry.

STRUC5: All operations are coverable within 5 steps.

STRUC6: MC/DC coverage with level 2 and depth 5 should be feasible for the model.

Now, we will describe how all these requirements are validated by VOs. Particularly, we will present at least one VO for each requirement. Since the requirements described above do not necessarily have to be validated by VTs from all types, we will also present alternative VTs to demonstrate all VT types. Here, we will mainly focus on validation in ProB. Furthermore, the VOs are formalized using operators in the B method. Regarding probabilistic model checking, we will also take an example in PRISM into account.

In order to validate **FUN1**, it is necessary to check whether both traffic lights are red in all initial states. Thus, this requirement could be validated by a VO validating an LTL model check to expect a positive result, as shown in **VO1**.

LTL1/TrafficLight/LTL: {tl_cars = red & tl_peds = red}, SUCCESS

VO1: validate(LTL1)

For validation of **FUN2**, one needs to check that whenever the cars' traffic light is red and yellow, it has been red and yellow since both traffic lights are red, one step ago. Thus, this behavior can be validated by a VO applying an LTL model check to expect a positive result as shown in **VO2**.

LTL2/TrafficLight/LTL: G ({tl_cars=redyellow} $\implies$ ({tl_cars=redyellow} S {tl_cars=red & tl_peds=red})), SUCCESS

VO2: validate(LTL2)

For validation of **FUN3**, one needs to check that whenever the cars' traffic light is green, it has been green since the cars' traffic light is red and yellow, and the pedestrians' traffic light is red, one step ago. Thus, this behavior can be validated by **VO3** which applies an LTL model check **LTL3** to expect a positive result.

LTL3/TrafficLight/LTL: G ({tl_cars=green} $\implies$ ({tl_cars = green} S {tl_cars=redyellow & tl_peds=red})), SUCCESS

VO3: validate(LTL3)

For validation of **FUN4**, one needs to check that whenever the cars' traffic light is yellow, it has been yellow since the cars' traffic light is green, and the pedestrians' traffic light is red, one step ago. This behavior is also validated by a VO validating an LTL model check expecting a positive result.

LTL4/TrafficLight/LTL: G ({tl_cars=yellow} $\implies$ ({tl_cars=yellow} S {tl_cars=green & tl_peds=red})), SUCCESS

VO4: validate(LTL4)

For validation of **FUN5**, one needs to check two behaviors:

- The cars' traffic light might change its color unequal red (realized by **LTL5.1.**).
- Assuming that the cars' traffic light has already switched its color unequal to red: Whenever the cars' traffic light is red, it has been red since the cars' traffic light is yellow, and the pedestrians' traffic light is red, one step ago (realized by **LTL5.2.**).

While the first property is expected to fail, the second property is expected to hold. Regarding the first behavior, it would also be possible to apply explicit-state model checking searching for a goal. As both tasks are independent of each other, they are validated independently.

LTL5.1/TrafficLight/LTL: $\neg$ (F{tl_cars $\neq$ red}), FAIL

LTL5.2/TrafficLight/LTL: ({tl_cars = red} W ({tl_cars $\neq$ red} & G({tl_cars=red} $\implies$ ({tl_cars=red} S {tl_cars=yellow & tl_peds=red})))), SUCCESS

VO5: validate(LTL5.1) & validate(LTL5.2)

For validation of **FUN6**, one needs to check that whenever the pedestrians' traffic light is green, it has been green since the cars' traffic light is red, and the pedestrians' traffic light is red, one step ago. This behavior is validated by **VO6** applying an LTL model check **LTL6** expecting a positive result.

> **LTL6**/TrafficLight/LTL: G ({tl_peds=green} $\implies$ ({tl_peds=green} S {tl_cars=red & tl_peds=red})), SUCCESS

> **VO6**: validate(LTL6)

For validation of **FUN7**, one needs to check two behaviors:

- The pedestrians' traffic light might change its color unequal red.
- Assuming that the pedestrians' traffic light has already switched its color unequal to red: Whenever the pedestrians' traffic light is red, it has been red since the cars' traffic light is red, and the pedestrians' traffic light is green, one step ago.

Both behaviors can be formulated as an LTL model check, too. While the first property is expected to fail, the second property is expected to hold. Regarding the first behavior, it would also be possible to apply explicit-state model checking searching for a goal. Both validation tasks are independent of each other. Thus, they are composed in parallel before validation as illustrated in **VO7**.

> **LTL7.1**/TrafficLight/LTL: $\neg$ (F{tl_peds $\neq$ red}), FAIL

> **LTL7.2**/TrafficLight/LTL: ({tl_peds = red} W ({tl_peds $\neq$ red} & G({tl_peds=red} $\implies$ ({tl_peds=red} S {tl_cars=red & tl_peds=green})))), SUCCESS

> **VO7**: validate(LTL7.1 || LTL7.2)

The properties for **SAF1** - **SAF3** can be encoded as invariants. Thus, they can be validated by an explicit-state model check, an LTL model check, or a symbolic model check. In the following, we will validate them by VOs (**VO8** - **VO10**) applying respective explicit-state model checks (**MC1** - **MC3**).

Validation of **SAF1**:

MC1/TrafficLight/MC: <INV, tl_cars = red or tl_peds = red>

VO8: validate(MC1)

Validation of **SAF2**:

MC2/TrafficLight/MC: <INV, tl_cars ∈ {red, redyellow, yellow, green}>

VO9: validate(MC2)

Validation of **SAF3**:

MC3/TrafficLight/MC: <INV, tl_peds ∈ {red, green}>

VO10: validate(MC3)

In contrast, **LIV1** is a requirement describing a liveness property. Thus, it can be checked by **VO11** validating an LTL model check to expect a positive result as formalized in **LTL8**.

LTL8/TrafficLight/LTL: GF({tl_cars = red & tl_peds = red}), SUCCESS

VO11: validate(LTL8)

For the validation of **SCENARIO1** and **SCENARIO2**, one needs (1) to replay them by executing the corresponding events, and (2) to check the desired behavior afterwards. To generate those scenarios, the modeler could animate the model, encode postconditions, and store the trace for replay afterwards. This is realized in **TR1** and **TR2** respectively. Afterwards, one could then define two respective VOs validating both VTs as shown in **VO12** and **VO13**.

TR1/TrafficLight/TR: [INITIALISATION <tl_cars = red & tl_peds = red>, cars_ry <tl_cars = redyellow & tl_peds = red>, cars_g <tl_cars = green & tl_peds = red}, cars_y <tl_cars = yellow & tl_peds = red>, cars_r <tl_cars = red & tl_peds = red>]

VO12: validate(TR1)

TR2/TrafficLight/TR: [INITIALISATION <tl_peds = red & tl_cars = red>, peds_g <tl_peds = green & tl_cars = red>, peds_r <tl_peds = red & tl_cars = red>]

VO13: validate(TR2)

As mentioned before, a domain expert could be interested in sending commands only, without taking the traffic lights' colors into account. The diagram portrayed in **PRC1** (Figure 2) could then be validated by projecting the state space on `queuedCmd` for inspection (formalized in **SPRJ1**) after applying explicit-state model checking to cover the complete state space (realized in **MC4**). Both tasks are composed in **VO14** sequentially as **SPRJ1** depends on **MC4**.

MC4/TrafficLight/MC: <FIN>

SPRJ1/TrafficLight_Ref/SPRJ: queuedCmd, $S_{queuedCmd}$ = {<undefined>, cmd_none, cmd_cars_ry, cmd_cars_y, cmd_cars_g, cmd_cars_r, cmd_peds_r, cmd_peds_g} &
$T_{queuedCmd}$ = {(<undefined>, INITIALISATION, cmd_none)} ∪
{cmd_none ↦ Send_cmd(cars_r) ↦ cmd_cars_r,
cmd_none ↦ Send_cmd(cars_ry) ↦ cmd_cars_ry,
cmd_none ↦ Send_cmd(cars_g) ↦ cmd_cars_g,
cmd_none ↦ Send_cmd(cars_y) ↦ cmd_cars_y,
cmd_none ↦ Send_cmd(peds_g) ↦ cmd_peds_g,
cmd_none ↦ Send_cmd(peds_r) ↦ cmd_peds_r} ∪
{cmd_cars_r ↦ Reject_cmd ↦ cmd_none,
cmd_cars_ry ↦ Reject_cmd ↦ cmd_none,
cmd_cars_g ↦ Reject_cmd ↦ cmd_none,
cmd_cars_y ↦ Reject_cmd ↦ cmd_none,
cmd_peds_g ↦ Reject_cmd ↦ cmd_none,
cmd_peds_r ↦ Reject_cmd ↦ cmd_none} ∪
{cmd_cars_r ↦ cars_r ↦ cmd_none, cmd_cars_ry ↦ cars_ry ↦ cmd_none,
cmd_cars_g ↦ cars_g ↦ cmd_none, cmd_cars_y ↦ cars_y ↦ cmd_none,
cmd_peds_g ↦ peds_g ↦ cmd_none, cmd_peds_r ↦ peds_r ↦ cmd_none}

VO14: validate(MC4;SPRJ1)

When validating **PROB-TIM1**, the modeler could apply hypothesis testing, estimation of probability, or inspect the simulation statistics. Here, we will demonstrate the validation of this requirement by applying hypothesis testing (see **HT1**). The configuration to define a hypothesis depends on the tool. In the context of ProB, in particular SimB, one needs to define the starting condition (here a predicate stating that both traffic lights are red), the ending condition (here a time of 30 seconds), the property to be checked (here a predicate checking that the cars' traffic light eventually turns green), the kind of the hypothesis test (here left-tailed), and the desired probability (here 80 %). Furthermore, the modeler also has to provide the number of simulations (here 1 000 000), and the significance level (here 1%). Validating **PROB-TIM2** is done similarly to **PROB-TIM1** with the main difference that the property to be checked states that the pedestrians' traffic light is green in the final state instead of the cars' traffic light (realized by **HT2**). The respective VOs validating both VTs are shown in **VO15** and **VO16**.

HT1/TrafficLight, TrafficLight_Sim/HT: 1000000, (<PRED, tl_cars = red & tl_peds = red>, <TIME, 30000>, <EVENTUALLY, tl_cars = green>, LEFT_TAILED, 0.8), 0.01

VO15: validate(HT1)

HT2/TrafficLight, TrafficLight_Sim/HT: 1000000, (<PRED, tl_cars = red & tl_peds = red>, <TIME, 30000>, <EVENTUALLY, tl_peds = green>, LEFT_TAILED, 0.8), 0.01

VO16: validate(HT2)

In the following, we are going to demonstrate the validation of non-functional requirements.

As described before **COV1**, **COV2**, and **COV3** are coverage criteria for the validation of **SCENARIO1** and **SCENARIO2**. In the following, we assume that **SCENARIO1** and **SCENARIO2** are already validated by other VOs, e.g., **TR1** and **TR2** before.

In order to validate **COV1**, the modeler needs to inspect the variable coverage table after running **TR1** and **TR2** which validates **SCENARIO1** and **SCENARIO2** respectively. Here, it is necessary to check whether the values for `tl_cars` and `tl_peds` are equal to 4 and 2 respectively (realized in **VCT1**).

VCT1/TrafficLight/VCT: $R_{vct}(\text{tl\_cars}) = 4$ & $R_{vct}(\text{tl\_peds}) = 2$

VO17: validate((TR1 || TR2); VCT1)

Similar to the validation of **COV1**, the modeler must also run **TR1** and **TR2** validating **SCENARIO1** and **SCENARIO2** before validating **COV2**. It is then necessary to inspect the operation coverage table manually, to check whether all events have been covered (realized by **OCT1**).

OCT1/TrafficLight/OCT:
{(cars_ry, COVERED), (cars_r, COVERED), (cars_y, COVERED),
(cars_r, COVERED), (peds_r, COVERED), (peds_g, COVERED)} = R_{oct}

VO18: validate((TR1 || TR2); OCT1)

COV3 describes the desired statistics for the number of states and transitions after running **TR1** and **TR2**, validating **SCENARIO1** and **SCENARIO2**. As **SCENARIO1** and **SCENARIO2** should also cover the whole state space, the statistics are also expected to be equal to the statistics when applying explicit-state model checking. To check this coverage criterion, the modeler has to run **STAT1** after validating **SCENARIO1** and **SCENARIO2**. Furthermore, **STAT1** has to be checked after running explicit-state model checking to cover the whole state space as shown in **MC4**.

STAT1/TrafficLight/STAT: R_{spstat}("Number of States") = 6 & R_{spstat}("Number of Transitions") = 7

VO19: validate((TR1 || TR2); STAT1) & validate(MC4;STAT1)

The requirements **STRUC1** and **STRUC2** desire `tl_cars` and `tl_peds` to be written by certain events. This can be validated by the respective VOs **VO20** and **VO21**, validating the respective tasks **RWM1** and **RWM2**. Those VTs has to be checked by inspecting the read/write matrix manually.

RWM1/TrafficLight/RWM: R_{rwm}[{WRITE}]~[{tl_cars}] = {cars_ry, cars_g, cars_y, cars_r}

VO20: validate(RWM1)

RWM2/TrafficLight/RWM: R_{rwm}[{WRITE}]∼[{tl_peds}] = {peds_g, peds_r}

VO21: validate(RWM2)

Again, ensuring that there are no vacuous parts in the invariant and guards of the Traffic Light model (**STRUC3**) can be checked by the VTs shown in **VAP1** and **VAP2**. The corresponding VO applying both tasks is realized in **VO22**.

VAP1/TrafficLight/VAP: INV

VAP2/TrafficLight/VAP: GRD

VO22: validate(VAP1) & validate(VAP2)

STRUC4 describes how events enable each other. This can be translated to task **ED1**. The validation of **ED1** is done by **VO23**.

ED1/TrafficLight/ED:
{(cars_ry, cars_g), (cars_g , cars_y), (cars_y , cars_r), (cars_r , cars_ry), (cars_r, peds_g), (peds_g , peds_r), (peds_r , peds_g), (peds_r, cars_ry)} = R_{ed}.

VO23: validate(ED1)

For the validation of **STRUC5** and **STRUC6**, one could apply test case generation. In order to validate **STRUC5**, test case generation covering all operations with depth 5 could be applied (realized by **OC1**). Again, MCDC coverage test case generation with depth 5 is suitable to validate **STRUC6** (see **MCDC1**). Afterwards, **STRUC5** and **STRUC6** are validated by **VO24** and **VO25** respectively.

OC1/TrafficLight/OC:[cars_r, cars_ry, cars_g, cars_r, peds_g, peds_r], 5

VO24: validate(OC1)

MCDC1/TrafficLight/MCDC:2,5

VO25: validate(MCDC1)

Other VOs to validate requirements Using the previous VOs, all requirements for the traffic light model have already been covered. In the following, we will now demonstrate VTs that have not been used yet. Some VTs will be demonstrated on existing requirements of the Traffic Light example. In contrast, there will also be VTs that will be demonstrated on new requirements, or even other models.

Instead of validating **SAF1** by checking **MC1**, it would also be possible to apply symbolic model checking. The corresponding VO, applying symbolic model checking (see **SMC1**) is shown in **VO26**.

SMC1/TrafficLight/SMC: <INV, tl_cars = red or tl_peds = red>

VO26: validate(SMC1)

Another possibility to validate **SAF1** could be done by proving multiple proof obligations. Here, it would be necessary to generate a PO for the initialization, and for each event checking whether it preserves the invariant describing **SAF1**. As result, this would lead to seven POs (denoted as **PO1** - **PO7**) being generated, one for each operation, to validate **SAF1**. In order to fully validate **SAF1**, it is thus necessary to prove all POs as realized in **VO27**. **PO1** shows the proof obligation (also used as validation task) for invariant preservation of the property describing **SAF1** from the event `cars_ry`. Note that proving POs might need some human interaction.

PO1/TrafficLight/PO: tl_cars ∈ colors, tl_peds ∈ {red, green}, tl_peds = red or tl_cars = red, tl_cars = red, tl_peds = red, tl_cars' = redyellow, tl_peds' = tl_peds ⊨ tl_peds' = red or tl_cars' = red

VO27: validate(PO1) & validate(PO2) ... validate(PO7)

As an alternative to **STAT1**, one could also inspect the state space visualization (realized by **SVIS1**) after running **TR1** and **TR2**. The result could then be checked against the coverage after applying **MC4**. Both are realized in **VO28**. As the state space can grow very fast, it is often better in practice to inspect the state space statistics after checking **MC4** as realized by **STAT1**.

SVIS1/TrafficLight/SVIS: $card(Z_{svis}) = 6$ & $card(T_{svis}) = 7$

VO28: validate((TR1 || TR2); SVIS1) & validate(MC4; SVIS1)

As an alternative to **LTL5.1** and **LTL7.1.**, it would also be possible to apply CTL model checking to expect a positive result. The VOs applying the respective tasks **CTL1** and **CTL2** are shown in **VO29** and **VO30**.

CTL1/TrafficLight/CTL: EF{tl_cars $\neq$ red}, SUCCESS

VO29: validate(CTL1)

CTL2/TrafficLight/CTL: EF{tl_peds $\neq$ red}, SUCCESS

VO30: validate(CTL2)

Instead of applying hypothesis testing, it would also be possible to validate **PROB-TIM1** by estimating the probability. The configuration for the check depends on the tool. Similar to hypothesis testing, the parameters contain the number of simulations, the starting condition, the ending condition, the property to be checked, the kind of estimation checking, and the desired probability. The only difference is the δ value which is used instead of the α value. The corresponding VO is shown in **VO31**, validating the corresponding task **EOP1**.

EOP1/TrafficLight, TrafficLight_Sim/EOP: 1000000, (<PRED, tl_cars = red & tl_peds = red>, <TIME, 30000>, <EVENTUALLY, tl_cars = green>, LEFT_TAILED, 0.8), 0.01

VO31: validate(EOP1)

Now, we will introduce a new requirement to demonstrate the VO for probabilistic/statistical model checking:

> **PROB1:** Whenever both traffic lights are red, the pedestrians' traffic light will turn green with a probability of 50% next.

In order to apply probabilistic/statistical model checking, the modeler has to encode a markov chain as well. So, the demonstration of the corresponding VO is the only one that is not demonstrated using the B method and ProB. An encoding of the Traffic Light model in PRISM is shown in Listing 3. This is also the context for the VT **PSMC1**. Here, the probability to choose between the cars' cycle and the pedestrians' cycle is defined as 50% for each.

```
mdp

module TrafficLight_PRISM

    tl_cars : [0..3] init 0;
    tl_peds : [0..3] init 0;


    [] tl_cars=0 & tl_peds = 0 -> 0.5:(tl_cars'=1) + 0.5:(tl_peds'=2);
    [] tl_cars=1 -> (tl_cars' = 2);
    [] tl_cars=2 -> (tl_cars' = 3);
    [] tl_cars=3 -> (tl_cars' = 0);
    [] tl_peds=2 -> (tl_peds' = 0);

endmodule
```

Listing 3: Traffic Light in PRISM

Validating **PROB1** is then done by checking **VO32**, validating **PSMC1** with PCTL formula.

> **PSMC1**/TrafficLight_PRISM/PSMC: P>0.9999[¬(true U (¬((tl_cars = 0 & tl_peds = 0) $\implies$ (P>0.49 [X (tl_peds = 2)] & P<0.51 [X (tl_peds = 2)])))], SUCCESS

> **VO32**: validate(PSMC1)

PROB1 could also be validated by inspecting the simulation statistics (realized in **SISTAT1**). The corresponding VO is shown in **VO33**.

> **SISTAT1**/TrafficLight, TrafficLight_Sim/SISTAT: 10000, <PRED, 1=1>, <STEPS, 100>, R_{sistat}(enabled $\mapsto$ peds_g)/R_{sistat}(executed $\mapsto$ peds_g) $\in$ [0.49, 0.51]

VO33: validate(SISTAT1)

Figure 3 shows a taxonomy of a subset of requirements (**FUN1**, **FUN5**, **SCENARIO1**, **SCENARIO2**, **COV1**, **PRC1**), as well as VOs, and VTs validating them. Based on this taxonomy, we will demonstrate how possible error sources could be traced when a VO fails. For example, assuming that **VO1** has failed, the possible error sources could be **LTL1**, the requirement **FUN1**, or the model. Similarly, **VO5** is traced to **LTL5.1**, **LTL5.2.**, **FUN5**, or the model. Again, **VO14** is traced to **SPRJ1**, **MC4**, **PRC1**, or the model. While **VO12** and **VO13** are traced to **TR1** and **TR2**, and thus also **SCENARIO1** and **SCENARIO2** respectively, tracing VOs to VTs and requirements is more complicated for **VO18**. Here, it is necessary to determine which parts of the VO has failed. In the case that only $\mathbf{OCT_1}$ fails, it means that the set of scenarios might be incomplete. In contrast, **TR3** and **TR4** are traced via **VO12** and **VO13** respectively. This means that **VO18** could have failed because of the failure of **VO12** or **VO13**.

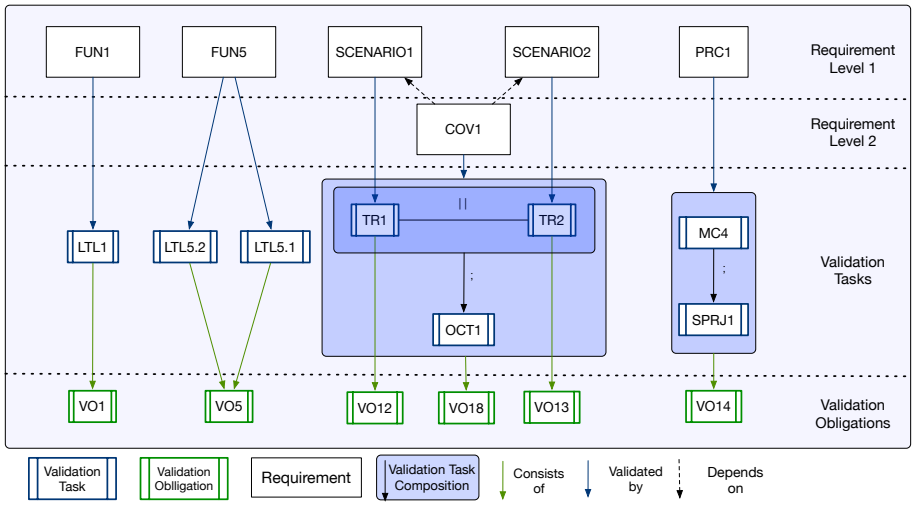

Figure 3: Taxonomy with Requirements FUN1, FUN5, SCENARIO1, SCENARIO2, COV1, PRC1, and corresponding VTs, and VOs validating them

As the Traffic Light model contains variables from the type `colors` only, it is not possible to inspect minimum and maximum values. Let us consider a lift moving between the ground level and the 100th level. Furthermore, assume that the level is modeled by a variable `level`. Consider a scenario shown in **SCENARIO-LIFT**.

SCENARIO-LIFT:/In the beginning, the lift is located at the ground level. It then moves floor by floor until it reaches the third level.

After validating **SCENARIO-LIFT**, i.e., after re-playing the scenario realized by a task **TR-LIFT**, it is expected that the lift has moved between the ground floor and the third level. The corresponding requirement is shown in **COV-LIFT**.

COV-LIFT:/After validating **SCENARIO-LIFT**, it is expected that the lift has moved between the ground floor and the third level.

COV-LIFT could then be validated by **VO-LIFT**, which inspects the minimum and maximum value as shown in **MMV-LIFT** after running **TR-LIFT**.

MMV-LIFT/Lift/MMV: $\min(R_{mmv}(\text{level})) = 0$ & $\max(R_{mmv}(\text{level})) = 3$

VO-LIFT: validate(TR-LIFT; MMV-LIFT)

Finally, we will also show a VO, applying model checking to search for a goal, which is used as an initial state to run trace replay. This will be demonstrated in an automotive case study [64]. Consider the scenario shown in **SCENARIO-AUTO**.

SCENARIO-AUTO:
In this scenario, it is assumed that the engine is turned on, and the blinker is in position `Downward7`.
After 500 ms, the lights on the left-hand side turn on with an intensity of 100.
When passing another 500ms, the lights on the left-hand side turn off.
Both events are repeated in the same order one more time.
While the lights on the left-hand side are blinking, those on the right-hand side are always turned off.

First, the assumption of the scenario (engine turned on, and blinker in position `Downward7`) is validated by finding a state from which the other events of the scenario are executed. This is realized by the explicit-state model check **MC-AUTO**. Outgoing from the state that should be found, the rest of the scenario is then validated via trace replay which is realized by **TR-AUTO**.

MC-AUTO/PitmanController_Time_MC_v4/MC:
<GOAL, engineOn = TRUE & pitmanArmUpDown = Downward7>

TR-AUTO/PitmanController_Time_MC_v4/TR:
[RTIME_BlinkerOn(delta=500) <blinkLeft = 100, blinkRight=0>,
RTIME_BlinkerOff(delta=500) <blinkLeft = 0, blinkRight=0>,
RTIME_BlinkerOn(delta=500) <blinkLeft = 100, blinkRight=0>,
RTIME_BlinkerOff(delta=500) <blinkLeft = 0, blinkRight=0>]

VO-AUTO: validate(MC-AUTO; TR-AUTO)

A Changes History

Version 1.0.0: Initial Version

Version 1.1.0:

- Introduction updated
- Classification of Requirements - now Section 2 instead of Section 4
- "Overlap between Validation and Verification" renamed to "Verification and Validation"
- "Verification and Validation" updated
- "Comparison with Proof Obligations" (Section 2.1) removed - now discussed in Section 1
- Validation Obligations Approach - now Section 4 instead of Section 2
- Section 2.4. ("Refinement") and Section 2.5. ("Refactoring") - merged to "Refinement and Refactoring" and moved to Section 4.1.
- Section 5 ("Validation Techniques and Validation Obligations") - now moved to Section 4.2. and renamed to "Validation Techniques and Tasks"
- Separate *validation obligation* and *validation task*
- Introduce definition for *validation task* in "Validation Techniques and Tasks"
- Introduce parallel (||) and sequential (;) composition for validation task
- Introduce Table 1 - showing overview of all validation tasks
- Remove mathematical formulations of validation tasks - now described in Table 1
- Remove Trace Refinement as validation task (originally Section 5.2.)

- Minor changes in Section 4.2. (originally Section 5.)
- Vacuous Guards/Invariants - not part of "Validation by Visualization, Statistics, and Metrics" anymore
- Introduce Section 4.3. ("Definition of Validation Obligation")
- Formal Definition of Validation Obligation updated
- Minor changes in Formulation of "Definition of Validation Obligation"
- Section 2.3 "Creating Validation Tasks" - now renamed to "Creating Validation Obligations" (Section 4.4.)
- Section 2.2. "Development" - now renamed to "VO-guided Workflow" and moved to Section 4.5.
- Remove Figure 2 in "VO-guided Workflow" (originally Section 2.2.)
- Section 6 "Demonstration of Validation Obligations" - now Section 5
- Minor changes in formulations in "Demonstration of Validation Obligations"
- Remove **PRC1** in "Demonstration of Validation Obligations"
- Separation between *validation task* and *validation obligation* in "Demonstration of Validation Obligations" following the new definitions in Section 4.2. "Validation Techniques and Tasks" and Section 4.3. "Definition of Validation Obligation"
- Use parallel and sequential composition of *validation task* in VOs in "Demonstration of Validation Obligations"
- Remove **TRF1** validating **PRC1** in "Demonstration of Validation Obligations"
- Add and Explain Figure 3: "Taxonomy with Requirements FUN1, FUN5, SCENARIO1, SCENARIO2, COV1, PRC1, and corresponding VTs, and VOs validating them"
- Rename **MMV1** to **MMV-LIFT**
- Minor changes in formulations in Glossary
- Update *validation task* in Glossary
- Add *validation obligation*, *validation predicate*, and *validation function* in Glossary
- Rename Appendix C "Overview VO Examples" to "Overview VT Examples"
- Rename VO to VT in "Overview VT Examples"

B Glossary

State The state of a (software) system is represented by the values of its variables (and constants).

Operation An operation is a term that is well-known from the formal B method. Analogous terms are, e.g., events or actions. It consists of a guard, several effects, and optionally input and return parameters. A guard is a predicate corresponding to an operation which is true when the operation is enabled. When executing the operation, the effects are applied to the current state, modifying it to the succeeding state.

Transition A transition is labeled with an operation (and its parameters), and defined between two states s_1 and s_2 under the following condition: The operation together with its parameters is enabled in s_1, and executing the operation with the parameters modifies s_1 resulting in s_2.

State Space A state space shows all possible executions of the system. It consists of a set of states and transitions between them.

Trace A trace is a list of transitions describing a path through the state space.

Scenario A scenario is a sequence of events (written in natural language) that describes certain desired behavior patterns. It is formalized as a trace, and can be the result of a simulation, or test case generation.

Differences between Scenario and Traces While researching literature it became apparent that the terms of *trace* and *scenario* have different meanings in the formal methods community. Scenarios have also different meanings depending on the domain and context they are used in [51, 76, 65, 81]. In the referenced paper it is somewhat agreed that a scenario describes a desired behavior. For software development, a scenario is then often expressed in a non-ambiguous DSL like Gherkin [103]. In software engineering, there is the sentiment that traces are a realisation of a scenario, shown for example in [70]. Depending on the underlying formalism a scenario can therefore have multiple traces that satisfy it.

Verification Verification checks whether a model meets its specification. So, here we ask the question: *Are we building the software correctly?*.

Validation Validation checks whether a model meets the stakeholder's requirements. So the main question is: *Are we building the right software?*.

Validation Obligation A *validation obligation* (VO) is composed of (multiple) validation tasks (VT) associated with a model to check its compliance with a particular requirement. Thus, validating a requirement succeeds if all associated VOs yield successful results. Formally, a VO is annotated with an id, and consists of a *validation predicate* (written as: $\mathbf{VO_{id}} : \mathrm{VO}_{\mathrm{predicate}}$).

Validation Predicate The *validation predicate* (VP) consists of logical operations on the associated validation tasks' validation results. It is part of the validation obligation.

Validation Function Let V be the set of validation tasks, and let $\mathbb{B} = \{\mathrm{TRUE}, \mathrm{FALSE}\}$ be the set of booleans. Validating a validation task is described as a function *validate* which executes a validation task in the corresponding context. After executing the validation task, the result is a boolean value describing whether the validation succeeds.

$$\mathrm{validate} : V \to \mathbb{B}$$

Validation Technique A validation technique is a technique to validate a requirement. For example, one could validate a requirement describing a temporal property by LTL model checking. In this case, LTL model checking is the validation technique.

Validation Task A validation task (VT) is identified with an identifier, and consists of a validation technique that is applied with the given validation parameters to the corresponding context. Executing a VT possibly modifies the internal state of the validation tool, e.g., consisting of the currently explored state space, and the current trace. The notation for a VT is as follows: $\mathbf{VT_{id}}/\mathrm{VT}_{\mathrm{context}}/\mathrm{VT}_{\mathrm{technique}}\colon \mathrm{VT}_{\mathrm{parameters}}$

C Traffic Light Refinement

```
MACHINE TrafficLight2
SETS colors = {red, redyellow, yellow, green};
     COMMANDS = {cmd_cars_ry, cmd_cars_y, cmd_cars_g,
                 cmd_cars_r, cmd_peds_r, cmd_peds_g, cmd_none}
VARIABLES tl_cars, tl_peds
INVARIANT tl_cars : colors & tl_peds : {red, green} &
          (tl_peds = red or tl_cars = red)

INITIALISATION  tl_cars := red || tl_peds := red
OPERATIONS
 Send_cmd(cmd) = SELECT cmd : COMMANDS THEN skip END;
 Reject_cmd = skip;

 cars_ry = SELECT tl_cars = red & tl_peds = red THEN tl_cars := redyellow END;
 cars_y = SELECT tl_cars = green THEN tl_cars := yellow END;
 cars_g = SELECT tl_cars = redyellow THEN tl_cars := green END;
 cars_r = SELECT tl_cars = yellow THEN tl_cars := red END;
 peds_r = SELECT tl_peds = green THEN tl_peds := red END;
 peds_g = SELECT tl_cars = red & tl_peds = red THEN tl_peds := green END
END
```

Listing 4: Abstract Traffic Light

```
REFINEMENT TrafficLightCommand_Ref REFINES TrafficLight2
VARIABLES tl_cars, tl_peds, queuedCmd
INVARIANT queuedCmd : COMMANDS
INITIALISATION  tl_cars := red || tl_peds := red || queuedCmd := cmd_none
OPERATIONS
  Send_cmd(cmd) =
    SELECT cmd : COMMANDS & cmd /= cmd_none & queuedCmd = cmd_none
    THEN
      queuedCmd := cmd
    END;
  Reject_cmd =
    SELECT queuedCmd /= cmd_none
    THEN
      queuedCmd := cmd_none
    END;

   cars_ry =
     SELECT
       tl_cars = red & tl_peds = red & queuedCmd = cmd_cars_ry
     THEN
       tl_cars := redyellow ||
       queuedCmd := cmd_none
     END;

   cars_y =
     SELECT
       tl_cars = green & queuedCmd = cmd_cars_y
     THEN
      tl_cars := yellow ||
      queuedCmd := cmd_none
     END;

   cars_g =
     SELECT
       tl_cars = redyellow & queuedCmd = cmd_cars_g
     THEN
       tl_cars := green ||
       queuedCmd := cmd_none
     END;

   cars_r =
```

```
    SELECT
      tl_cars = yellow & queuedCmd = cmd_cars_r
    THEN
      tl_cars := red ||
      queuedCmd := cmd_none
    END;

  peds_r =
    SELECT
      tl_peds = green & queuedCmd = cmd_peds_r
    THEN
      tl_peds := red ||
      queuedCmd := cmd_none
    END;

  peds_g =
     SELECT
       tl_cars = red & tl_peds = red & queuedCmd = cmd_peds_g
     THEN
       tl_peds := green ||
       queuedCmd := cmd_none
     END


END
```

Listing 5: Traffic Light Refinement

D Overview VT Examples

Trace Replay VT:

TR1/TrafficLight/TR: [INITIALISATION <tl_cars = red & tl_peds = red>, cars_ry <tl_cars = redyellow & tl_peds = red>, cars_g <tl_cars = green & tl_peds = red}, cars_y <tl_cars = yellow & tl_peds = red>, cars_r <tl_cars = red & tl_peds = red>]

Operation Coverage Test Case Generation VT:

OC1/TrafficLight/OC:[cars_r, cars_ry, cars_g, cars_r, peds_g, peds_r], 5

MC/DC Coverage Test Case Generation VT:

MCDC1/TrafficLight/MCDC:2,5

Hypothesis Testing VT:

HT1/TrafficLight, TrafficLight_Sim/HT: 1000000, (<PRED, tl_cars = red & tl_peds = red>, <TIME, 30000>, <EVENTUALLY, tl_cars = green>, LEFT_TAILED, 0.8), 0.01

Estimation of Probability VT:

EOP1/TrafficLight, TrafficLight_Sim/EOP: 1000000, (<PRED, tl_cars = red & tl_peds = red>, <TIME, 30000>, <EVENTUALLY, tl_cars = green>, LEFT_TAILED, 0.8), 0.01

Simulation Statistics VT:

SISTAT1/TrafficLight, TrafficLight_Sim/SISTAT: 10000, <PRED, 1=1>, <STEPS, 100>, R_{sistat}(enabled $\mapsto$ cars_ry)/R_{sistat}(executed $\mapsto$ cars_ry) $\in$ [0.49, 0.51]

Explicit-State Model Checking VT:

MC1/TrafficLight/MC: <INV, tl_cars = red or tl_peds = red>

LTL Model Checking VT:

LTL1/TrafficLight/LTL: {tl_cars = red & tl_peds = red}, SUCCESS

CTL Model Checking VT:

CTL1/TrafficLight/CTL: EF{tl_cars $\neq$ red}, SUCCESS

Probabilistic/Statistical Model Checking VT:

PSMC1/TrafficLight_PRISM/PSMC: P>0.9999[¬(true U (¬((tl_cars = 0 & tl_peds = 0) $\implies$ (P>0.49 [X (tl_peds = 2)] & P<0.51 [X (tl_peds = 2)])))], SUCCESS

Symbolic Model Checking VT:

SMC1/TrafficLight/SMC: <INV, tl_cars = red or tl_peds = red>

Proving VT:

PO1/TrafficLight/PO: tl_cars $\in$ colors, tl_peds $\in$ {red, green}, tl_peds = red or tl_cars = red, tl_cars = red, tl_peds = red, tl_cars' = redyellow, tl_peds' = tl_peds $\models$ tl_peds' = red or tl_cars' = red

Variable Coverage Table VT:

VCT1/TrafficLight/VCT: R_{vct}(tl_cars) = 4 & R_{vct}(tl_peds) = 2

Min/Max Values VT:

MMV-LIFT/Lift/MMV: min(R_{mmv}(level)) = 0 & max(R_{mmv}(level)) = 3

Operation Coverage Table VT:

OCT1/TrafficLight/OCT:
{(cars_ry, COVERED), (cars_r, COVERED), (cars_y, COVERED),
(cars_r, COVERED), (peds_r, COVERED), (peds_g, COVERED)} = R_{oct}

Read/Write Matrix VT:

RWM1/TrafficLight/RWM: R_{rwm}[{WRITE}]~[{tl_cars}] = {cars_ry, cars_g, cars_y, cars_r}

Enabling Diagram VT:

ED1/TrafficLight/ED:
{(cars_ry, cars_g), (cars_g , cars_y), (cars_y , cars_r), (cars_r , cars_ry), (cars_r, peds_g), (peds_g , peds_r), (peds_r , peds_g), (peds_r, cars_ry)} = R_{ed}.

Vacuous Parts VT:

VAP1/TrafficLight/VAP: INV

State Space Visualization VT:

SVIS1/TrafficLight/SVIS: $card(Z_{svis}) = 6$ & $card(T_{svis}) = 7$

State Space Projection VT:

SPRJ1/TrafficLight_Ref/SPRJ: queuedCmd, $S_{queuedCmd}$ = {<undefined>, cmd_none, cmd_cars_ry, cmd_cars_y, cmd_cars_g, cmd_cars_r, cmd_peds_r, cmd_peds_g} &
$T_{queuedCmd}$ = {(<undefined>, INITIALISATION, cmd_none)} ∪
{cmd_none ↦ Send_cmd(cars_r) ↦ cmd_cars_r,
cmd_none ↦ Send_cmd(cars_ry) ↦ cmd_cars_ry,
cmd_none ↦ Send_cmd(cars_g) ↦ cmd_cars_g,
cmd_none ↦ Send_cmd(cars_y) ↦ cmd_cars_y,
cmd_none ↦ Send_cmd(peds_g) ↦ cmd_peds_g,
cmd_none ↦ Send_cmd(peds_r) ↦ cmd_peds_r} ∪
{cmd_cars_r ↦ Reject_cmd ↦ cmd_none,
cmd_cars_ry ↦ Reject_cmd ↦ cmd_none,
cmd_cars_g ↦ Reject_cmd ↦ cmd_none,
cmd_cars_y ↦ Reject_cmd ↦ cmd_none,
cmd_peds_g ↦ Reject_cmd ↦ cmd_none,
cmd_peds_r ↦ Reject_cmd ↦ cmd_none} ∪
{cmd_cars_r ↦ cars_r ↦ cmd_none, cmd_cars_ry ↦ cars_ry ↦ cmd_none,
cmd_cars_g ↦ cars_g ↦ cmd_none, cmd_cars_y ↦ cars_y ↦ cmd_none,
cmd_peds_g ↦ peds_g ↦ cmd_none, cmd_peds_r ↦ peds_r ↦ cmd_none}

State Space Statistics VT:

STAT1/TrafficLight/STAT: R_{spstat}("Number of States") = 6 & R_{spstat}("Number of Transitions") = 7

Composed VTs:

MC-AUTO/PitmanController_Time_MC_v4/MC:
<GOAL, engineOn = TRUE & pitmanArmUpDown = Downward7>

TR-AUTO/PitmanController_Time_MC_v4/TR:
[RTIME_BlinkerOn(delta=500) <blinkLeft = 100, blinkRight=0>,
RTIME_BlinkerOff(delta=500) <blinkLeft = 0, blinkRight=0>,
RTIME_BlinkerOn(delta=500) <blinkLeft = 100, blinkRight=0>,
RTIME_BlinkerOff(delta=500) <blinkLeft = 0, blinkRight=0>]

MC-AUTO; TR-AUTO

E Published Papers

In the following, we list the papers that are published in the context of D 1.1. of the IVOIRE project:

- Atif Mashkoor, Michael Leuschel, Alexander Egyed. Validation Obligations: A Novel Approach to Check Compliance between Requirements and their Formal Specification [69]
- Fabian Vu, Michael Leuschel, Atif Mashkoor. Validation of Formal Models by Timed Probabilistic Simulation [99]
- Atif Mashkoor, Alexander Egyed. Evaluating the alignment of sequence diagrams with system behavior [68]
- Jens Bendisposto, David Geleßus, Yumiko Jansing, Michael Leuschel, Antonia Pütz, Fabian Vu, Michelle Werth. ProB2-UI: A Java-Based User Interface for ProB [10] (extended in context of IVOIRE)

List of Figures

List of Tables

Listings

References

[1] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge university press, 2005.

[2] Jean-Raymond Abrial et al. "Rodin: An Open Toolset for Modelling and Reasoning in Event-B". In: *Int. J. Softw. Tools Technol. Transf.* 12.6 (Nov. 2010), pp. 447–466. ISSN: 1433-2779.

[3] Sten Agerholm. "Translating specifications in VDM-SL to PVS". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 1996, pp. 1–16.

[4] Linas Laibinis Anton Tarasyuk Elena Troubitsyna. *Reliability Assessment in Event-B Development*. Linköping electronic Press, 2009.

[5] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. "AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications". In: *Abstract State Machines, Alloy, B and Z*. Ed. by Marc Frappier et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 61–74. ISBN: 978-3-642-11811-1.

[6] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. "SMT-based automatic proof of ASM model refinement". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2016, pp. 253–269.

[7] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[8] Michael Balser et al. "Formal System Development with KIV". In: vol. 1783. Mar. 2000, pp. 363–366. ISBN: 978-3-540-67261-6. DOI: 10.1007/3-540-46428-X_25.

[9] Samuel Lincoln Magalhães Barrocas and Marcel Oliveira. "JCircus 2.0: an Extension of an Automatic Translator from Circus to Java." In: *CPA*. 2012, pp. 15–36.

[10] Jens Bendisposto et al. “ProB2-UI: A Java-based User Interface for ProB”. In: *Proceedings FMICS*. Springer. 2021.

[11] Bendisposto, Jens and Clark, Joy and Dobrikov, Ivaylo and Körner, Philipp and Krings, Sebastian and Ladenberger, Lukas and Leuschel, Michael and Plagge Daniel. “ProB 2.0 Tutorial”. In: *Proceedings of the 4th Rodin User and Developer Workshop*. TUCS Lecture Notes, 2013.

[12] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. “AsmetaA: Animator for Abstract State Machines”. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Ed. by Michael Butler et al. Cham: Springer International Publishing, 2018, pp. 369–373. ISBN: 978-3-319-91271-4.

[13] Silvia Bonfanti et al. “Asm2C++: a tool for code generation from abstract state machines to Arduino”. In: *NASA Formal Methods Symposium*. Springer. 2017, pp. 295–301.

[14] Egon Börger. “The ASM refinement method”. In: *Formal aspects of computing* 15.2 (2003), pp. 237–257.

[15] Achim Brucker, Frank Rittinger, and Burkhart Wolff. “HOL-Z 2.0: A proof environment for Z-specifications”. In: *J. UCS* 9 (Jan. 2003), pp. 152–172.

[16] Julien Brunel et al. “Simulation under arbitrary temporal logic constraints”. In: *arXiv preprint arXiv:1912.10634* (2019).

[17] Julien Brunel et al. “The electrum analyzer: model checking relational first-order temporal specifications”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018, pp. 884–887.

[18] Michael Butler and Michael Leuschel. “Combining CSP and B for Specification and Property Verification”. In: *FM 2005: Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 221–236. ISBN: 978-3-540-31714-2.

[19] Alessandro Carioni et al. “A Scenario-Based Validation Language for ASMs”. In: *Abstract State Machines, B and Z*. Ed. by Egon Börger et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 71–84. ISBN: 978-3-540-87603-8.

[20] J Carter and William B Gardner. “Mise en scene: converting scenarios to CSP traces in support of requirements-based programming”. In: *31st IEEE Software Engineering Workshop (SEW 2007)*. IEEE. 2007, pp. 41–52.

[21] Kaustuv C. Chaudhuri et al. *A TLA+ Proof System*. 2008. arXiv: `0811.1914 [cs.LO]`.

[22] ClearSy. *Atelier B, User and Reference Manuals*. Available at `http://www.atelierb.eu/`. Aix-en-Provence, France, 2016.

[23] Mathieu Comptier et al. “Property-Based Modelling and Validation of a CBTC Zone Controller in Event-B”. In: *Proceedings RSSRail*. 2019, pp. 202–212. DOI: 10.1007/978-3-030-18744-6_13. URL: https://doi.org/10.1007/978-3-030-18744-6%5C_13.

[24] Alcino Cunha and Nuno Macedo. “Validating the Hybrid ERTMS/ETCS Level 3 Concept with Electrum”. In: Jan. 2018, pp. 307–321. ISBN: 978-3-319-91270-7. DOI: 10.1007/978-3-319-91271-4_21.

[25] John Derrick, Siobhán North, and Anthony JH Simons. “Z2SAL-building a model checker for Z”. In: *International Conference on Abstract State Machines, B and Z*. Springer. 2008, pp. 280–293.

[26] John Derrick, Siobhán North, and Anthony JH Simons. “Z2SAL: a translation-based model checker for Z”. In: *Formal Aspects of Computing* 23.1 (2011), pp. 43–71.

[27] J. Desharnais et al. “Integration of sequential scenarios”. In: *IEEE Transactions on Software Engineering* 24.9 (1998), pp. 695–708. DOI: 10.1109/32.713325.

[28] Ivaylo Dobrikov and Michael Leuschel. “Enabling analysis for Event-B”. In: *Science of Computer Programming* (Aug. 2017). DOI: 10.1016/j.scico.2017.08.004.

[29] Georg Droschl. “Design and application of a test case generator for VDM-SL”. In: *Workshop Materials: VDM in Practice*. 1999.

[30] Aaron W. Fifarek et al. “SpeAR v2.0: Formalized Past LTL Specification and Analysis of Requirements”. In: *Proceedings NFM*. LNCS 10227. 2017, pp. 420–426. ISBN: 978-3-319-57288-8.

[31] Ronald A. Fisher. “Theory of Statistical Estimation”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 22.5 (1925), pp. 700–725. DOI: 10.1017/S0305004100009580.

[32] John Fitzgerald et al. “Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems”. In: *Integrated Formal Methods*. Ed. by Dominique Méry and Stephan Merz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

[33] L. Freitas. “Proving Theorems with Z / Eves”. In: 2005.

[34] Leonardo Freitas. “Model checking circus”. PhD thesis. University of York, 2005.

[35] A. Gargantini and E. Riccobene. “ASM-Based Testing: Coverage Criteria and Automatic Test Sequence”. In: *J. Univers. Comput. Sci.* 7 (2001), pp. 1050–1067.

[36] Angelo Gargantini. “Using model checking to generate fault detecting tests”. In: *International Conference on Tests and Proofs*. Springer. 2007, pp. 189–206.

[37] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. "Using Spin to generate tests from ASM specifications". In: *International Workshop on Abstract State Machines*. Springer. 2003, pp. 263–277.

[38] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. "A Metamodel-based Language and a Simulation Engine for Abstract State Machines". In: *Journal of Universal Computer Science* 14 (Jan. 2008), pp. 1949–1983.

[39] Dimitra Giannakopoulou et al. "Generation of Formal Requirements from Structured Natural Language". In: *Proceedings REFSQ*. LNCS 12045. 2020.

[40] Thomas Gibson-Robinson et al. "FDR3–a modern refinement checker for CSP". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2014, pp. 187–201.

[41] Dominik Hansen and Michael Leuschel. "Translating TLA+ to B for validation with ProB". In: *International Conference on Integrated Formal Methods*. Springer. 2012, pp. 24–38.

[42] Dominik Hansen et al. "Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains". In: *Proceedings ABZ*. 2018, pp. 292–306. DOI: 10.1007/978-3-319-91271-4_20. URL: https://doi.org/10.1007/978-3-319-91271-4%5C_20.

[43] Miran Hasanagić et al. "Code generation for distributed embedded systems with VDM-RT". In: *Design Automation for Embedded Systems* 23.3 (2019), pp. 153–177.

[44] Steffen Helke, Thomas Neustupny, and Thomas Santen. "Automating test case generation from Z specifications with Isabelle". In: *International conference of Z users*. Springer. 1997, pp. 52–71.

[45] IBM. *Engineering Requirements Management DOORS Overview*. Available at https://www.ibm.com/docs/en/ermd/9.7.2?topic=engineering-requirements-management-doors-overview. 2021.

[46] "IEEE Standard Glossary of Software Engineering Terminology". In: *ANSI/ IEEE Std 729-1983* (1983), pp. 1–40. DOI: 10.1109/IEEESTD.1983.7435207.

[47] Yoshinao Isobe and Markus Roggenbach. "Proof principles of CSP–CSP-Prover in practice". In: *Dynamics in Logistics*. Springer, 2008, pp. 425–442.

[48] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[49] Jean-Pierre Jacquot and Atif Mashkoor. *The Role of Validation in Refinement-Based Formal Software Development*. 2018.

[50] Michael Jastram. "The ProR Approach: Traceability of Requirements and System Descriptions". PhD thesis. Heinrich-Heine-Universität Düsseldorf, 2012.

[51] Michał Kempka et al. "ViZDoom: A Doom-based AI research platform for visual reinforcement learning". In: *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. 2016, pp. 1–8. DOI: 10.1109/CIG.2016.7860433.

[52] Maurice G. Kendall, Alan Stuart, and J. Keith Ord. "Kendall's Advanced Theory of Statistics". In: *Oxford University Press*. 1987. ISBN: 0195205618.

[53] Igor Konnov, Jure Kukovec, and Thanh Tran. "BmcMT: Bounded Model Checking of TLA+ Specifications with SMT". In: *TLA+ Community Meeting 2018*. 2018.

[54] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. "TLA+ model checking made symbolic". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–30.

[55] Sebastian Krings. "Towards infinite-state symbolic model checking for B and Event-B". PhD thesis. Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2017.

[56] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. "The TLA+ Toolbox". In: *Electronic Proceedings in Theoretical Computer Science* 310 (Dec. 2019), pp. 50–62. ISSN: 2075-2180. DOI: 10.4204/eptcs.310.6. URL: http://dx.doi.org/10.4204/EPTCS.310.6.

[57] Lukas Ladenberger and Michael Leuschel. "Mastering the Visualization of Larger State Spaces with Projection Diagrams". In: *Proceedings ICFEM 2015*. LNCS 9407. 2015, pp. 153–169. DOI: 10.1007/978-3-319-25423-4_10. URL: https://doi.org/10.1007/978-3-319-25423-4%5C%5F10.

[58] Lukas Ladenberger et al. "Validation of the ABZ Landing Gear System Using ProB". In: *International Journal on Software Tools for Technology Transfer* 19.2 (Apr. 2017), pp. 187–203. ISSN: 1433-2779. DOI: 10.1007/s10009-015-0395-9. URL: https://doi.org/10.1007/s10009-015-0395-9.

[59] Axel van Lamsweerde et al. "The KAOS Project: Knowledge Acquisition in Automated Specification of Software". In: *Proceedings AAAI Spring Symposium Series*. 1991, pp. 59–62.

[60] Axel Legay, Benoît Delahaye, and Saddek Bensalem. "Statistical Model Checking: An Overview". In: *Runtime Verification*. Vol. 6418. LNCS. 2010.

[61] Axel Legay et al. "Statistical Model Checking". In: *Computing and Software Science: State of the Art and Perspectives*. Vol. 10000. LNCS. Cham: Springer International Publishing, 2019, pp. 478–504. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_23. URL: https://doi.org/10.1007/978-3-319-91908-9%5C%5F23.

[62] Michael Leuschel and Michael Butler. "ProB: A model checker for B". In: *International symposium of formal methods europe*. Springer. 2003, pp. 855–874.

[63] Michael Leuschel, Thierry Massart, and Andrew Currie. "How to make FDR spin: LTL model checking of CSP by refinement". In: vol. 2021. Jan. 2001, pp. 99–118.

[64] Michael Leuschel, Mareike Mutz, and Michelle Werth. "Modelling and Validating an Automotive System in Classical B and Event-B". In: *Proceedings ABZ*. LNCS. 2020, pp. 335–350.

[65] Clayton Lewis and John Rieman. "Task-centered user interface design". In: *A practical introduction* (1993).

[66] Hsin-Hung Lin et al. "Towards verifying VDM using SPIN". In: *International Workshop on Formal Techniques for Safety-Critical Systems*. Springer. 2015, pp. 241–256.

[67] Savi Maharaj and Juan Bicarregui. "On the verification of VDM specification and refinement with PVS". In: *Proceedings 12th IEEE International Conference Automated Software Engineering*. IEEE. 1997, pp. 280–289.

[68] Atif Mashkoor and Alexander Egyed. "Evaluating the alignment of sequence diagrams with system behavior". In: *Procedia Computer Science* 180 (Jan. 2021), pp. 502–506. DOI: 10.1016/j.procs.2021.01.267.

[69] Atif Mashkoor, Michael Leuschel, and Alexander Egyed. *Validation Obligations: A Novel Approach to Check Compliance between Requirements and their Formal Specification*. 2021. arXiv: 2102.06037 [cs.SE].

[70] Nabor C. Mendonca et al. "Detecting Implied Scenarios from Execution Traces". In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. 2007, pp. 50–59. DOI: 10.1109/WCRE.2007.19.

[71] David Mentré et al. "Discharging proof obligations from Atelier B using multiple automated provers". In: *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*. Springer. 2012, pp. 238–251.

[72] Christopher Z. Mooney. "Monte Carlo Simulation". In: *Sage publications*. Vol. 116. 1997.

[73] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. "Guided Test Generation from CSP Models". In: *Theoretical Aspects of Computing - ICTAC 2008*. Ed. by John S. Fitzgerald, Anne E. Haxthausen, and Husnu Yenigun. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[74] Tomohiro Oda et al. "VDM animation for a wider range of stakeholders". In: *Proceedings of the 13th Overture Workshop*. Citeseer. 2015, pp. 18–32.

[75] *On proving alloy specifications using KeY*.

[76] Ana C.R. Paiva et al. “End-to-end Automatic Business Process Validation”. In: *Procedia Computer Science* 130 (2018). The 9th International Conference on Ambient Systems, Networks and Technologies (ANT 2018) / The 8th International Conference on Sustainable Energy Information Technology (SEIT-2018) / Affiliated Workshops, pp. 999–1004. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.04.104. URL: https://www.sciencedirect.com/science/article/pii/S1877050918304666.

[77] Daniel Plagge and Michael Leuschel. “Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more”. In: *International journal on software tools for technology transfer* 12.1 (2010), pp. 9–21.

[78] Daniel Plagge and Michael Leuschel. “Validating Z specifications using the ProB animator and model checker”. In: *International Conference on Integrated Formal Methods*. Springer. 2007, pp. 480–500.

[79] Victor Rivera et al. “Code generation for Event-B”. In: *International Journal on Software Tools for Technology Transfer* 19.1 (2017), pp. 31–52.

[80] John Rushby. *Formal Methods and the Certification of Critical Systems*. Tech. rep. SRI-CSL-93-7. Menlo Park, CA: Computer Science Laboratory, SRI International, 1993. URL: http://www.csl.sri.com/papers/csl-93-7/.

[81] Johannes Ryser and Martin Glinz. “A scenario-based approach to validating and testing software systems using statecharts”. In: *Proc. 12th International Conference on Software and Systems Engineering and their Applications*. Citeseer. 1999.

[82] Aymerick Savary et al. “Model-Based Robustness Testing in Event-B Using Mutation”. In: *Software Engineering and Formal Methods*. Ed. by Radu Calinescu and Bernhard Rumpe. Cham: Springer International Publishing, 2015, pp. 132–147. ISBN: 978-3-319-22969-0.

[83] Graeme Smith and Luke Wildman. “Model Checking Z Specifications Using SAL”. In: *ZB 2005: Formal Specification and Development in Z and B*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 85–103. ISBN: 978-3-540-32007-4.

[84] Colin Snook and Michael Butler. “UML-B: Formal modeling and design aided by UML”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15.1 (2006), pp. 92–122.

[85] Colin Snook et al. “Domain-specific scenarios for refinement-based methods”. In: *Journal of Systems Architecture* 112 (2021), p. 101833.

[86] Ian Sommerville. *Software Engineering*. 9th ed. Harlow, England: Addison-Wesley, 2010. Chap. 4.

[87] David WJ Stringer-Calvert, Susan Stepney, and Ian Wand. "Using PVS to prove a Z refinement: A case study". In: *International Symposium of Formal Methods Europe*. Springer. 1997, pp. 573–588.

[88] Allison Sullivan et al. "Automated test generation and mutation testing for Alloy". In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2017, pp. 264–275.

[89] Jun Sun et al. "Bounded model checking of compositional processes". In: *2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE. 2008, pp. 23–30.

[90] Toufik Taibi, Angel Herranz-Nieva, and Juan José Moreno-Navarro. "Stepwise Refinement Validation of Design Patterns Formalized in TLA+ using the TLC Model Checker." In: *J. Object Technol.* 8.2 (2009), pp. 137–161.

[91] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. "Azucar: A SAT-Based CSP Solver Using Compact Order Encoding". In: *Theory and Applications of Satisfiability Testing – SAT 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 456–462. ISBN: 978-3-642-31612-8.

[92] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. "From Formal Specification in Event-B to Probabilistic Reliability Assessment". In: *Dependability, International Conference on* (July 2010), pp. 24–31. DOI: 10.1109/DEPEND.2010.12.

[93] *The Alloy Analyzer*. URL: https://alloytools.org/ (visited on 06/16/2021).

[94] *The ProB Animator and Model Checker Wiki*.

[95] Casper Thule et al. "Maestro: The INTO-CPS co-simulation framework". In: *Simul. Model. Pract. Theory* 92 (2019), pp. 45–61. DOI: 10.1016/j.simpat.2018.12.005. URL: https://doi.org/10.1016/j.simpat.2018.12.005.

[96] Amirhossein Vakili and Nancy A Day. "Temporal logic model checking in Alloy". In: *International Conference on Abstract State Machines, Alloy, B, VDM, and Z*. Springer. 2012, pp. 150–163.

[97] "Validation". In: *IEEE Std 610* (1991), pp. 1–217. DOI: 10.1109/IEEESTD.1991.106963.

[98] "Verification". In: *IEEE Std 610* (1991), pp. 1–217. DOI: 10.1109/IEEESTD.1991.106963.

[99] Fabian Vu, Michael Leuschel, and Atif Mashkoor. "Validation of Formal Models by Timed Probabilistic Simulation". In: *International Conference on Rigorous State-Based Methods*. Springer. 2021, pp. 81–96.

[100] Fabian Vu et al. "A multi-target code generator for high-level B". In: *International Conference on Integrated Formal Methods*. Springer. 2019, pp. 456–473.

[101] Michelle Werth and Michael Leuschel. "VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics". In: *Rigorous State-Based Methods*. Ed. by Alexander Raschke, Dominique Méry, and Frank Houdek. Cham: Springer International Publishing, 2020.

[102] Sebastian Wieczorek et al. "Applying Model Checking to Generate Model-Based Integration Tests from Choreography Models". In: *Testing of Software and Communication Systems*. Ed. by Manuel Núñez, Paul Baker, and Mercedes G. Merayo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 179–194. ISBN: 978-3-642-05031-2.

[103] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[104] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model checking TLA+ specifications". In: *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer. 1999, pp. 54–66.