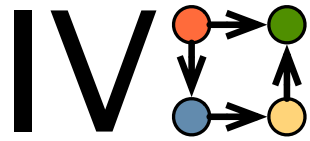

IVOIRE - Deliverable D 1.1

November 29, 2021

Version 1.0.0

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.
During the software development process, validation and verification play an important role to ensure the quality of each development stage until the final product. 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 [89]. **Verification** answers the question: "Are we building the software correctly?" In contrast, **validation** checks whether a model met the stakeholder's requirements [88]. It answers the question: "Are we building the right software?".

Proof obligations (POs) were introduced to tackle the former question, i.e., to check the model's consistency with its specification. Here, it is also checked whether a refinement preserves the properties described at abstract levels. Formally, POs are formulas extracted from the model which have to be proven. POs are already used in a refinement-based software development process. Here, POs are logical formulas that are extracted from the specification and checked by different solvers. [38]

This report defines *validation obligation* (VO) tackling the second question. In the following, we will describe how VOs are used in a refinement-based software development process to validate requirements (see Section 2). Afterwards, we will discuss the borderline between validation and verification (see Section 3). We will also describe the validation tasks associated with the VOs (see Section 5). 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.

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

2 Validation Obligations Approach

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.

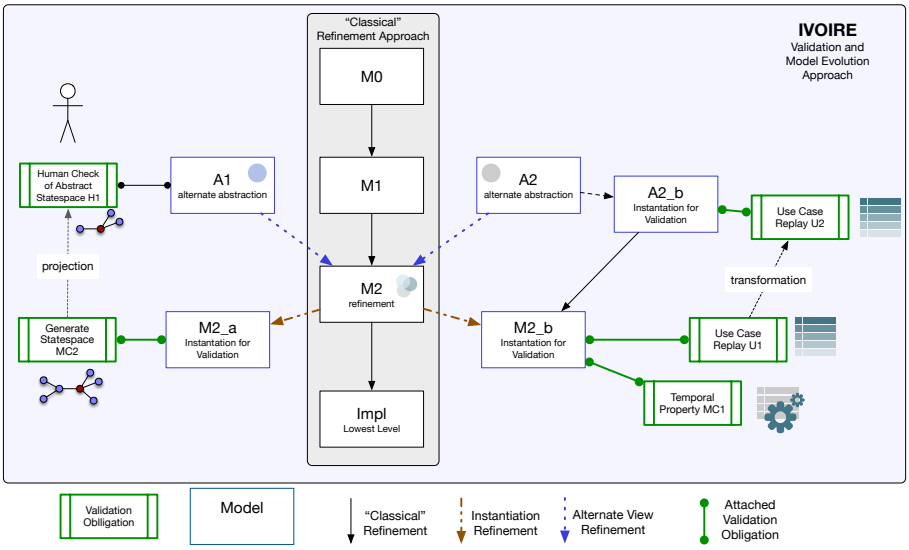

Figure 1: Refinement-based Software Development Process with VOs

We define the term of a VO as follows:

> A validation obligation (VO) is a validation task (VT) associated with one/many formal model(s) to check its/their compliance with a particular requirement.

A VO has a name, and consists of a validation task together with its parameters. Here, the parameters are optional. There are also validation tasks that do not expect any parameters. Executing a VO is done by executing the VO's validation task together with the parameters within a context resulting in TRUE (successful) or FALSE (failed). The context could be a model, or even multiple models. For example, the modeler could validate a requirement describing a safety property for a single model, while trace-refinement checking requires two models. Furthermore, each model can be an abstraction, specialization, or instantiation in the refinement-based software development process as shown in Figure 1. Validating a requirement succeeds if all associated validation tasks yield successful results. Thus, a validation task can be one of many tasks which are used to validate a single requirement.

Formally, the process of validation is described as a function *validate* which executes a VO in a context (consisting of one or many models), resulting in a boolean value:

$$validate : V \rightarrow \mathbb{B}$$

V denotes the set of VOs, and $\mathbb{B}$ = {TRUE, FALSE} denotes the set of booleans.

When introducing the VOs for each validation task in Section 5, we will use the notation in this form:

VO
Name: <Name>
Task: <Task>
Context: $< Model_1, \ldots, Model_n >$
Parameters: $< p_1, \ldots, p_n >$

Information
Action: *What has to be done?*
Automatable: <Yes/No/Partial>
Optional Information ...

The *information* clarifies how a VO is discharged and whether it is automatable. Furthermore, it also describes special abilities of the corresponding VO, e.g., a special ability of LTL model checking is the generation of a counterexample.

2.1 Comparison with Proof Obligations

POs can be extracted from the model using pre-defined rules (e.g. a rule for generating a PO for well-definedness). Furthermore, they are also discharged by the static information the model provides. This often happens via automatic proving but sometimes it has to be done manually.
In contrast, creating a VO is a manual process that has to be done by the modeler. Thus, the modeler's challenge is to define a VO with the correct configurations to validate the desired requirement. While some VOs have to be discharged by the modeler manually, there are also some VOs that are discharged automatically (see Section 5).

2.2 Development

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 being evolved or changed. Figure 2 shows the modeling process from the VO's perspective. First, a requirement is implemented into a model and a VT is defined to show that the model satisfies the requirement. A VT of a VO is executed in the context of the model leading to the result being either TRUE (successful) or FALSE (failure).

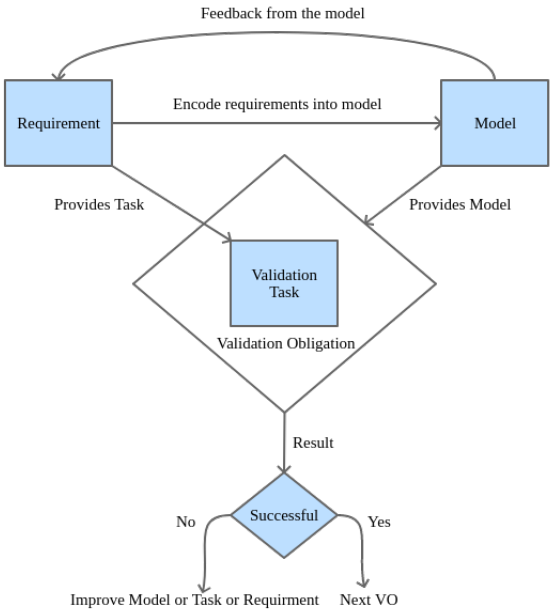

Figure 2: Relation of VO, VT, Requirement and Model

Once a VO is successfully discharged it is expected to hold in each following refinement. Thus, the requirement is ensured to be fulfilled in each subsequent element of the refinement chain. If it is not possible to discharge a requirement, the modeler may re-consider the requirement. Possible questions that the modeler could then ask are:

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

While modeling or validating a requirement, a behavior may be encountered which is not recognized in the requirements yet. In Figure 2 this is shown by the arrow going back from the model to the requirements and as the possible outcome of the VO.
In the case that original requirements need to be reconsidered, it might be necessary to change the original documents. This can also lead to cascading changes in the models and the VOs.

2.3 Creating Validation Tasks

The creation of VTs is similar to the creation of models. One needs significant knowledge about the modeling language, but also about the environment and

the techniques to create solvable tasks. It is the modeler's responsibility to choose a suitable validation technique.

The choice of the technique will influence how the modeler formulates the task e.g. the preservation of an invariant can be shown by model checking or proving. When creating a task to validate a requirement, the modeler must be aware of the available tools, and the complexity of the model as well as the property. E.g. proving and symbolic model checking are more suitable than explicit-state model checking to check an invariant in an infinite-state system. Currently, the creation of VTs is done manually due to its complexity. There are also tools like UML-B [75] which attempt an automatic translation from the specification to a model. Regarding the future, one could explore whether and how VTs can be extracted from the requirements automatically, and whether using a requirement language is useful. Noteworthy is that adding another requirement language also adds more complexity and another error source.

2.4 Refinement

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 validation obligations, 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 a combination of all abstraction/specializations/refinements 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.

2.5 Refactoring

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.

3 Overlap between Validation and Verification

While researching, we encountered the phenomenon that validation and verification are sometimes used as weak synonyms for each other. So, the purpose of this section is to discuss both terminologies and the area where they overlap. For VOs it is particularly important that they cover techniques that are classified as validation.

By definition, validation checks whether a model meets the stakeholder's requirements. So, the main question is: "Are we building the right software?". In contrast, verification checks whether a model meets its specification. So, here we ask the question: "Are we building the software correctly?"
Software is usually validated by by checking the behavior for specific inputs, e.g. when applying unit testing.
The techniques for VOs will be discussed in Section 5. Validation techniques which are clearly classified as validation are, e.g., animation, trace replay, testing, test case generation, and simulation. These techniques can be clearly stated as validation techniques.
Validation does not only mean that certain behavior is validated by a scenario. Referring to the definition, it means that it also checks whether the stakeholder's requirements are satisfied. Thus, those requirements cannot be covered by scenarios only. Instead, there are also requirements where it is necessary to determine the system's behavior in each possible state. For this purpose, model

checking techniques, proving, and (trace) refinement checking are also taken as validation techniques into account.
As an example, let us consider a requirement describing a safety property such as "The lift can always 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 stakeholder's requirements are fulfilled. Note that we do not disagree with model checking, proving, and (trace) refinement checking being a verification technique. Instead, we believe that they are both validation and verification.

We have also encountered techniques that we clearly see as verification (and not as validation):

- Well-definedness Checking
- Checking for integer overflows
- Checking absence of infinite loops

However, these techniques could still be important to ensure the model's and VOs' consistency.

4 Classification of Requirements

By definition according to the IEEE standard 729 [42], 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. [77]

4.1 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. [77]

Thus, functional requirements include descriptions of e.g. 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. [77]

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. [77]

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 trace refinement might play an important role during the validation.

4.2 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. [77]

5 Validation Techniques and Validation Obligations

This section takes possible validation techniques into account which can be used in a validation task to validate requirements. We will also discuss their strengths and weaknesses, and the tools supporting them. Furthermore, we will define validation obligations for each validation technique. As already explained in Section 2.2, the formulation of a VT is up to the modeler.

5.1 Validation by Animation, Trace Replay, Testing

Animation provides the opportunity for a human to interact with the model. 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. [44]

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. [44]

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. Trace replay is applied similar to animation, but with the main difference that it is done automatically.

Let $T = [t_1, \ldots, t_n]$ be a trace consisting of a list with 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. Furthermore, we will use the notation $t <\psi>$ for a transition t and a predicate ψ. In the case that there is no postcondition to be checked after re-playing a transition, 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. The trace replay VO is then defined as follows:

Trace Replay VO
Name: TR
Task: Trace Replay (or Animation manually)
Context: Model
Parameters: T

Information
Action: Succeeds if all transitions $t_1, \ldots, t_n$ can be re-played and all correspnding postconditions are checked successfully, fails otherwise

Automatable: Yes

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.

5.2 Validation by Trace Refinement

As described earlier, animation and trace replay helps to determine whether a behavior is present in a model. When refining a model, additional behaviors are encoded. Now, we want to check whether the trace of the abstract model still conforms with the concrete one. This is done by trace refinement.
Trace refinement adapts an abstract trace for the refinement. It is common for

modeling languages to even use traces as evidence of a correct refinement of the model. In CSP, a refinement check is a trace refinement check over all accessible traces. Such a tool performing refinement checking is FDR [37].
In languages like ASM, runs (trace-like structures) and their relationship between each other define the refinement's behavior. Trace refinement is possible under the assumption that a trace can be created and that some sort of formalized refinement rules exists. One can apply the refinement rules onto the trace and create a trace working on the refinement this way.
A core problem is to determine if two traces are equivalent. There are two points of view here:

1. A trace represents a high-level behavior or scenario. If the scenario is still possible the trace should be transformed in a way to represent the scenario.

2. A trace is a low-level sequence of states and transitions. Those are transformed with the help of the refinement rules (e.g. hiding transitions in CSP). If the resulting trace can be executed, the behavior represented by the trace is preserved.

While the latter can be properly formalized, the former one cannot. Thus, the checking trace refinement is applied to the low-level representation of a trace.

Let M_a and M_c be two machines with M_c refining M_a, and let T be a valid trace in M_a. Executing the corresponding VO checks whether the behavior of T is preserved in M_c. It is defined as follows:

Trace Refinement VO
Name: TRF
Task: Trace Refinement
Context: Abstract Model (M_a), Concrete Model (M_c)
Parameters: T

Information
Action: Succeeds if shown that T can be adapted to M_c successfully according to the refinement rules
Automatable: Yes
Special Ability: Provides an adapted/concrete trace

5.3 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 and MC/DC coverage. As already suggested by the name, the goal of operation coverage is to cover each operation. In contrast, MC/DC coverage is used to cover all possible outcomes of each operation. [73, 93]

Let $O = \{o_1, \ldots, o_n\}$ be the set of operations that should be covered, and let δ be the desired search depth. Then, the operation coverage test case generation VO is defined as follows:

Operation Coverage Test Case Generation VO
Name: OC
Task: Test Case Generation for Operation Coverage
Context: Model
Parameters: O, δ

Information
Action: Succeeds if all operations can be covered to the given search depth, fails otherwise

Automatable: Yes
Special Ability: Generates a trace for each covered operations

Again, let l be the desired MC/DC level, and let δ be the desired search depth. Then, the MC/DC coverage test case generation VO is defined as follows:

MC/DC Test Case Generation VO
Name: MCDC
Task: Test Case Generation for MC/DC Coverage
Context: Model
Parameters: l, δ

Information
Action: Succeeds if MC/DC criterion to the given search depth can be fulfilled, fails otherwise

Automatable: Yes
Special Ability: Generates traces for MC/DC criterion

5.4 Validation by Simulation

There are different kinds of simulation techniques that can be used for validation.

Assuming that a model is too abstract for execution, one technique is to enable the user to define constraints and concrete instantiations for the non-executable constructs which should match future refinements. This makes it possible to validate these models anyway. So, the model can still be executed

and validated automatically without creating a whole refinement model. The disadvantage of this technique is that the configuration is up to the user. On the one hand, the resulting state space should focus on the relevant parts concerning further refinements and implementations. On the other hand, the defined constraints and instantiations must be consistent with the model. [95, 44]

Another simulation technique is called *Co-Simulation*. 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, it is necessary that the subsystems exchange data with each other which again might trigger events. [86]

Again, there also exists a simulation technique which is called *timed probabilistic simulation*. Here, 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. Monte Carlo simulation [64] can also be applied in the context of timed probabilistic simulation to generate a various number of simulations. Based on the generated simulations, the modeler could also apply statistical validation techniques such as hypothesis testing [46] and estimation [29] to validate timing and probabilistic behavior. To get more information about the simulations, the modeler could also take simulation statistics into account, e.g. the percentage of how often an event was executed in the simulations when it was enabled. By considering other validation techniques such as state space visualization, and state space statistics, it is also possible to gain information about the part of the state space that is covered by the simulations. [90]

As result, we define VOs for hypothesis testing, estimation of probability, simulation, and simulation statistics.

Let N be the number of simulations, let H be the hypothesis formalized as a property to be checked, and α be the significance level. Then the VO for hypothesis testing is defined as follows:

Hypothesis Testing VO
Name: HT
Parameters: N, H, α
Context: Model, Simulation
Task: Hypothesis Testing

Information
Action: Succeeds if hypothesis accepted within the significance level α
Automatable: Yes
Special Ability: Generates simulations, each of them representing a scenario

Let N be the number of simulations, let P be the property (containing the desired probability) to be checked, and δ be a value defining the range of the desired probability. The VO for estimating the probability is then defined as follows:

Estimation of Probability VO
Name: EPR
Parameters: N, P, δ
Context: Model, Simulation
Task: Estimation of Probability

Information
Action: Succeeds if the estimated probability for the given property is within a given range defined by δ

Automatable: Yes
Special Ability: Generates simulations, each of them representing a scenario

Furthermore, it is often also useful to inspect the simulation statistics after applying multiple simulations. Let P_{sistat} be the set of simulation statistics properties, and let $R_{sistat} \subseteq P_{sistat} \times \mathbb{R}_{\geq 0}$ be a well-defined function mapping each statistic property to its value. Given a number of simulations N, a starting condition C_{start} for each simulation, an ending condition C_{end} for each simulation, and ϕ a predicate over R_{sistat}, the corresponding VO is defined as follows:

Simulation Statistics VO
Name: SISTAT
Task: Simulation Statistics
Context: Model, Simulation
Parameters: N, C_{start}, C_{end}, ϕ

Information
Action: Generates simulation statistics (can be transformed to R_{sistat}) based on N simulations which has to be inspected by the modeler. The VO succeeds if ϕ is true.

Automatable: No
Special Ability: Generates simulations, each of them representing a scenario

5.5 Validation by Model Checking

Explicit-state Model Checking Explicit-state model checking checks state-based behaviors of a system by exploring its state space exhaustively. Exhaustive exploration leads to full coverage of the system's behavior when the

model checking process terminates. Furthermore, it is then ensured whether the checked properties are true or false. In the case that a property is violated, explicit-state model checking can return a counter-example. Nevertheless, this validation technique 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]

Let F be the set of logical formulas,and let c be a model checking configuration with $c \in Conf_{MC,F}$, where as $Conf_{MC,F}$ denotes the set of model checking configurations accepting formulas in F. Currently, we have defined the set $Conf_{MC,F}$ as:

$$Conf_{MC,F} = \{<\text{FIN}>, <\text{DLF}>\} \cup \{<\text{INV}, \psi > | \ \psi \in F\} \cup \{<\text{GOAL}, \psi > | \ \psi \in F\}$$

Regarding the future, the $Conf_{MC,F}$ could be extended by new configurations. <FIN> denotes the configuration applying explicit-state model checking without checking any properties. So, after the check finishes, the modeler only knows that the state space is finite. Thus, the aim is to check whether explicit-state model checking terminates and therefore the state space is finite. <INV, ψ > represents the configuration for invariant checking with a formula ψ. Again, <DLF> is the configuration for deadlock checking. Furthermore, <GOAL, ψ > stands for a configuration to find a goal/a state satisfying the formula ψ.

The corresponding explicit-state model checking VO is then defined as follows:

Explicit-State Model Checking VO
Name: MC
Task: Explicit-State Model Checking
Context: Model
Parameters: c

Information
Action: Succeeds if configuration satisfied on state space, fails otherwise
Automatable: Yes
Special Ability: Provides (counter-)example

Temporal Model Checking 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. Using the transition system and the Büchi automaton that is created from the LTL formula, LTL model checking can check 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]

Let ψ be an LTL formula, and let $c \in$ {SUCCESS, FAIL}. The VO for LTL model checking is defined as follows:

LTL Model Checking VO
Name: LTL
Task: LTL Model Checking
Context: Model
Parameters: ψ, c

Information
Action: Succeeds if LTL formula either satisfied on state space for $c =$ SUCCESS, or failed for $c =$ FAIL
Automatable: Yes
Special Ability: Provides (counter-)example

To formulate more expressive temporal properties, the modeler could also write CTL formulas and apply CTL model checking. 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]

Let ψ be a CTL formula, and let $c \in$ {SUCCESS, FAIL}. The VO for CTL model checking is defined as follows:

CTL Model Checking VO
Name: CTL
Task: CTL Model Checking
Context: Model
Parameters: ψ, c

Information
Action: Succeeds if CTL formula either satisfied on state space for $c =$ SUCCESS, or failed for $c =$FAIL
Automatable: Yes
Special Ability: Provides (counter-)example

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

Symbolic Model Checking Symbolic model checking 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. [49]

Let F be a set of logical formulas, and let c be a model checking configuration with $c \in Conf_{MC,F} \setminus \{<\text{FIN}>\}$, whereas $Conf_{MC,F}$ is defined as described before. The corresponding symbolic model checking VO is then defined as follows:

Symbolic Model Checking VO
Name: SMC
Task: Symbolic Model Checking
Context: Model
Parameters: c

Information
Action: Succeeds if configuration satisfied on state space, fails otherwise
Automatable: Yes
Special Ability: Provides (abstracted) (counter-)example

Probabilistic and Statistical Model Checking 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. Here, probabilistic temporal model checking is taken into account. 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. [52, 53]

Let ψ be a probabilistic temporal formula, and let $c \in$ {SUCCESS, FAIL}. The VO for probabilistic/statistical model checking is defined as follows:

Probabilistic/Statistical Model Checking VO
Name: PSMC
Task: Probabilistic/Statistical Model Checking
Context: Model
Parameters: ψ, c

Information
Action: Succeeds if probabilistic temporal formula either satisfied on state space or simulations for c = SUCCESS, or failed for c = FAIL
Automatable: Yes
Special Ability: Provides statistics, and (counter-)examples

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. [90]

5.6 Validation by Proving

Proving is a technique which 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. 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.

Let ψ be a logical formula, then the VO for proving is defined as follows:

Proving VO
Name: PO
Task: Proving
Context: Model
Parameters: ψ

Information
Action: Succeeds if formula satisfied on the model, fails otherwise
Automatable: Partial

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.

5.7 Validation by Visualization, Statistics, and Metrics

State Space Visualization 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 see possible states and transitions between them to understand and validate the model. This can be done by visualizing and inspecting the whole state space. Let Z_{svis} be the set of reachable states, and T_{svis} be the set of possible transitions between them, i.e., $(z_1, t, z_2) \in T_{svis} \Leftrightarrow$ *there exists a transition from* z_1 *to* z_2 *which is labeled with* t. The state space is then defined as a 2-tuple $S_{svis} = (Z_{svis}, T_{svis})$. Given ϕ a predicate over S, the corresponding VO is then defined as:

State Space Visualization VO
Name: SVIS
Task: State Space Visualization
Context: Model
Parameters: ϕ

Information
Action: Generates state space visualization representing the state space S_{svis} which has to be inspected by the modeler manually. The VO succeeds if ϕ is true

Automatable: No

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. Projecting the state space on a formula results in an abstract visualization of the state space which is easier to understand. [51]

Let ψ be an expression formula to project the state space on an abstracted state space $S_\psi = (Z_\psi, T_\psi)$. Z_ψ and T_ψ then represent the abstracted set of reachable states and the abstracted set of possible transitions wrt. to ψ respectively. Given ϕ a predicate over S_ψ, the state space projection VO is then defined as:

State Space Projection VO
Name: SPRJ
Task: State Space Projection
Context: Model
Parameters: ψ, ϕ

Information
Action: Generates state space projection S_ψ on value of ψ which has to be inspected by the modeler manually. The VO succeeds if ϕ is true.

Automatable: No

Based on the abstracted state space, one could apply other VOs, e.g. LTL model checking.

Enabling Diagram An enabling diagram is a diagram describing for each operation which operations are enabled when executing this operation [27]. Let E be the set of events, and let $R_{ed} \subseteq E \times E$ be a relation with $(e_1, e_2) \in R_{ed} \Leftrightarrow$ e_1 *enables* e_2. Given ϕ a predicate over R_{ed}, the corresponding VO is defined as follows:

Enabling Diagram VO
Name: ED
Task: Enabling Diagram
Context: Model
Parameters: ϕ

Information
Action: Generates an enabling diagram of the model (can be transformed to relation R_{ed}) which has to be inspected by the user. The VO succeeds if ϕ is true.

Automatable: No

Vacuous Guards/Invariants Analysing vacuous invariants and guards is applied to the corresponding predicate, checking whether there is a redundant conjunct.

Let $c \in$\{GRD, INV\}, then the corresponding VO is defined as follows:

Vacuous Parts VO
Name: VAP
Task: Vacuous Parts
Context: Model
Parameters: c

Information
Action: Calculates whether there are vacuous parts in invariants or guards
Automatable: Yes
Special Ability: Provides sub-predicate containing the vacuous parts, can thus be used to improve the predicate

Operation Metrics/Table

- *Operation Coverage Table*: This table describes for each operation whether it is covered yet. There is also a version of this metric that is limited to feasible operations. Let E be the set of operations, and let $R_{oct} \subseteq E \times \{COVERED, UNCOVERED\}$ be a well-defined function with $(e, UNCOVERED) \in R_{oct} \Leftrightarrow e$ *not covered yet* and $(e, COVERED) \in R_{oct} \Leftrightarrow e$ *already covered*. Given ϕ a predicate over R_{oct}, the corresponding VO is defined as follows:

Operation Coverage Table VO
Name: OCT
Task: Operation Coverage Table
Context: Model
Parameters: ϕ

Information
Action: Generates a table describing for each operation whether it has been covered yet (can be transformed to relation R_{oct}) which has to be inspected by the user. The VO succeeds if ϕ is true.

Automatable: No

- *Read/Write Matrix*: There are two types of read/write matrices: one for operations and one for variables. The read/write matrix for operations stores which variables an operation reads and writes. In contrast, the read/write matrix for variables stores for each variable by which operation it is read and written.

  Let E be the set of events, let V be the set of variables, and let $R_{rwm} \subseteq \{READ, WRITE\} \times (E \times V)$ be a relation with $(READ, (e, v)) \in R_{rwm} \Leftrightarrow$

e *reads* v and $(WRITE, (e, v)) \in R_{rwm} \Leftrightarrow e$ *writes* v. Given ϕ a predicate over R_{rwm}, the corresponding VO is defined as follows:

Read/Write Matrix VO
Name: RWM
Task: Read/Write Matrix
Context: Model
Parameters: ϕ

Information
Action: Generates a read/write matrix storing read/write relation between operations and variables (can be transformed to relation R_{rwm}) which has to be inspected by the user. The VO succeeds if ϕ is true.

Automatable: No

Variable Metrics/Table

- *Variable Coverage*: This metric provides the number of values a variable can actually be assigned to. Let V be the set of variables, and let $R_{vct} \subseteq V \times \mathbb{N}_0$ be a well-defined function mapping each variable to the number of values it can be assigned to. Given ϕ a predicate over R_{vct}, the corresponding VO is defined as follows:

Variable Coverage VO
Name: VCT
Task: Variable Coverage Table
Context: Model
Parameters: ϕ

Information
Action: Generates a table containing the variable coverage metrics for each variable (can be transformed to relation R_{vct}) which has to be inspected by the user. The VO succeeds if ϕ is true.

Automatable: No

- *Min/Max Values for Variables*: This metric provides the minimum and maximum value each variable can actually be assigned to.

  Let V be the set of variables, and for each $v \in V$ let $\tau(v)$ be the set of all values that can have the type of v. Furthermore, let $R_{mmv} \subseteq V \times (\bigcup_{t \in V} \tau(t) \times \bigcup_{t \in V} \tau(t))$ be a function mapping each variable to its minimum and maximum value. For each variable v' where the minimum

and maximum value is not defined due to its type, R_{mmv} is defined such that $v' \notin dom(R_{mmv})$. For each tuple $(m, n) \in ran(R_{mmv})$, we define $min(m, n) = m$ and $max(m, n) = n$. Given ϕ a predicate over R_{mmv}, the corresponding VO is defined as follows:

Min/Max Values VO
Name: MMV
Task: Min/Max Values Table
Context: Model
Parameters: ϕ

Information
Action: Generates a table containing the minimum and maximum value for each variable (can be transformed to R_{mmv}) which has to be inspected by the user. The VO succeeds if ϕ is true.

Automatable: No

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. In order to determine which events are particularly important for the model, one could also take the number of transitions for each event into account.

Let P_{spstat} be the set of state space statistics properties, and let $R_{spstat} \subseteq P_{spstat} \times \mathbb{R}_{\geq 0}$ be a well-defined function mapping each statistic property to its value. Given ϕ a predicate over R_{spstat} the corresponding VO is defined as follows:

State Space Statistics VO
Name: STAT
Task: State Space Statistics
Context: Model
Parameters: ϕ

Information
Action: Generates state space statistics containing number of states, number of transitions etc. (can be transformed to R_{spstat}) which has to be inspected by the user. The VO succeeds if ϕ is true.

Automatable: No

5.8 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 [91] into account. This code generator 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 where other 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.

5.9 Languages and their tools

For our research we have investigates nine major modeling languages regarding their tool support for different task. The results are shown in Table 1. 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.

6 Demonstration of Validation Obligations

In this section, we will demonstrate based on an example 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:

[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

Table 1: Specification Languages and Supported Validation Techniques

Tools	Alloy	ASM	B	Event-B	VDM	TLA+	Z	CSP	Circus
Animation	✗	✓([12])	✓([54, 92])	✓([11, 54, 92])	✓([66])	✓([39])	✓([11, 24, 70])	✓([11, 18])	✗
Trace Replay/Testing	✓([84, 23]) [1]	✓([19])	✓([10])	✓([76, 10])	✗	✓([39, 10])	✓([26])	✓([20])	✗
Test Case Generation	✓([79])	✓([33, 34, 35])	✓([85])	✓([73, 93, 85])	✓([28])	✓([39, 85])	✓([41, 85])	✓([65])	✗
Simulation	✓([16])	✓([36])	✓([90])	✓([90])	✓([86, 30])	✓([90])	✓([90])	✓([90])	?
Explicit-State MC	✓([17, 23])	✓([5])	✓([54])	✓([11, 54])	✓([58])	✓([96, 39])	✓([70, 11])	✓([37, 11])	✗
LTL MC	✓([17, 23])	✓([5])	✓([69])	✓([69])	✓([58])	✓([39, 69])	✓([69, 25])	✓([69, 55])	✗
CTL MC	✓([87])	✓([5])	✓([54])	✓([11, 54])	✓([58])	✓([11, 54, 39])	✓([25])	✓([54])	✗
Symbolic MC	✓([17, 23])	✓([5])	✓([49])	✓([49])	✗	✓([48, 47, 49])	✓([74])	✓([80])	✗
Probabilistic/Statistical MC	✗	✗	✗	✓([4, 83])	✗	✗	✗	✗	✗
Proving	✓([67])	✓([8])	✓([63])	✓([1, 2])	✓([3])	✓([21])	✓([15, 31])	✓([82, 43])	✓([32])
Refinement Checking	✗	✓([6, 14])	✓([63])	✓([1, 2])	✓([59])	✓([81])	✓([78])	✓([37])	✓([32])
State Space Visualization	?	?	✓([51])	✓([51])	✗	✓([50, 51])	✓([51])	✓([54, 11, 18])	✗
Code Generation [2]	✗	✓([13])	✓([91])	✓([71])	✓([40])	✗	✗	✓	✓([9])

First, we will describe the functional requirements from which the model will be created.

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 the traffic lights is at least always red.

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.

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.

Regarding **SCENARIO1**, a trace refinement criterion is expected to hold as shown in **TRC1**.

TRC1: SCENARIO1 should be feasible from a perspective described in **FUN8 - FUN10**.

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.

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 trace refinement, the machine shown in Listing 1 will also 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 3.

Based on the model, the designer could also run different simulations. Listing 2 shows a SimB file [90] 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.

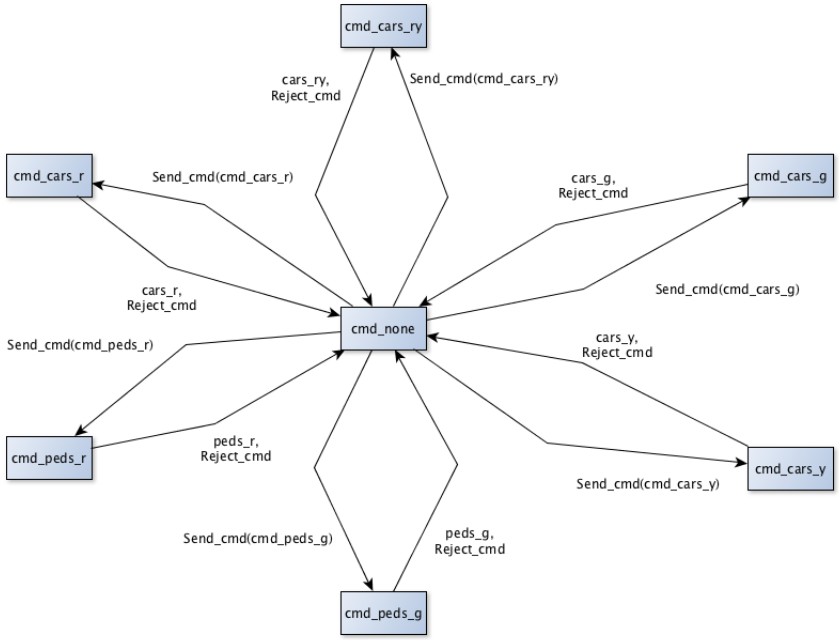

Figure 3: **PRC1** as Diagram

```
{
 "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)

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 the model such that all events are at least explored once.

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

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 enabling relation of events is 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 should be 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 VOs from all types, we will also present alternative VOs to demonstrate all VO 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.

The notion we will use is as follows:

> $\mathbf{VO_{id}}/VO_{context}/VO_{name} : VO_{parameters}$

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 an LTL model check expecting a positive result:

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

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 an LTL model check expecting a positive result.

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

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 an LTL model check expecting a positive result.

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

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. Thus, this behavior can be validated by 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

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

- The cars' traffic light might change its color unequal red.
- 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.

Both behaviors can be formulated as an LTL model check. 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.

LTL5.1/TrafficLight/LTL: ¬ (F{tl_cars ≠ red}), FAIL

LTL5.2/TrafficLight/LTL: ({tl_cars = red} W ({tl_cars ≠ red} & G({tl_cars=red} ⟹ ({tl_cars=red} S {tl_cars=yellow & tl_peds=red})))), SUCCESS

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. Thus, this behavior can be validated by an LTL model check expecting a positive result.

LTL6/TrafficLight/LTL: G ({tl_peds=green} ⟹ ({tl_peds=green} S {tl_cars=red & tl_peds=red})), SUCCESS

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.

LTL7.1/TrafficLight/LTL: ¬ (F{tl_peds ≠ red}), FAIL

LTL7.2/TrafficLight/LTL: ({tl_peds = red} W ({tl_peds ≠ red} & G({tl_peds=red} ⟹ ({tl_peds=red} S {tl_cars=red & tl_peds=green})))), SUCCESS

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. Here, we will present the VOs for explicit-state model checks.

Validation of **SAF1**:

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

Validation of **SAF2**:

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

Validation of **SAF3**:

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

In contrast, **LIV1** is a requirement describing a liveness property. Thus, it can be checked by an LTL model check expecting a positive result which is formalized as:

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

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.

> **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>]

> **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>]

To validate **TRC1**, we have developed the evolved abstract, and the concrete machine for traffic light into account (see Listing 4 and Listing 5). Now, we will demonstrate trace refinement of **TR1**. As new events refining `skip` are added to the abstract machine, it is necessary to adapt the trace in **TR1** to the one shown in **TRF1**. The trace should then be refined and replayed on the concrete machine afterwards.

TRF1/TrafficLight2, TrafficLightCommand_Ref/TRF: [INITIALISATION <tl_cars = red & tl_peds = red>, Send_cmd(cmd=cmd_cars_ry), cars_ry <tl_cars = redyellow & tl_peds = red>, Send_cmd(cmd=cmd_cars_g), cars_g <tl_cars = green & tl_peds = red}, Send_cmd(cmd=cmd_cars_y), cars_y <tl_cars = yellow & tl_peds = red>, Send_cmd(cmd=cmd_cars_r), cars_r <tl_cars = red & tl_peds = red>]

As mentioned before, a domain expert could be interested in the logic for sending commands only, without taking the traffic lights' colors into account. The diagram portrayed in **PRC1** (Figure 3) could then be validated by projecting the state space on `queuedCmd` for inspection afterwards. This is formalized in **SPRJ1**.

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}

When validating **PROB-TIM1**, the modeler could apply hypothesis testing, estimation of probability, or inspecting the simulation statistics. Here, we will demonstrate the validation of this requirement by applying the hypothesis testing VO **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**).

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

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

In the following, we are going to demonstrate 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 validating **SCENARIO1** and **SCENARIO2**. Here, it is necessary to check whether the values for `tl_cars` and `tl_peds` are equal to 4 and 2 respectively.

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

Similar to the validation of **COV1**, the modeler must also apply **SCENARIO1** and **SCENARIO2** first to validate **COV2**. It is then necessary to inspect the operation coverage table manually, to check whether all events have been covered. This is 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}

COV3 describes the desired statistics for the number of states and transitions after 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. In order to check this coverage criterion, the modeler has to check the VO shown in **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**.

MC4/TrafficLight/MC: <FIN>

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

The requirements **STRUC1** and **STRUC2** desire `tl_cars` and `tl_peds` to be written by certain events. This can be encoded to the respective VOs **RWM1** and **RWM2** directly. Those VOs 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}

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

Again, ensuring that there are no vacuous parts in the invariant and guards of the Traffic Light model (**STRUC3**) can be checked by the VOs shown in **VAP1** and **VAP2**.

VAP1/TrafficLight/VAP: INV

VAP2/TrafficLight/VAP: GRD

STRUC4 describes how events enable each other. This can be translated to the corresponding VO **EN1**. In order to validate the requirement, this VO is checked by inspecting the enabling diagram manually.

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}.

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 which is realized by **OC1**. Again, MCDC coverage test case generation with depth 5 is suitable to validate **STRUC6** (see **MCDC1**).

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

MCDC1/TrafficLight/MCDC:2,5

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 VOs that have not been used yet. Some VOs will be demonstrated on existing requirements of the Traffic Light example. In contrast, there will also be VOs that will be demonstrated on other requirements, or even other models.

Instead of validating **SAF1** by checking **MC1**, it would also be possible to apply symbolic model checking as mentioned before. The corresponding VO for symbolic model checking would be as follows:

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

Another possibility to validate **SAF1** could be done by proving multiply proof obligations. Here, it would be necessary to generate a PO for the initialisation, and for each event checking whether it preserves the invariant describing **SAF1**. As result, this would lead to seven POs being generated, one for each operation, to validate **SAF1**. The proof obligation **PO1** shows the proof obligation 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

As an alternative to **STAT1**, one could also inspect the state space visualization after validating **SCENARIO1** and **SCENARIO2**, and applying **MC4** manually. The corresponding VO is shown in **SVIS1**. 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: $card(Z_{svis}) = 6$ & $card(T_{svis}) = 7$

As an alternative to **LTL5.1** and **LTL7.1.**, it would also be possible to apply CTL model checking to expect a positive result. This is illustrated in **CTL1** and **CTL2**.

CTL1/TrafficLight/CTL: EF{tl_cars ≠ red}, SUCCESS

CTL2/TrafficLight/CTL: EF{tl_peds ≠ red}, SUCCESS

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.

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

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 VO **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 the VO defined in **PSMC1** which expects a 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

Another alternative to validate **PROB1** is the simulation statistics VO:

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]

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, 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 the VO inspecting the minimum and maximum value as shown in **MMV1**.

MMV1/Lift/MMV: min(R_{mmv}(level)) = 0 & max(R_{mmv}(level)) = 3

A requirement describing a scenario could also be validated by multiple VOs from different types. This will be demonstrated in an automotive case study [56]. 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>]

A 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 finite number of transitions describing a path through the state space. Each transition consists out of the name of the operation/event/-function that was used to reach the next state and predicates that describe the next state, and the input and output of the operation/event/function.

Scenario A scenario is a sequence of events in which certain behavior patterns are desired. In general, a scenario can be described in natural language. It is realized as a trace, and can be the result of a simulation.

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 [45, 68, 57, 72]. 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 [94]. In software engineering, there is the sentiment that traces are a realisation of a scenario, shown for example in [62]. 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 Technique A validation technique is a technique that can be used 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 a task corresponding to a validation technique that is formulated by the modeler, in order to validate a requirement. Executing the validation task leads to a binary outcome (TRUE or FALSE) describing whether the task was applied successfully. Validating a requirement succeeds if all associated validation tasks yield successful results. Thus, a validation task can be one of many tasks which are used to validate a single requirement.

As an example for a validation task, we consider the requirement "Cars' traffic light can either be red, red and yellow, yellow, or green.". Here, it could be validated by an explicit-state model check with the invariant $tl\_cars \in \{red, redyellow, yellow, green\}$.

B 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

C Overview VO Examples

Trace Replay VO:

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>]

Trace Refinement VO:

TRF1/TrafficLight2, TrafficLightCommand_Ref/TRF: [INITIALISATION <tl_cars = red & tl_peds = red>, Send_cmd(cmd=cmd_cars_ry), cars_ry <tl_cars = redyellow & tl_peds = red>, Send_cmd(cmd=cmd_cars_g), cars_g <tl_cars = green & tl_peds = red}, Send_cmd(cmd=cmd_cars_y), cars_y <tl_cars = yellow & tl_peds = red>, Send_cmd(cmd=cmd_cars_r), cars_r <tl_cars = red & tl_peds = red>]

Operation Coverage Test Case Generation VO:

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

MC/DC Coverage Test Case Generation VO:

MCDC1/TrafficLight/MCDC:2,5

Hypothesis Testing VO:

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 VO:

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 VO:

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 VO:

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

LTL Model Checking VO:

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

CTL Model Checking VO:

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

Probabilistic/Statistical Model Checking VO:

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 VO:

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

Proving VO:

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 VO:

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

Min/Max Values VO:

MMV1/Lift/MMV: $\min(R_{mmv}(\text{level})) = 0$ & $\max(R_{mmv}(\text{level})) = 3$

Operation Coverage Table VO:

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 VO:

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

Enabling Diagram VO:

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 VO:

VAP1/TrafficLight/VAP: INV

State Space Visualization VO:

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

State Space Projection VO:

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 VO:

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

Multiple VOs for Validation:

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>]

D 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 [61]
- Fabian Vu, Michael Leuschel, Atif Mashkoor. Validation of Formal Models by Timed Probabilistic Simulation [90]
- Atif Mashkoor, Alexander Egyed. Evaluating the alignment of sequence diagrams with system behavior [60]
- 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] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

[31] L. Freitas. "Proving Theorems with Z / Eves". In: 2005.

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

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

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

[35] 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.

[36] 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.

[37] 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.

[38] Stefan Hallerstede. *The Event-B proof obligation generator*. Tech. rep.

[39] 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.

[40] 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.

[41] 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.

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

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

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

[45] 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.

[46] Maurice G. Kendall, Alan Stuart, and J. Keith Ord. “Kendall’s Advanced Theory of Statistics”. In: *Oxford University Press*. 1987. ISBN: 0195205618.

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

[48] 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.

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

[50] 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.

[51] 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.

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

[53] 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.

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

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

[61] 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].

[62] 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.

[63] 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.

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

[65] 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.

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

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

[68] 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.

[69] 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.

[70] 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.

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

[72] 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.

[73] 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.

[74] 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.

[75] 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.

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

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

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

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

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

[86] 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.

[87] 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.

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

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

[90] 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.

[91] 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.

[92] 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.

[93] 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.

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

[95] Faqing Yang, Jean-Pierre Jacquot, and Jeanine Souquières. "JeB: Safe Simulation of Event-B Models in JavaScript". In: *Proceedings APSEC, Volume 1*. IEEE, 2013, pp. 571–576.

[96] 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.