International Journal on Software Tools for Technology Transfer (2024) 26:147–168
https://doi.org/10.1007/s10009-024-00739-0

GENERAL

Special Issue: FMICS 2022

Generating interactive documents for domain-specific validation of formal models

Fabian Vu[1] · Christopher Happe[1] · Michael Leuschel[1]

Accepted: 6 February 2024 / Published online: 20 February 2024
© The Author(s) 2024

Abstract
Especially in industrial applications of formal modeling, validation is as important as verification. Thus, it is important to integrate the stakeholders' and the domain experts' feedback as early as possible. In this work, we propose two approaches to enable this: (1) a static export of an animation trace into a single HTML file, and (2) a dynamic export of a classical B model as an interactive HTML document, both based on domain-specific visualizations. For the second approach, we extend the high-level code generator B2Program by JavaScript and integrate VisB visualizations alongside SimB simulations with timing, probabilistic and interactive elements. An important aspect of this work is to ease communication between modelers and domain experts. This is achieved by implementing features to run simulations, sharing animated traces with descriptions and giving feedback to each other. This work also evaluates the performance of the generated JavaScript code compared with existing approaches with Java and C++ code generation as well as the animator, constraint solver, and model checker ProB.

Keywords Code generation · Validation · B method · Domain-specific · Interactive · Visualization

1 Introduction and motivation

Verification shows the correctness of software, thus tackling the question "Are we building the software correctly?" [36]. During the verification process, it might indicate errors. Just as important is validation, which checks whether the stakeholders' requirements are fulfilled and thus tackles the question "Are we building the right software?" [36].

An important aspect of validation is the dialog between modelers and stakeholders or domain experts. The latter are usually not familiar with the formal method and notation, while the modeler only has limited knowledge of the domain. Animation, simulation, and visualization of scenarios are important enabling technologies: a domain expert can grasp the behavior of a model by looking at visualizations without having to understand the underlying mathematical notation. Even for modelers, visualization is important; for instance, some errors are immediately obvious in a visual rendering, while they can remain hidden within the mathematical, textual counterpart (see various case studies, e.g., Vehicle's Light System [52], Landing Gear [44], ETCS Hybrid Level 3 [33], Air Traffic Control Software [28]).

In this paper, we tackle one further hurdle that domain experts or stakeholders face: in addition to lacking knowledge and experience with formal notations, they typically also lack the knowledge to drive the particular tool or possibly even install it. Even when a domain expert successfully installs such a tool, they have to work with features, techniques, and notations they usually are not familiar with. In this article, we implement two solutions to this:

– A static export of an animation trace into a single HTML file, that can be sent by email and rendered in any current browser. This export is available for all models supported by the animator, constraint solver, and model checker ProB [48, 49], and enables the user to navigate within the trace.
– A dynamic export of a classical B model (and optionally pre-configured traces), to an HTML document which can also be rendered in a current browser. This export uses the high-level B code generator B2Program [65] which is extended by JavaScript. While not applicable to all models, the export is completely dynamic: a user can freely navigate the model's state space, not just one pre-configured trace. Furthermore, a user can even run various simula-

✉ F. Vu
fabian.vu@uni-duesseldorf.de

M. Leuschel
leuschel@uni-duesseldorf.de

1 Heinrich-Heine-Universität Düsseldorf, Mathematisch-Naturwissenschaftliche Fakultät, Institut für Informatik, Düsseldorf, Germany

tions automatically and modify descriptions of traces. The dynamic export includes a domain-specific VisB visualization [71] and supports timed probabilistic simulation with SimB [66] (including user interaction [64]). This allows a domain expert to interact with a prototype in VisB and experience probabilistic and timing behavior.

With both solutions, one just needs to open a browser and the HTML document. A domain expert can then interact with the trace or model with a domain-specific visualization and familiar techniques.

First, we give some background in Sect. 2. We then present the validation workflow in Sect. 3. Section 4 describes the static export of an animation trace into a single HTML file. In Sect. 5, we describe a dynamic export of a classical B model to an interactive HTML document. Section 6 demonstrates how this work improves the validation of requirements by domain experts, and communication between modelers and domain experts. We also evaluate and discuss the applicability of the dynamic export, including the performance in Sect. 7. Finally, we compare our work with related work in Sect. 8, and conclude in Sect. 9.

This paper is an extended version of the FMICS 2022 paper [68]. For this, we implemented new features, such as (1) timed probabilistic simulation with SimB [66], (2) interactive simulation [64], and (3) model checking support [67] for JavaScript for the performance analysis. In this extension, we also allow domain experts to give more feedback on execution traces by writing description texts. Furthermore, we describe the domain experts' validation process and the generation of certain GUI components in more detail. We have also demonstrated the effectiveness of the SimB features and the feedback through descriptions using the existing case studies.

2 Background

The B method was introduced by Jean-Raymond Abrial and is a formal method for specifying and verifying software systems [2]. The B method includes the formal B modeling language, which is based on first-order logic and set theory. Within the B language, a component is a machine, which contains constants, variables, sets, together with the initialization, and operations. While variables represent the model's current state, the initialization and operations can be defined with substitutions (a.k.a. statements) that change the variables and thus also the state. Usually, an operation consists of a guard and a substitution, which means that the substitution is applied when the guard is true. Furthermore, each model contains an invariant, which is a predicate that must always be true.

Listing 1 shows an example of Eratosthenes sieve modeled in B. It has three variables: `numbers` (the candidates for prime numbers), `cur` (the current number being processed) and a `limit` to stop the algorithm. The model has three operations, one to start the sieve, one to treat the next number and one to finish. As shown in Listing 1, one can see that the model consists of arithmetic, logical, and set operations.

```
MACHINE Sieve
VARIABLES numbers, cur, limit
INVARIANT
  numbers <: INTEGER & cur:NATURAL1 & limit:NATURAL1
INITIALISATION numbers := {} || cur := 1 || limit :=
     1
OPERATIONS
  StartSieve(lim) = PRE cur=1 & lim > MINLIM & lim <=
        MAXINT THEN
    numbers := 2..lim ||
    cur := 2 ||
    limit := lim
  END;

  prime <-- TreatNumber(cc) =
  PRE cc=cur & cur>1 & cur*cur<= limit THEN
    IF cc:numbers THEN
       numbers := numbers - ran(%n.(n:cur..limit/cur|
            cur*n))
      || prime := TRUE
    ELSE
       prime := FALSE
    END ||
    cur := cur+1
  END;

  r <-- Finish = PRE cur*cur>limit THEN
    cur := 1 || r := card(numbers)
  END
END
```

Listing 1 Example of Prime Number Sieve in B

The formal B language is a refinement-based modeling language. This means that the development chain consists of multiple machines that are gradually refined by further details. If the modeler intends to generate code for embedded systems from this, it must some point be refined to B0, which is a subset of B. B0 contains constructs from B that are at the implementation level. Those constructs are close to imperative programming languages, such as `WHILE` loops or `IF-THEN-ELSE` substitutions. The use of sets and relations is only restricted here. For example, one can use elements of enumerated sets or define total functions for arrays. Set and relation operators, however, are not allowed. High-level constructs such as nondeterminism or set definitions are also no longer allowed.

The main application fields of the B method are safety-critical systems. For example, railway systems such as the Paris Métro Line 14 [16], and the New York Canarsie Line [19] have been modeled and verified with the B method, and afterward, code generation has been applied. Another use case of the B model in the railway domain is the ETCS Hybrid Level 3, where formal B models are used at runtime [33]. Moreover, the B method has also been used for other industrial-motivated case studies in safety-critical domains, such as automotive [52] and aviation [44].

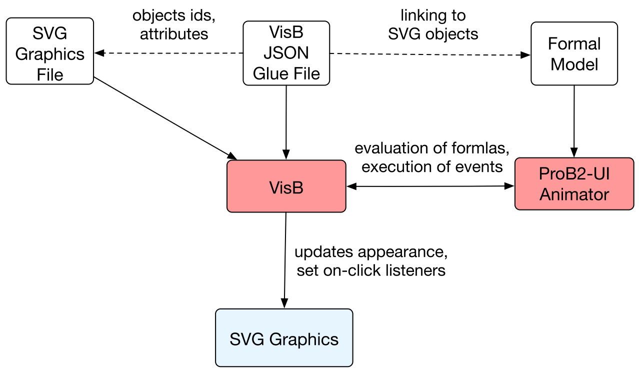


Fig. 1 Architecture of VisB and ProB2-UI (also proposed in Fig. 1 of [71]); idea related to Model-View-Controller (MVC) pattern [69]; SVG graphics file and VisB glue file are loaded in VisB, while formal model is loaded in ProB2-UI's animator; The VisB glue file connects the SVG graphics with the formal model; ProB2-UI's animator is used to evaluate formulas and execute events affecting the SVG objects' appearances; formal model events can also be executed via VisB (Color figure online)

ProB [48, 49] is an animator, constraint solver, and model checker for formal methods, such as B, Event-B, Z, TLA+, CSP, and Alloy. ProB's core, which also includes interpreters for various formalisms (including B), is implemented in SICStus Prolog [13]. Consequently, B models are interpreted during execution, animation and model checking.

ProB2-UI [5] is a JavaFX-based graphical user interface which has been developed on top of ProB by using ProB's Java API [39]. The following features of ProB2-UI are especially relevant for this work:

– the persistent storage and replay of traces [5],
– domain-specific VisB visualization [71],
– timed probabilistic SimB simulation [66] with user interaction [64].

ProB2-UI supports many techniques to create traces: animation (also via interactions in VisB), test case generation, model checking, or simulation with SimB. Once a trace is created, it is possible to add description text corresponding to each step. Later, a trace can be replayed (1) to check if the scenario is still feasible and (2) to perform additional checks on a trace. The corresponding description text helps to communicate with domain experts.

VisB [71] is a component of ProB to create interactive visualizations of formal models using SVG images and a glue file. The VisB glue file defines the main SVG image, as well as observers and click listeners that link the graphical elements with the model's state. Using VisB, a user can view the model's current state graphically, and execute operations by clicking on visual elements. An overview (also proposed in Fig. 1 of [71]) is shown in Fig. 1. Many features have been added in response to feedback from academic and industrial uses since VisB's original publication [71]. New features include iterators for groups of related SVG objects, multiple click events for SVG objects, dynamic SVG object creation, and SVG class manipulation for hovers. Furthermore, VisB's core has been re-implemented in Prolog and integrated into ProB's core. Thus, VisB can now be used from ProB's command-line interface directly (without ProB2-UI [5]).

SimB [66] is a simulator for formal models which is part of ProB2-UI [5]. Using SimB, a modeler can annotate a formal model with timing and probabilistic elements for simulation. These annotations take the form of an activation diagram that describes how events trigger each other with delays and probabilities. As a result, SimB helps to validate requirements with timing and probabilistic aspects. More recently, SimB has been extended by a feature called *interactive simulation*, which allows user interaction to trigger a system response in a real-time simulation [64]. Interactive simulation can also be linked to responding to manual interaction in VisB visualizations. This helps a user or domain experts to validate user requirements more easily, as user interaction and system reaction can be better understood. In general, SimB helps to create prototypes for formal models with timing, probabilistic, and interactive behavior, emulating real-world behavior. For example, in this work, we used a visualization of a user interface for a vehicle's lighting system [52].

B2Program [65] is a code generator for high-level B models, which targets Java, C++, Python, Rust [17], and also TypeScript/JavaScript now. Unlike other B code generators, the model does not have to be refined to B's low-level subset B0. Instead, B2Program enables code generation from a formal B model at various abstraction levels for validation and demonstration purposes. This also means that B2Program allows code generation for constructs such as set operation, set comprehensions, relation operations, non-determinism, etc. Consequently, B2Program cannot be used for embedded systems because memory consumption cannot be verified for these constructs, especially due to the use of external libraries. B2Program also supports code

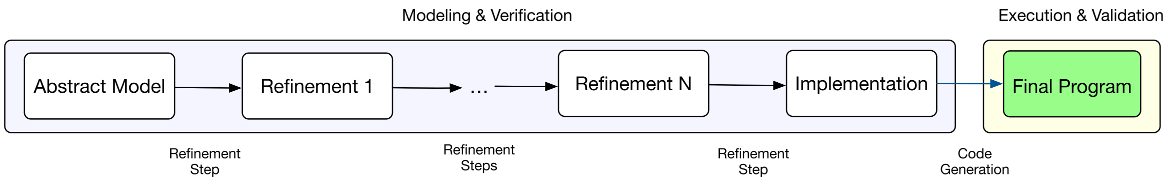
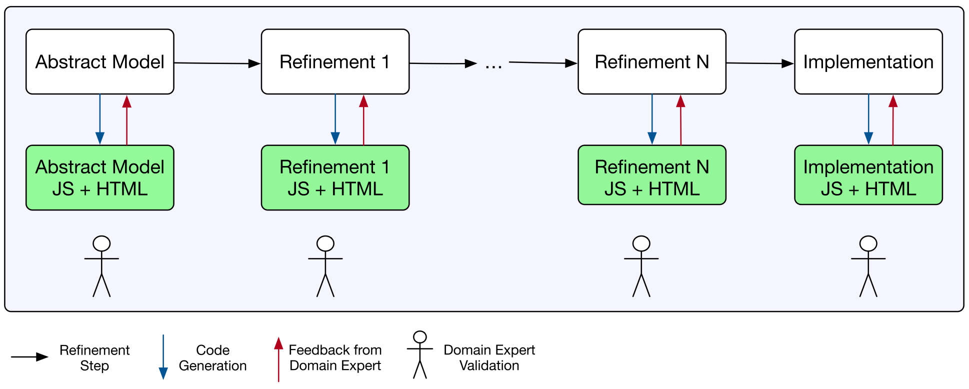


Fig. 2 Typical formal methods workflow with refinement; each refinement step adds more detail (represented by events, variables, etc.) to the previous abstraction level; the final refinement step refines the model to B0, from which low-level B0 code generators are applied, e.g., for usage in embedded systems (Color figure online)

Fig. 3 Workflow: code generation for validation with refinement; each refinement step adds more detail (represented by events, variables, etc.) to the previous abstraction level; code generation can be applied at each abstraction level for validation purposes; high-level constructs are supported for code generation, but memory usage cannot be verified and thus usage in embedded systems is not possible (Color figure online)



generation of specialized model checkers[1] for a machine [67]. The generated code for model checking builds the entire state space to check for invariant violations and deadlocks. To explore the complete state space, this code generates functions to compute all outgoing transitions and thus all succeeding states. Those functions are invoked for each explored state until they cover the complete state space. B2PROGRAM is implemented using the STRINGTEMPLATE [58] engine, which allows targeting multiple languages with a single code generator. This is achieved by mapping each construct to a template that is rendered to the target code.

3 Validation workflow

In the following, we compare the typical formal methods workflow with the one that is enabled in this research, i.e., by code generation for validation.

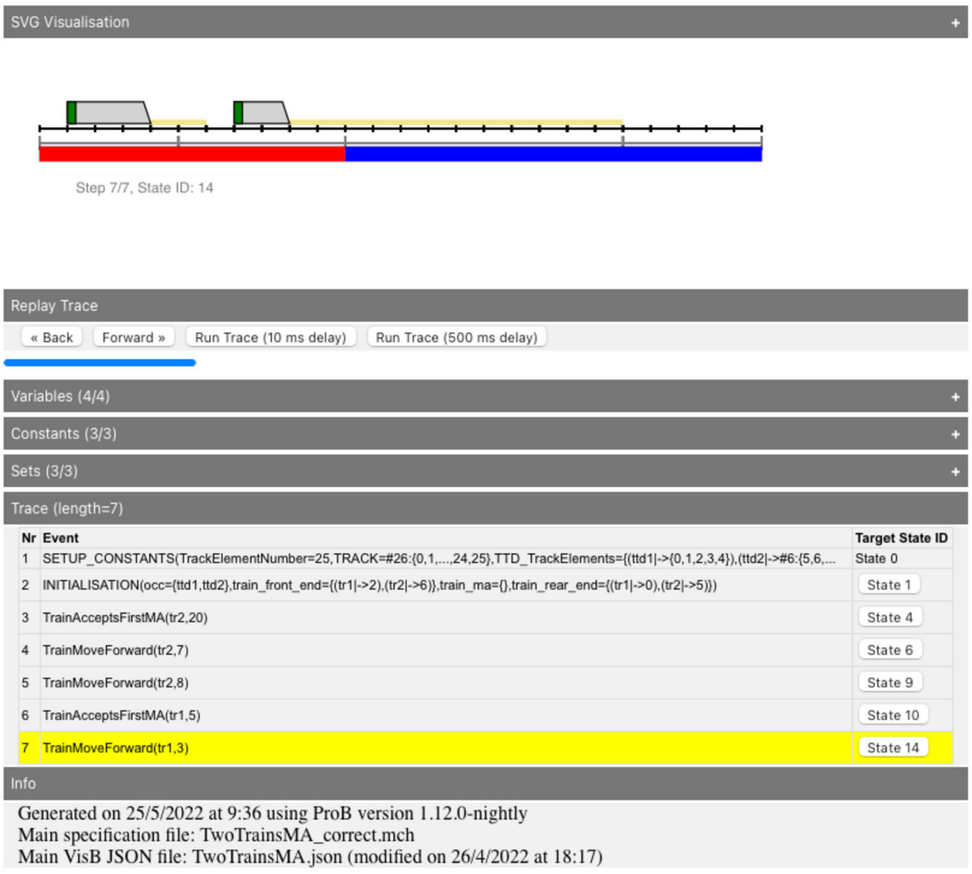
Figure 2 shows a typical formal method workflow with refinement: A system or software is modeled step-by-step until all requirements are encoded. Furthermore, the model is refined until reaching an implementable subset of the modeling language (e.g., B0 in the B method). Each development step of the model is verified by provers such as ATELIERB [15], or by model checkers such as PROB. After finishing the modeling process, a low-level code generator (e.g. an ATELIERB B0 code generator) is applied to generate the final program from a verified model.

A disadvantage of this typical workflow is that software is often validated too late during the development process, possibly after generating the final code. Figure 3 describes the approach followed by this work: We extend the high-level B code generator B2PROGRAM [65] by JavaScript generation and supporting validation techniques such as animation, trace replay, VISB visualizations, and SIMB simulations. In particular, an HTML document is generated, supporting early-stage validation with the aforementioned techniques by a domain expert. As a result, domain experts are integrated into the development process at an early stage.

While Fig. 3 is also feasible with existing animators like PROB, our approach enables communication via "interactive validation documents", where the model's formal aspects are hidden and no formal methods tool has to be installed by the domain expert.

As a simple example of a refinement-based development approach, let us consider a lift which is modeled as follows: the abstract level model's the lift's movement, the first refinement models the doors, and the second refinement introduces the lift's buttons. According to Fig. 2, a modeler can then refine further to B0 to apply code generation for the embedded system to be used in a real lift. Validation would then be

[1] Model checking is a technique that checks whether a system (modeled by a formal model) meets a certain specification, i.e., property. To do this, model checking computes all possible states and execution paths of the system. There are exhaustive approaches, in which the entire system is executed in all possible states, and symbolic approaches, which over-approximate the state space [4].



Nr	Event	Target State ID
1	SETUP_CONSTANTS(TrackElementNumber=25,TRACK=#26:{0,1,...,24,25},TTD_TrackElements={(ttd1|->{0,1,2,3,4}),(ttd2|->#6:{5,6,...	State 0
2	INITIALISATION(occ={ttd1,ttd2},train_front_end={(tr1|->2),(tr2|->6)},train_ma={},train_rear_end={(tr1|->0),(tr2|->5)})	State 1
3	TrainAcceptsFirstMA(tr2,20)	State 4
4	TrainMoveForward(tr2,7)	State 6
5	TrainMoveForward(tr2,8)	State 9
6	TrainAcceptsFirstMA(tr1,5)	State 10
7	TrainMoveForward(tr1,3)	State 14

Generated on 25/5/2022 at 9:36 using ProB version 1.12.0-nightly
Main specification file: TwoTrainsMA_correct.mch
Main VisB JSON file: TwoTrainsMA.json (modified on 26/4/2022 at 18:17)

Fig. 4 Static VisB export of trace from railway domain; static export consists of the domain-specific VisB visualization, the values of variables, constants, and sets, the trace, which consists of events + parameters that were executed, and metadata; case study shows two trains that are driving on the same track; no block must be occupied by more than one train (Color figure online)

applied at the final stage when code is already generated. Following our approach, as shown in Fig. 3, we can generate prototypes for validation (rather than embedded systems) for each refinement level when developing the lift. The purpose of code generation in this work is to enable us to check whether requirements have been implemented correctly in each, especially at earlier development stages. A domain expert can then already inspect whether the lift's movement is correctly modeled at the abstract level.

4 Static VisB HTML export

In this section, we present another new feature of VisB to export a trace as a standalone HTML file containing the visualization of the entire trace. This approach is supported by all formalisms in ProB. The trace can either be constructed interactively in the animator or automatically by other techniques such as test case generation, model checking, or simulation. The HTML file enables the user to navigate the trace and inspect the visualization of each state in the trace, without installing ProB. The model's variables and constant values are also accessible. Furthermore, the trace can be replayed automatically at different speeds. An example export can be seen in Fig. 4.[2]

This feature has been used for the communication of modelers with domain experts, e.g., in follow-on projects of the ETCS Hybrid Level 3 [33]. In particular, we (as modelers) animated traces which contain critical behavior. Those were traces we animated to validate important behaviors, or traces where we suspected errors. We then created static exports for these traces, and sent them to domain experts. The domain experts were then able to open them in the browser directly, and give feedback by email. With the dynamic export (later explained in Sect. 5), domain experts can provide feedback as description texts into the traces directly.

When exporting the trace to an HTML file, a JavaScript function is generated for each state, hard-coding the SVG objects' changed attributes. Listing 3 shows parts of the function that is generated for the state shown in Fig. 4. Focusing on the VisB item for the SVG object `occupied_ttd_polygon` (see Listing 2), one can see its hard-coded value for the state. When a domain expert steps through the trace, the visualization is updated according to the current state by executing the corresponding function. Figure 4 also contains meta-

[2] A more complex one is available at https://www3.hhu.de/stups/models/visb/train_4_POR_mch.html.

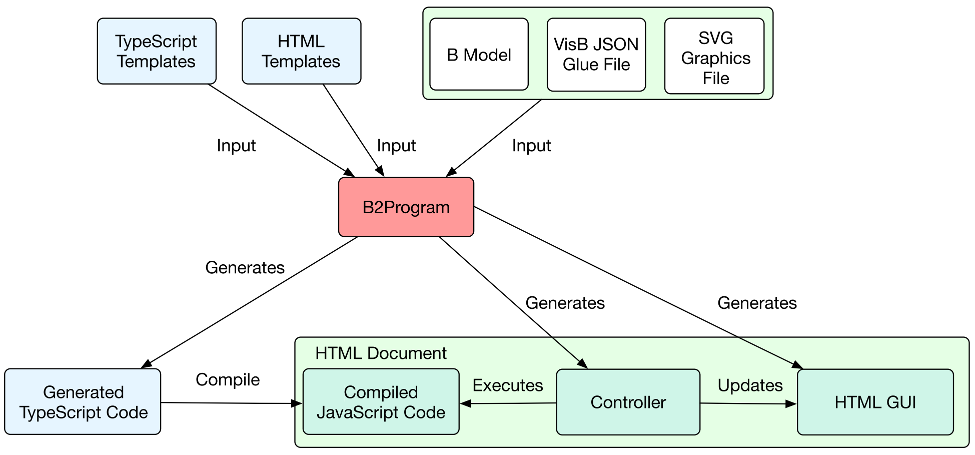


Fig. 5 Code generation from B model and VisB to HTML and JavaScript; templates are used as input to generate the TypeScript code for the B model, and the HTML GUI and its controller; the generated TypeScript code for the B model is compiled to JavaScript (Color figure online)

```
{
 "id":"occupied_ttd_polygon",
 "attr":"points",
 "value":"svg_set_polygon(OCC_TE,100.0/real(
      TrackElementNumber+1),100.0,2.0)"
}
```

Listing 2 VisB item for occupied section on track

```
function visualise14(stepNr) {
   setAttr("visb_debug_messages","text","Step "+
        stepNr+"/7, State ID: 14");
   setAttr("occupied_ttd_polygon","points","0.0,0
        0.0,2.0 42.30769230769231,2.0
        42.30769230769231,0 100.0,0");
   ...
   highlightRow(stepNr);
}
```

Listing 3 JavaScript function for visualizing a particular state in Fig. 4

information. Thus, a stored HTML trace is also a standalone snapshot of the model. One can later compare the stored visualization and variables with the current model.

5 Dynamic HTML export: code generation to HTML and JavaScript

Instead of generating a static HTML file consisting of a single trace, we now present a second approach that allows a domain expert to interact with the model. This approach is only supported for (a subset of) classical B. In this section, we explain the implementation of the dynamic export, which was the main work of this paper. For this, we use the model of a vehicle's light system by Leuschel et al. [52] which was modeled using the specification by Houdek and Raschke [35]. This model encodes a subset of requirements from the specification, which contains the ignition key, the pitman arm, and the vehicle's lighting system, consisting of the blinking lights and the hazard warning lights. Later in Sect. 6, we demonstrate how modelers and domain experts can work with the dynamic export for the lighting system.

Within the dynamic export, state values are computed in JavaScript dynamically. This makes it possible for a domain expert to explore alternative paths and not just the exported one. The dynamic export supports animation, domain-specific visualization in VisB, timed probabilistic simulation in SimB,and import/export of scenarios with descriptions. For the dynamic export, we also implemented model checking code generation (after [67]) with some features being used for animation and SimB. Those features contain functions for evaluating enabled transitions and functions to compute the invariant. The complete model checking algorithm is not accessible to the user, but is later used to evaluate the performance of animation (see Sect. 7).

Figure 5 shows the infrastructure for code generation to HTML and JavaScript. In addition to the B model, B2Program also expects the VisB glue file and the associated SVG visualization as input. To support JavaScript, we extend B2Program by TypeScript, following the approach described in our previous work [65]. Here, we decided not to generate JavaScript directly, but to generate TypeScript code as an intermediate step, which is then transpiled to JavaScript. We consider TypeScript as easier to debug than JavaScript, as there are fewer implicit type casts due to a stricter type system. Furthermore, many errors are already detected at compile time when transpiling from TypeScript to JavaScript (with more detailed error messages). Following the steps described in [65], we first implement TypeScript templates, and the B data types in TypeScript.

Listing 4 shows parts[3] of a TypeScript template which is used for code generation from the `INITIALISATION` clause. Generating code from the `INITIALISATION` clause shown in Listing 5 results in the TypeScript code shown in Listing 6.

[3] This part of the template is used for code generation without constants and copy constructor.

```
initialization(..., body, ...) ::= <<
...
constructor() {
    <body>
}
>>
```

Listing 4 Parts of TypeScript template for `INITIALISATION`

```
INITIALISATION
    hazardWarningSwitchOn := switch_off ||
    pitmanArmUpDown := Neutral ||
    keyState := KeyInsertedOnPosition ||
    engineOn := FALSE
```

Listing 5 INITIALISATION clause of sensors machine in Light System model

```
constructor() {
    this.hazardWarningSwitchOn = new SWITCH_STATUS(
        enum_SWITCH_STATUS.switch_off);
    this.pitmanArmUpDown = new PITMAN_POSITION(
        enum_PITMAN_POSITION.Neutral);
    this.keyState = new KEY_STATE(enum_SWITCH_STATUS.
        KeyInsertedOnPosition);
    this.engineOn = new BBoolean(false);
}
```

Listing 6 Generated TypeScript code from `INITIALISATION` clause of sensors machine shown in Listing 5

By using the StringTemplate engine in B2Program, we could utilize the majority of B2Program's implementation for TypeScript/JavaScript without additional extensions. The main effort was to implement the B data types including the B operators in TypeScript, which has to be done by a programmer manually. Those B data types include integers, booleans, strings, tuples, structs, sets, and relations together with their operators. It would also be possible to implement those data types in JavaScript directly, but due to the aforementioned reasons, we decided to implement them in TypeScript and then transpile to JavaScript.

In addition to TypeScript templates, we also implemented HTML templates from which the graphical user interface (GUI) is generated. B2Program also generates a controller for the GUI and the translated B model. The controller's task is to execute operations in the translated model, and to update the GUI based on the model's current state.

5.1 Validation by domain expert

In the following, we describe how a domain expert can work with the dynamic export to support the validation of formal models. The steps are illustrated in Fig. 6.

In the first step, a domain expert can run various scenarios to check whether the model behaves as desired, i.e., whether the requirements are fulfilled. With the dynamic export, a

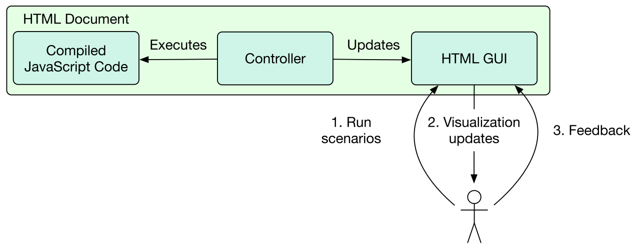


Fig. 6 Validation with HTML document by domain expert; steps consisting of running scenarios, inspecting visualization updates, and giving feedback

domain expert can run scenarios via animation, trace replay, or SimB simulation. Animation allows a domain expert to explore a new scenario and store it as a trace. It is also possible for a domain expert to re-play an existing trace. The trace can either be created by the domain expert itself or supplied by the modeler together with the dynamic export. SimB simulation makes it possible to run a scenario with timing, probabilistic, and interactive aspects. To achieve this, a modeler must encode a SimB simulation as an activation diagram (currently in a JSON representation). The challenge for domain expert here is also that one has to familiarize oneself with the syntax and semantics of SimB's activation diagrams to model them.

During the execution of a scenario, the VisB visualization might update, i.e., its appearance might change. A domain expert can then check from their perspective whether the system described by the underlying formal model behaves as desired.

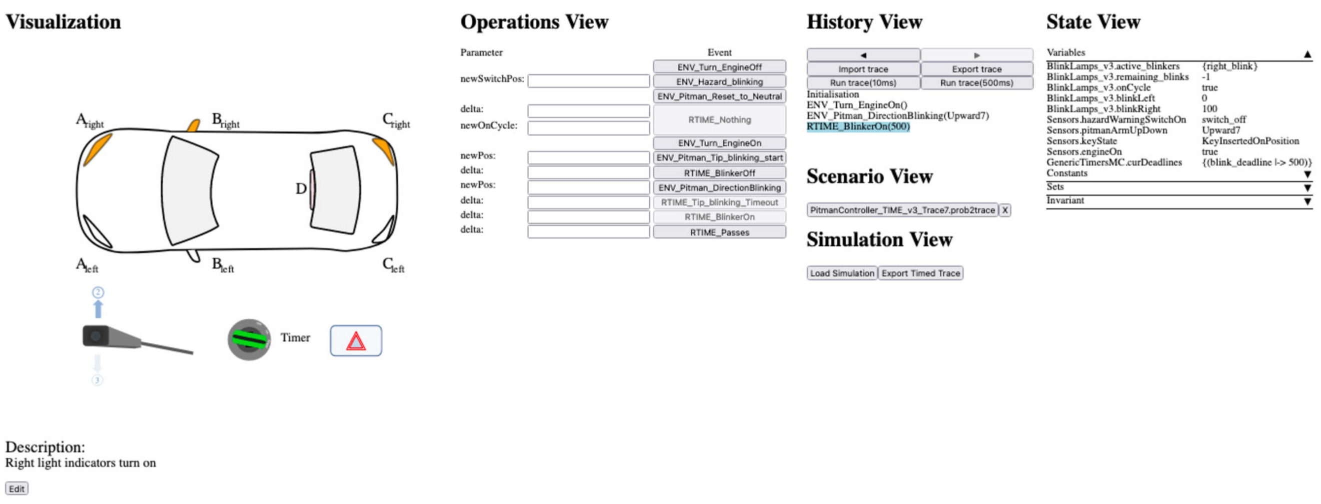
In the last step, a domain expert can give feedback to the modeler, e.g., by writing description text into the executed trace. The trace can finally be exported to a modeler who can load the trace in ProB2-UI.

In our design, we decided to generate code from the VisB visualization, while generating interpreters for replaying traces and running SimB simulations. Thus, the VisB visualization cannot be changed in the HTML document. In contrast, it is possible to customize SimB simulations and run various traces. We have decided on this design because a domain expert usually only has one specific view, but wants to run multiple scenarios. Thus, it would be too inflexible if it is only possible to run one SimB simulation.

5.2 Graphical user interface

In the following, we describe the GUI of the dynamic export generated from the Light System model (see Fig. 7).[4] The GUI is inspired by ProB2-UI [5] and consists of its main

[4] The example is also available at https://favu100.github.io/b2program/visualizations/LightModel/PitmanController_TIME_MC_v4.html.



Description:
Right light indicators turn on

Operations View

Parameter		Event
		ENV_Turn_EngineOff
newSwitchPos:		ENV_Hazard_blinking
		ENV_Pitman_Reset_to_Neutral
delta:		RTIME_Nothing
newOnCycle:		
		ENV_Turn_EngineOn
newPos:		ENV_Pitman_Tip_blinking_start
delta:		RTIME_BlinkerOff
newPos:		ENV_Pitman_DirectionBlinking
delta:		RTIME_Tip_blinking_Timeout
delta:		RTIME_BlinkerOn
delta:		RTIME_Passes

History View

Import trace | Export trace
Run trace(10ms) | Run trace(500ms)

Initialisation
ENV_Turn_EngineOn()
ENV_Pitman_DirectionBlinking(Upward7)
RTIME_BlinkerOn(500)

Scenario View

PitmanController_TIME_v3_Trace7.prob2trace X

Simulation View

Load Simulation | Export Timed Trace

State View

Variables	
BlinkLamps_v3.active_blinkers	{right_blink}
BlinkLamps_v3.remaining_blinks	-1
BlinkLamps_v3.onCycle	true
BlinkLamps_v3.blinkLeft	0
BlinkLamps_v3.blinkRight	100
Sensors.hazardWarningSwitchOn	switch_off
Sensors.pitmanArmUpDown	Upward7
Sensors.keyState	KeyInsertedOnPosition
Sensors.engineOn	true
GenericTimersMC.curDeadlines	{(blink_deadline |-> 500)}

Constants
Sets
Invariant

Fig. 7 Light System web GUI with domain-specific VisB visualization + description text, operation view, history view, scenario view, simulation view, and state view (Color figure online)

```
{
  "id":"A-right",
  "attr":"fill",
  "value":"IF right_blink : active_blinkers THEN
      IF blinkRight=lamp_off
      THEN \"#ffe6cc\" ELSE \"orange\" END
    ELSE \"white\" END"
}
```

Listing 7 Example of VisB item defining the color of the right indicator at the vehicle's front

```
_svg_vars["A-right"] = document.getElementById("
     LichtUebersicht_v4").contentDocument.
     getElementById("A-right")
_svg_vars["A-right"].setAttribute("fill",
   (_machine._BlinkLamps_v3._get_active_blinkers().
       elementOf(new DIRECTIONS(enum_DIRECTIONS.
       right_blink)).booleanValue() ?
     (_machine._BlinkLamps_v3._get_blinkRight().equal(
         _machine._BlinkLamps_v3._get_lamp_off())
         .booleanValue() ?
     new BString("#ffe6cc") : new BString("orange")) :
   new BString("white")).getValue());
```

Listing 8 JavaScript code generation from Listing 7

views. Besides describing the generation of the GUI, we will also focus on the challenges. Those challenges particularly occur when using B expressions dynamically, e.g., in the operations view or when loading traces or SimB simulations.

VisB view On the left-hand side of Fig. 7, one can see the domain-specific VisB visualization. Its features include (1) a graphical representation based on the model's current state, and (2) interaction with the model, i.e., executing an operation by clicking on a graphical object.

Listing 7 shows a VisB item defining an observer on the model's state. Particularly, this is a VisB item that defines the color of the right indicator at the vehicle's front. Assuming that the right blinks are active, the SVG object `A-right` should be filled in `#ffe6cc` (light orange) when the lamps are off, or `orange` when the lamps are on. When the right blinks are not active, `A-right` should be filled `white`.

As described in Sect. 4, values for the graphical objects' appearances are hard-coded in the static HTML export. To allow interactive animation, the visualization has to be updated based on the current state dynamically. For this purpose, the B expression is translated to JavaScript and is thus evaluated at runtime (and not statically hard-coded as described

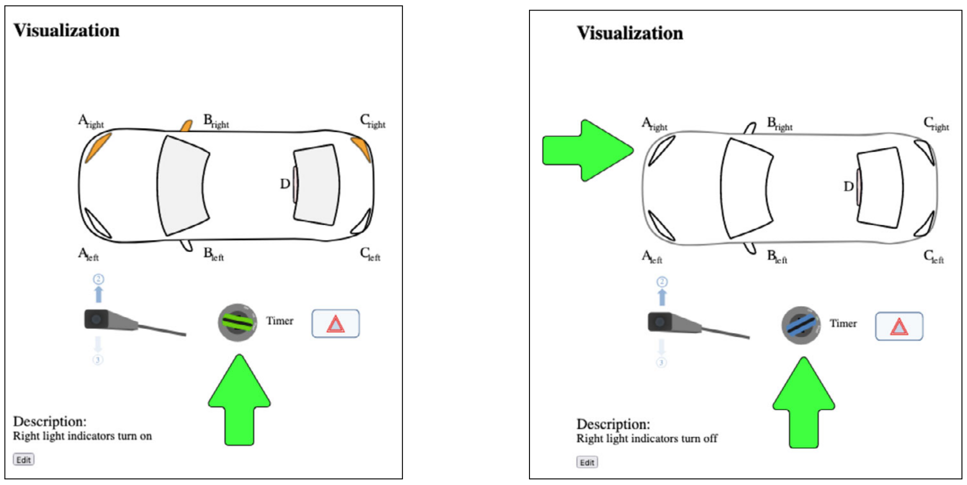
```
_svg_events["ENV_Turn_EngineOff"] = document.
     getElementById("LichtUebersicht_v4").
     contentDocument.getElementById("engine-start");
_svg_events["ENV_Turn_EngineOff"].onclick = function
     () {
    transition = _machine._tr_ENV_Turn_EngineOff();
    ... // Check whether transition is feasible
    var parameters = [];
    var returnValue = _machine.ENV_Turn_EngineOff(...
        parameters);
    ... // Update views and internals
}
```

Listing 9 JavaScript code generation from `{"id":"engine-start", "event":"ENV_Turn_EngineOff"}`

in Sect. 4). The code generated for Listing 7 is shown in Listing 8.

For a VisB event, B2Program generates a click listener on the SVG object which checks whether the corresponding B event is enabled, and executes it afterward. This makes it possible to interact with the model by clicking on the graphical element. `{"id":"engine-start", "event":"ENV_Turn_EngineOff"}` defines a click event

Fig. 8 VisB visualization (+ description texts) of Light System with pitman arm, ignition key, and warning lights button; green arrows point to corresponding elements in sub-captions (Color figure online)



(a) Light System with Right Light Indicators turned on

(b) Light System with Right Light Indicators turned off after turning off engine

on `engine-start`, triggering the `ENV_Turn_EngineOff` event. The generated code is shown in Listing 9.

The VisB view also includes a text area which allows a domain expert to provide feedback for the executed transition, and describe the current state. This description is saved when exporting the trace (see history view). It can be used by a modeler or another domain expert as valuable feedback. Description texts are also important as one might not see changes immediately when the current state changes.

Figure 8 shows the VisB visualization for the Light System model. In Fig. 8a, one can see that the right light indicators are active, i.e., they are orange, after executing the generated code shown in Listing 8. When the user presses the engine button (`engine-start`; also directed by the green arrow) in the state shown in Fig. 8a, the engine and the right light indicators turn off (see Fig. 8b).

Operations view The operations view allows the execution of operations as an alternative to the VisB view.

Within the operations view, B2Program generates functionalities of an animator, i.e., a user receives suggestions as to which operations (with which parameters) are enabled. In particular, B2Program generates a button for each operation in the formal model, and a text field for each corresponding parameter. Each button is enabled exactly when the operation is enabled; otherwise, the button is enabled. Furthermore, the text fields store a list of options for parameters the operation is feasible for execution. To compute the operations' enabledness and the feasible parameters, B2Program uses functions to compute enabled transitions which are originally generated for model checking [67]. The computation is done when reaching another state in the model, and works as follows:

1. Invoke the function to compute outgoing transitions for the operation.

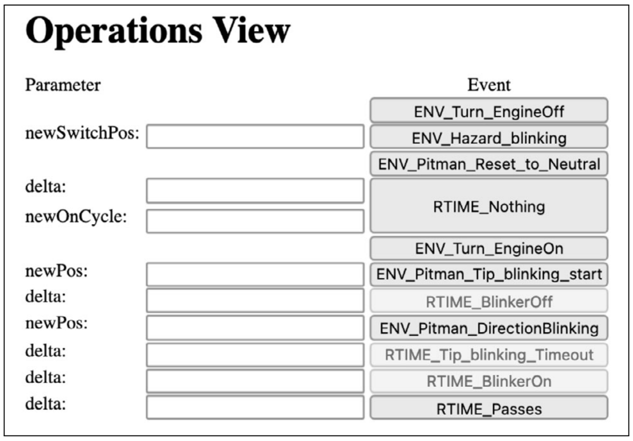
```
_machine_events["ENV_Turn_EngineOff"] = document.
      createElement("button");
_machine_events["ENV_Turn_EngineOff"].onclick =
      function() {
    transition = _machine._tr_ENV_Turn_EngineOff();
    ... // Check whether transition is feasible
    var parameters = [];
    var returnValue = _machine.ENV_Turn_EngineOff(...
          parameters);
    ... // Update views and internals
}
```

Listing 10 JavaScript code generation for button to execute `ENV_Turn_EngineOff` event

2. There are outgoing transitions, i.e., the function returns `true` or a non-empty set of tuples. Then check the inner guards by trying to execute the computed transitions. Enable the button for the operation, and use the set of tuples to fill the list of possible options for parameters.
3. Otherwise, there will be no outgoing transition, i.e., the function returns `false` or an empty set. Then disable the button for the operation. Clear the list of possible options for parameters.

Code generation for executing an operation via the operations view is done similarly to the VisB events (see Listing 10). To achieve better user-friendliness, the user does not have to explicitly specify the parameters here; by default, the first possible combination of parameters is used for execution.

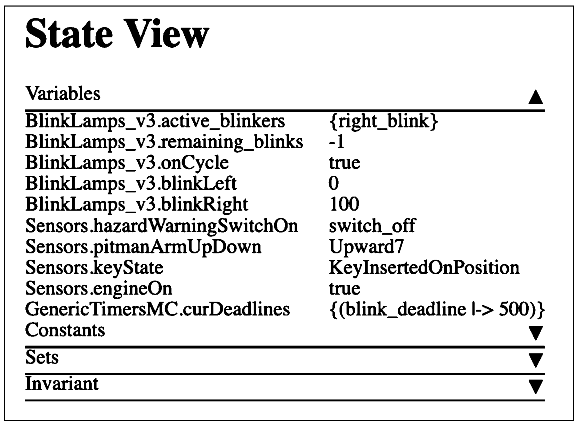
Since values for parameters are entered dynamically in the operations view, we encountered a problem that these values have to be evaluated. Compared to animators like ProB, we only allow constant values, e.g., 1, TRUE, or red (whereas red is a set element). These constant values still have to be parsed in a lightweight manner, in order to transform them from a string representation to a feasible representa-



Operations View

Parameter		Event
		ENV_Turn_EngineOff
newSwitchPos:		ENV_Hazard_blinking
		ENV_Pitman_Reset_to_Neutral
delta:		RTIME_Nothing
newOnCycle:		
		ENV_Turn_EngineOn
newPos:		ENV_Pitman_Tip_blinking_start
delta:		RTIME_BlinkerOff
newPos:		ENV_Pitman_DirectionBlinking
delta:		RTIME_Tip_blinking_Timeout
delta:		RTIME_BlinkerOn
delta:		RTIME_Passes

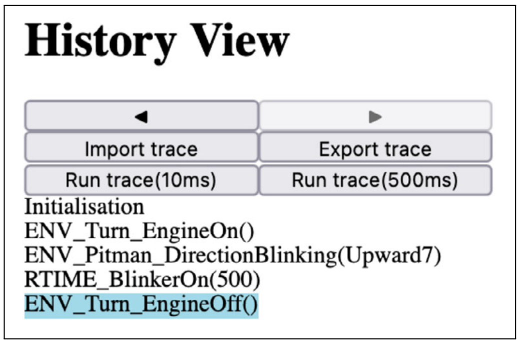
Fig. 9 Example: operations view for Light System



State View

Variables ▲	
BlinkLamps_v3.active_blinkers	{right_blink}
BlinkLamps_v3.remaining_blinks	-1
BlinkLamps_v3.onCycle	true
BlinkLamps_v3.blinkLeft	0
BlinkLamps_v3.blinkRight	100
Sensors.hazardWarningSwitchOn	switch_off
Sensors.pitmanArmUpDown	Upward7
Sensors.keyState	KeyInsertedOnPosition
Sensors.engineOn	true
GenericTimersMC.curDeadlines	{(blink_deadline |-> 500)}
Constants ▼	
Sets ▼	
Invariant ▼	

Fig. 10 Example: state view for Light System



History View

◀ ▶
Import trace Export trace
Run trace(10ms) Run trace(500ms)

Initialisation
ENV_Turn_EngineOn()
ENV_Pitman_DirectionBlinking(Upward7)
RTIME_BlinkerOn(500)
ENV_Turn_EngineOff()

Fig. 11 Example: history view for Light System

tion in B2Program. For example, 1 is transformed to `new BInteger(1)`. If we were to try to allow expressions in general, e.g., 1+1+a, it would be necessary to embed a complete B parser and evaluator for expressions. This would contradict the idea of code generation; thus, we decided to implement it in a lightweight manner only.

An example of the operations view in the dynamic export is shown in Fig. 9. This operations view shows all buttons and text fields for the Light System model, including the button for `ENV_Turn_EngineOff`, which was discussed before.

State view Within the state view, one can view the model's current state in mathematical notation. Although mathematical notations are difficult for a domain expert to understand, it can still be important to debug the model. We display the set, variable, constant, and invariant sections textually. For better readability, B2Program splits the invariant into its conjuncts. This feature was also implemented for model checking to achieve better performance [67].

Figure 10 shows an example of the state view for the Light System model. This state corresponds to the one shown in Fig. 8a.

History view The history view shows the currently animated trace. When executing an operation (via the operations view or by clicking inside the VisB visualization), the corresponding transition with input/output parameters is displayed in the history view. At the same time, this transition is saved in a list together with the model's state. By default, an empty description text is created, which can be modified by a domain expert in the VisB view. Those data are used to generate a ProB2-UI trace.

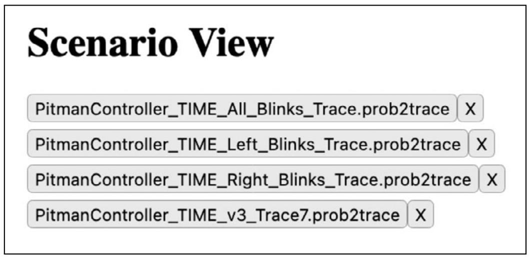
Within the history view, there are buttons to import, and export an animated trace represented in ProB2-UI's format. In Sect. 6, we demonstrate how this, together with the scenario view, improves communication between modelers and domain experts.

Figure 11 shows an example of the history view for the Light System model which contains an animated trace. This trace could have been animated via the VisB view or the operations view by hand, or loaded via the `Import trace` button.

Scenario view Within the scenario view, a domain expert can store a set of traces. Each trace is stored in a ProB2-UI trace file. Replaying a trace is done by iterating over its transitions; the ProB2-UI trace files contain for each transition the operation name and parameter values. Using the other views described above, a domain expert can then step through the scenario, check whether the system behaves as desired, and add a description text.

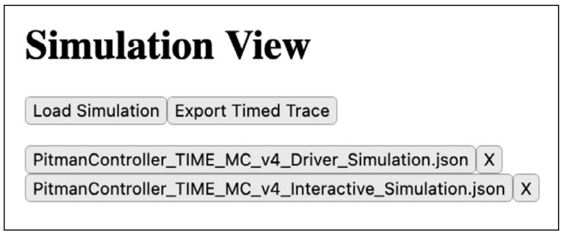
Figure 12 shows an example of the scenario view for the Light System model. The scenario view shows a set of traces that have been loaded via the `Import trace` button in the history view. By clicking on such a trace, it is loaded into the history view and set as the currently animated trace.

Simulation view The simulation view enables a user to perform real-time simulations using SimB files that can be loaded in the simulation view. As mentioned in Sect. 2,


Scenario View

PitmanController_TIME_All_Blinks_Trace.prob2trace X
PitmanController_TIME_Left_Blinks_Trace.prob2trace X
PitmanController_TIME_Right_Blinks_Trace.prob2trace X
PitmanController_TIME_v3_Trace7.prob2trace X

Fig. 12 Example: scenario view for Light System with a set of traces


Simulation View

Load Simulation | Export Timed Trace

PitmanController_TIME_MC_v4_Driver_Simulation.json X
PitmanController_TIME_MC_v4_Interactive_Simulation.json X

Fig. 13 Example: simulation view for Light System

SimB is based on activation diagrams describing how events activate each other with timing and probabilistic behavior.

Our generated Javascript code re-implements the SimB algorithm as described by Vu et al. [66]. There, activations of events are managed in a scheduling table which stores the times until the next activation. In particular, our implementation performs the following steps [66]:

1. Let time pass until reaching the next scheduled activations;
2. Update the activation times in the scheduling table;
3. Iterate over the activations in order of their priority
   - If the activation's time has expired then,
     (a) Execute the operation if it is enabled,
     (b) Activate SimB activations that are triggered by this activation,
     (c) Remove the activation from the scheduling table,
   - Otherwise, ignore the activation;
4. Compute the (minimal) time until the next activations shall be executed;
5. Specifically for the GUI: Update all views.

Both steps item 3a and item 3b can also take into account probabilistic behavior. SimB's interactive simulation [64] is also supported in the dynamic export, i.e., allowing a user to trigger additional events.

To support SimB in the dynamic export, we considered two options:

1. Either, to generate code for a specific SimB simulation, similar to the B model and the VisB visualization.
2. Or to provide a SimB interpreter which allows loading several SimB simulations.

Here, we decided to implement the second possibility, to allow a domain expert to load several simulations, and not only specific ones. However, to support the full power of SimB activation diagrams, we would have to provide an evaluator for general B expressions. For now, we thus only allow constant values for those B expressions. In the future, we could reconsider pursuing the first option, i.e., to generate code for SimB activation diagrams. But as mentioned above, this means that the user can only choose from specific SimB simulations that were generated together with the B model.

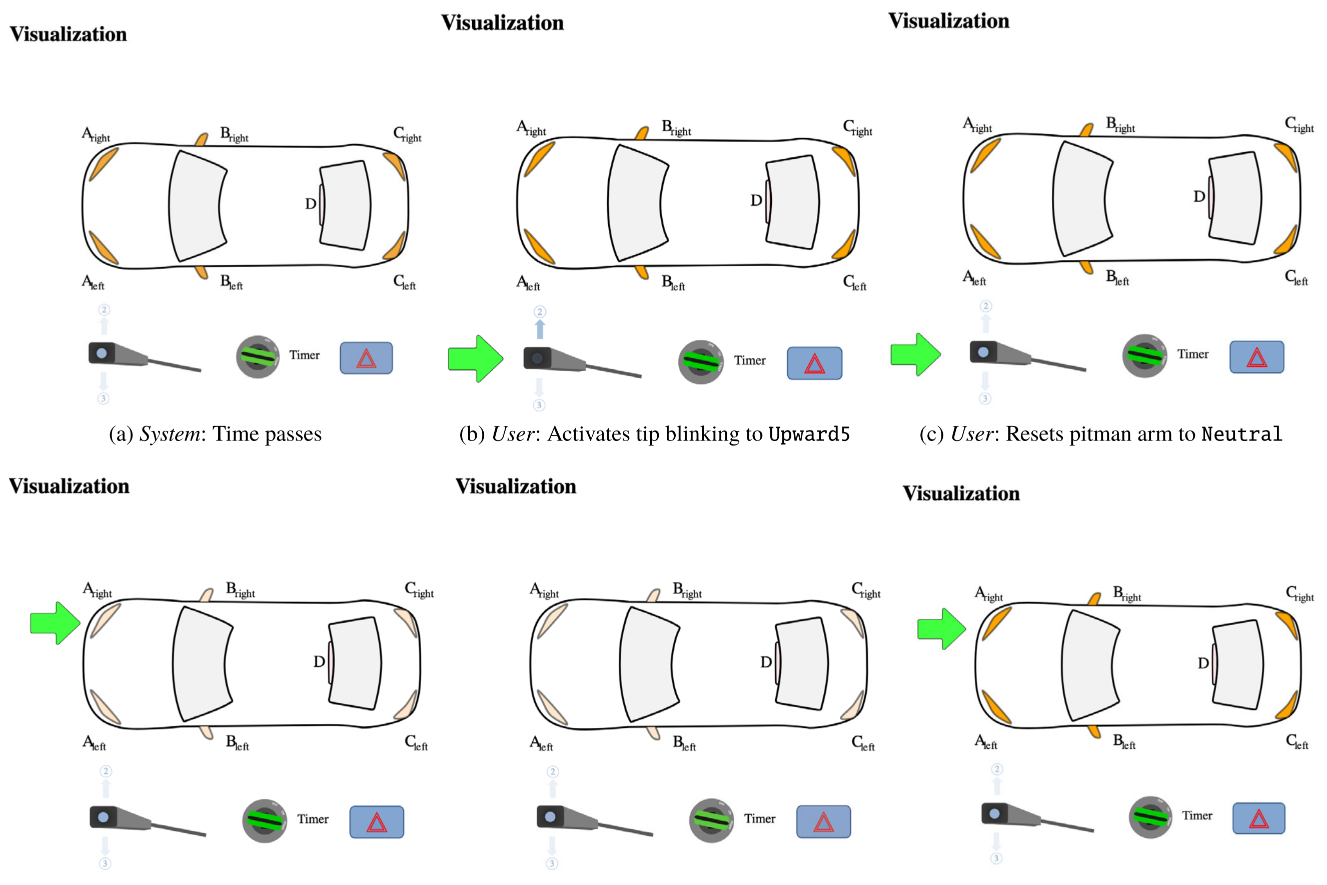
Figure 13 shows an example of the simulation view of the Light System. There, a modeler has loaded two simulations, one, which simulates the driver's and the vehicle's behaviors automatically, and a second one which simulates the vehicle's behaviors as a reaction to the driver's input (which has to be operated manually). In the dynamic export, it is also possible to export a simulated trace with timing behavior. This is called a *timed trace*, and corresponds to SimB's representation of an activation diagram where within the trace each event triggers the next one with time elapsing in between. Note that timed traces only contain constant values in their activation diagram; thus, a domain expert can re-play all timed traces with the SimB interpreter.

6 Case studies

This section demonstrates how this work (1) makes it possible for a domain expert to validate requirements, and (2) improves communication between the modelers and the domain experts. We will study two case studies: a vehicle lighting system from the automotive domain [52], and a landing gear from the aviation domain [44]. On the one hand, this section shows that our approach applies to different domains. On the other hand, the first case study focuses on the communication between domain experts and modelers, while the second case study focuses on the communication between domain experts with different perspectives.

Vehicle light system For the light system case study, domain experts provide a set of validation sequences (a.k.a. scenarios). The dynamic export allows a domain expert to run scenarios directly and then communicate with modelers afterward. In the following, we focus on Sequence 7, which is given in the specification [35]. Sequence 7 validates the behavior of the turn indicator and the hazard light. In particular, events for tip blinking, direction blinking, and the hazard warning lights are executed, and the desired behavior is checked afterward.

Figure 14 shows parts of Sequence 7 as domain-specific visualizations in the dynamic export. First, a modeler animates a trace in ProB2-UI to validate Sequence 7 (see



(a) *System*: Time passes

(b) *User*: Activates tip blinking to `Upward5`

(c) *User*: Resets pitman arm to `Neutral`

(d) *System*: All light indicators turn off

(e) *System*: Timeout for tip blinking with `Upward5`; no changes in visualization

(f) *System*: All lights indicators turn on

Fig. 14 Domain-specific visualization of states after executing (a) – (f) in Fig. 16; green arrows show changes compared to previous state (Color figure online)

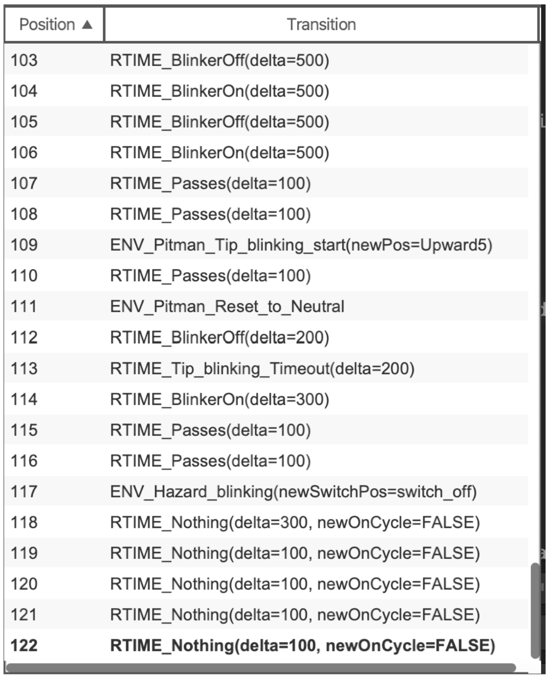
Figure 15). The sequence's feasibility in the model has already been shown by Leuschel et al. in [52]. Based on this sequence, we outline how our approach helps to improve communication between modelers and domain experts.

To ensure that the modeler has not misunderstood the requirements, they can then export the trace to a domain expert who could load this trace into the generated HTML document (see Fig. 16). The domain expert can then inspect whether the correct behavior was indeed implemented by the modeler.

A critical point in the sequence is to validate that "if the warning light is activated, any tip-blinking will be ignored or stopped if it was started before" (requirement ELS-13 in [35]). This part of the animation is shown in steps (a) – (f) in Fig. 16, which corresponds to Fig. 14. With the help of the domain-specific visualization (see Fig. 14), the domain expert can easily approve that the desired behavior has indeed been implemented. For example, in step (f) of Fig. 16, a domain expert can check the aforementioned behavior, and add or modify a description text for a modeler (see Fig. 17).

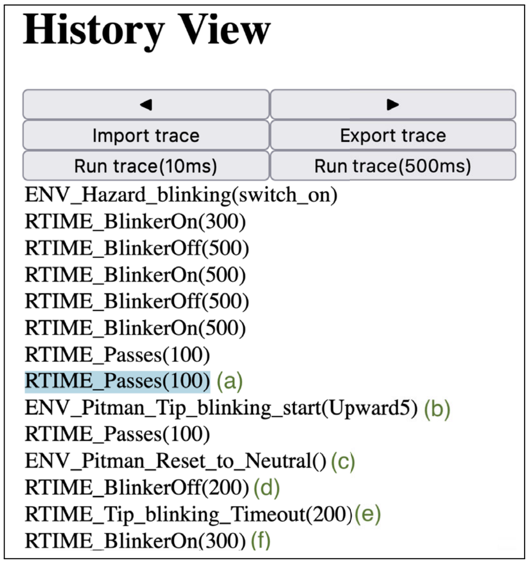
Furthermore, the dynamic export allows a domain expert to inspect alternate paths for the same requirement, thereby establishing a stronger guarantee of whether a requirement is fulfilled. This process is supported by the new SIMB implementation in this work. In particular, a user/domain expert can simulate user inputs and the vehicle's system reactions with timing and probabilistic behavior automatically. For more precise control over the user inputs, one can also use interactive simulation. Here, user input is applied manually, while the system reaction afterward is simulated automatically. For instance, a user/domain expert could execute user interactions in Fig. 14 (marked as *User*), whereafter the system's reaction (marked as *System*) could be observed with delay.

In our previous work [64], we already validated other requirements about the light system (in particular ELS-1, ELS-8, and ELS-12) by executing the user interaction, and observing the system's reaction afterward. Those validations could now also be done by a user or domain expert via the dynamic export of this work; and not only via PROB2-UI.



Position ▲	Transition
103	RTIME_BlinkerOff(delta=500)
104	RTIME_BlinkerOn(delta=500)
105	RTIME_BlinkerOff(delta=500)
106	RTIME_BlinkerOn(delta=500)
107	RTIME_Passes(delta=100)
108	RTIME_Passes(delta=100)
109	ENV_Pitman_Tip_blinking_start(newPos=Upward5)
110	RTIME_Passes(delta=100)
111	ENV_Pitman_Reset_to_Neutral
112	RTIME_BlinkerOff(delta=200)
113	RTIME_Tip_blinking_Timeout(delta=200)
114	RTIME_BlinkerOn(delta=300)
115	RTIME_Passes(delta=100)
116	RTIME_Passes(delta=100)
117	ENV_Hazard_blinking(newSwitchPos=switch_off)
118	RTIME_Nothing(delta=300, newOnCycle=FALSE)
119	RTIME_Nothing(delta=100, newOnCycle=FALSE)
120	RTIME_Nothing(delta=100, newOnCycle=FALSE)
121	RTIME_Nothing(delta=100, newOnCycle=FALSE)
122	**RTIME_Nothing(delta=100, newOnCycle=FALSE)**

Fig. 15 Parts of Sequence 7 in the history view of ProB2-UI



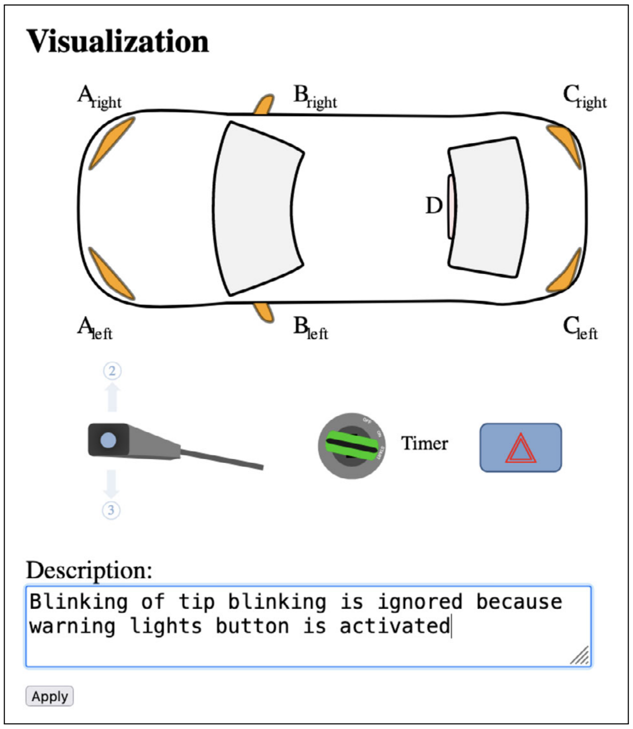
History View

◀ ▶
Import trace Export trace
Run trace(10ms) Run trace(500ms)

ENV_Hazard_blinking(switch_on)
RTIME_BlinkerOn(300)
RTIME_BlinkerOff(500)
RTIME_BlinkerOn(500)
RTIME_BlinkerOff(500)
RTIME_BlinkerOn(500)
RTIME_Passes(100)
RTIME_Passes(100) (a)
ENV_Pitman_Tip_blinking_start(Upward5) (b)
RTIME_Passes(100)
ENV_Pitman_Reset_to_Neutral() (c)
RTIME_BlinkerOff(200) (d)
RTIME_Tip_blinking_Timeout(200) (e)
RTIME_BlinkerOn(300) (f)

Fig. 16 Parts of Sequence 7 in the history view of the interactive validation document; (a) – (f) added manually; (a) – (f) corresponds to steps 108 – 114 in Fig. 15

Regarding the SimB features in the dynamic export, one can also export (timed) traces to a modeler again. As mentioned in Sect. 5.2, a timed trace is a special case of a SimB simulation. Consequently, a domain expert can export a timed trace which can then be re-played by a modeler in real-time.



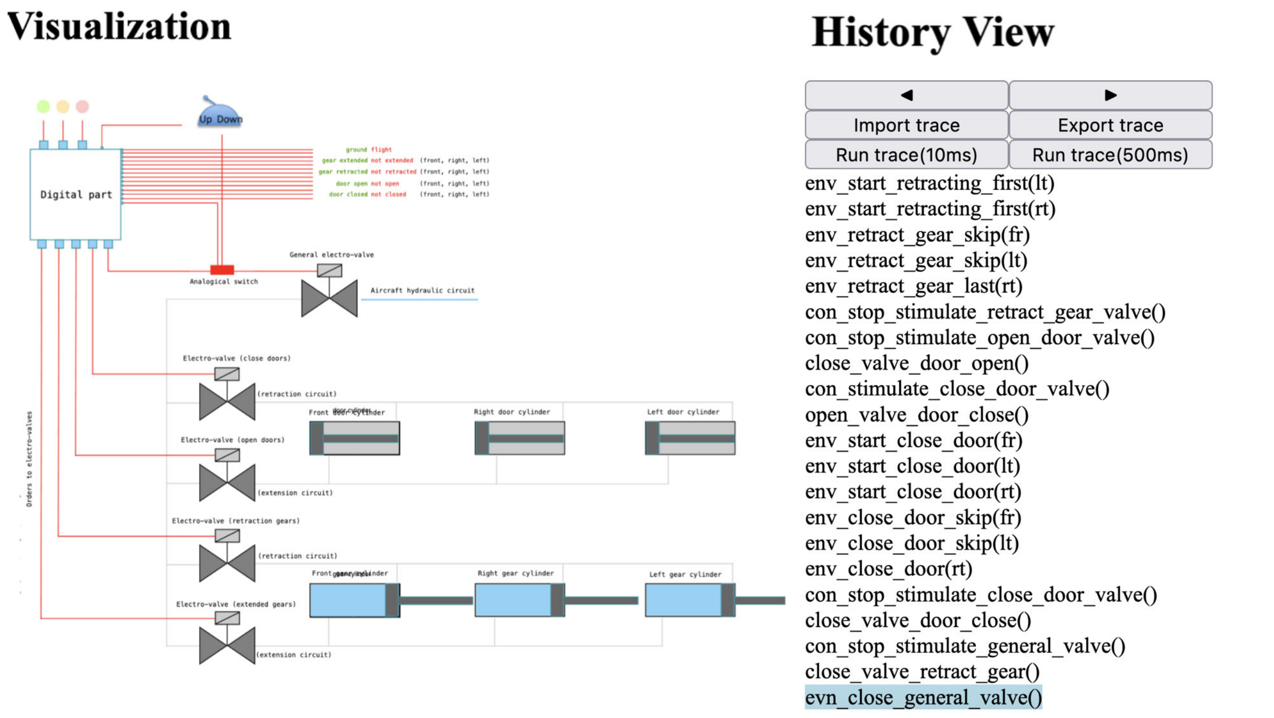
```
Blinking of tip blinking is ignored because
warning lights button is activated
```

Fig. 17 Modifying description for step (f) in Fig. 16 (Color figure online)

Landing gear The landing gear model [44] by Ladenberger et al. is modeled based on the specification by Boniol [10]. For the demonstration, we use the refinement level, which includes gears, doors, handles, switches, and electro-valves. To be able to use B2Program, we have manually translated the Event-B model to classical B. Figure 18 shows parts of the generated GUI from the landing gear model, which contains the VisB view and the history view. The domain-specific VisB view shows a hydraulic circuit consisting of the handle, the switch, the electro-valves, and the cylinders.

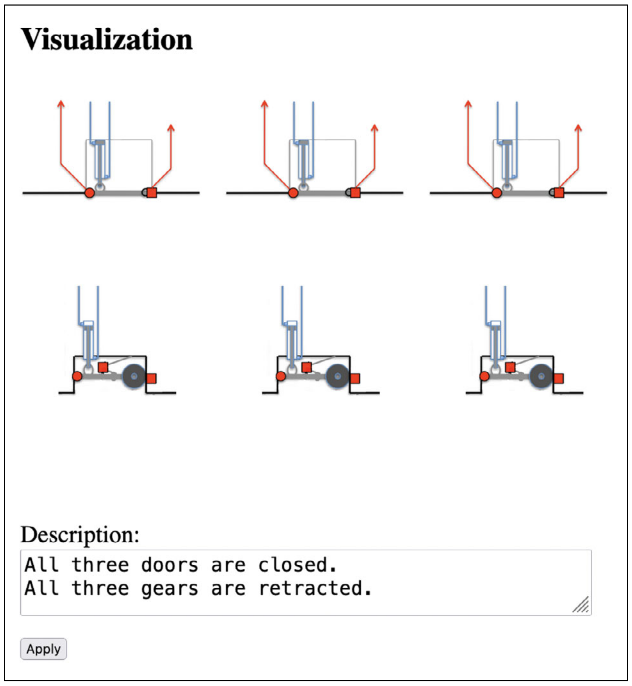
Using the operations view (which we omitted here due to space reasons), a domain expert can animate traces representing desired requirements. In this example, the domain expert has animated the retraction sequence from the specification. This trace can then be exported for ProB2-UI, to be used by a modeler. It can also be converted for use by another domain expert more focused on other aspects of the model. For instance, Fig. 19 shows an alternate domain-specific visualization with gears and doors, and description text provided by the first domain expert. The second domain expert can import the trace created from Fig. 18. Comparing Fig. 18 and Fig. 19, one can see that a pressurized door cylinder is equivalent to a closed door, and an unpressurized gear cylinder is equivalent to a retracted gear.[5] Thus, our approach does

[5] A partially pressurized door cylinder is equivalent to a moving door. An unpressurized door cylinder is equivalent to an opened door.



env_start_retracting_first(lt)
env_start_retracting_first(rt)
env_retract_gear_skip(fr)
env_retract_gear_skip(lt)
env_retract_gear_last(rt)
con_stop_stimulate_retract_gear_valve()
con_stop_stimulate_open_door_valve()
close_valve_door_open()
con_stimulate_close_door_valve()
open_valve_door_close()
env_start_close_door(fr)
env_start_close_door(lt)
env_start_close_door(rt)
env_close_door_skip(fr)
env_close_door_skip(lt)
env_close_door(rt)
con_stop_stimulate_close_door_valve()
close_valve_door_close()
con_stop_stimulate_general_valve()
close_valve_retract_gear()
evn_close_general_valve()

Fig. 18 Retraction sequence (also shown in history view) with hydraulic circuit as domain-specific visualization; hydraulic circuit contains the handle, the switch, the electro-valves, and the cylinders



Description:

```
All three doors are closed.
All three gears are retracted.
```

Fig. 19 Retraction sequence with gears and doors as domain-specific visualization, and description text

not only improve communication between modelers and domain experts, but also between domain experts from different perspectives.

To achieve a more realistic user interaction, a modeler can provide interactive timed SimB simulations to both domain experts. Both domain experts can then push up or push down the pilot's handle manually, and check whether the respective retraction sequence or outgoing sequence is executed within 15 seconds automatically (R_{11} and R_{12} from the specification [44]). In comparison to animation, users simply need to perform user actions, allowing them to experience and validate timing properties.

7 Applicability of JavaScript code generation

Another important aspect is the applicability of JavaScript code generation. In this section, we focus on the limitations and the performance.

Limitations As the JavaScript code generator is based on B2Program, it shares the same restrictions that are discussed in [65, 67].

B2Program has strong restrictions for quantified constructs. Indeed, for bounded variables $a_1 \ldots a_n$ constrained by a predicate P, the first n conjuncts of P must constrain

A pressurized gear cylinder is equivalent to an extended gear. A partially pressurized gear cylinder is equivalent to a moving gear.

the bounded variables in the exact order they are defined [65]. As discussed in [67], we plan to loosen this restriction in future, e.g., by allowing pruning predicates to reduce the enumeration size. Currently, B2Program iterates over all possible values before other predicates are formulated. The conjuncts in P can take the following form and are treated as follows [65]:

- $a = E$ is translated by assigning the value of E to the bound variable a.
- $a \in S$ is translated to a for loop where a is constrained while iterating over the set S.
- $a \subset S$, $a \subseteq S$ are translated to a for loop where a is constrained while iterating over the (strict) superset of S.

B2Program also forbids set operations on infinite sets, or storing them in variables [65]. We do not plan to support all operations on infinite sets, as some might require embedding a constraint solver, against which we decided to do [65].

B2Program chooses just one execution path for non-deterministic constructs such as `ANY`, `CHOICE`, or non-deterministic assignments [65]. Thus, models with those constructs cannot be animated exhaustively. For precise animation and model checking of `ANY`, `CHOICE`, and non-deterministic assignments, it is necessary to compute all choice points. Regarding animation, the user requires more control over the desired choice point for execution, while for model checking, it is necessary to cover all choice points.

An `ANY` substitution is of the form:

ANY $v_1,\ldots,v_n$ WHERE *predicate* THEN *substitutions* END

This means that for (local) variables $v_1,\ldots,v_n$ where *predicate* is true, execute *substitutions*. Otherwise, the entire operation in which the `ANY` substitution is used is not executable. As the *predicate* might constraint multiple values for $v_1,\ldots,v_n$, this substitution is non-deterministic.

A `CHOICE` substitution is of the form:

CHOICE *substitutions1* OR *substitutions2* END

This means that either *substitutions1* or *substitutions2* is executed. As there are two possibilities to choose from, this substitution is non-deterministic.

B2Program only allows top-level `PRE` and `SELECT` as non-determinism [67] for model checking. Inner guards, e.g., inner `SELECT`s, cause problems when calculating enabled transitions (discussed in [67]). Regarding animation in this work, a superset of possible transitions is computed from the top-level guards first; inner guards are checked during execution of the transition. To support inner `PRE` and `SELECT` for model checking, it would be necessary to adapt the algorithm slightly so that precondition violations are detected, and a state is discarded when an inner guard is not true. This is already the case for animation.

A `SELECT` substitution is of the form:

SELECT *guard* THEN *substitutions* END

This means that, when the *guard* is true, then execute *substitutions*. Otherwise, the entire operation in which the `SELECT` substitution is used is not executable. Furthermore, this substitution is often used at the top level to constrain possible values for the parameters.

A `PRE` (precondition) substitution has the form:

PRE *predicate* THEN *substitutions* END

This means that if the *predicate* is true, then execute *substitutions*. Otherwise, the entire operation in which the `PRE` substitution is used will lead to a *precondition violation*. Top-level `PRE` substitutions behave similarly to `SELECT`. `PRE` substitutions are also often used at the top level to constrain possible values for the parameters.

In conclusion, some models must be rewritten according to these rules; still, there are also models where it is not possible. Note that B2Program supports a significantly larger subset than B0 code generators. So, B2Program can be used at an early development stage; but especially at a very early stage, some models are too high-level for B2Program. One must then refine the model further to enable B2Program for validation, or use the static export from Sect. 4.

Performance In the previous work [65], we already compared Java and C++ code generation with ProB. To achieve good performance, we implemented the B data types `BSet` and `BRelation` using persistent data structures (similar to Java and C++, see [65]). For this, we used the Immutable[6] Javascript library, which also uses the structural sharing [3]. Furthermore, we used primitive integers in the generated Java, JavaScript, and C++ code here. As already explained in Sect. 5, code generation is similar to the one for the previously supported languages Java and C++: we adapted the existing templates for TypeScript, but still use the same implementation for generating code. Afterward, the generated TypeScript code for the B model is transpiled to JavaScript. Below we investigate how much the change of target language and libraries affects the performance of the generated code.

We have benchmarked the models from [65] and [67] for ProB,[7] Java,[8] and C++[9] and compared them with

[6] https://immutable-js.com/.

[7] ProB CLI 1.12.2-nightly built with SICStus 4.8.0 (arm64-darwin-20.1.0).

[8] OpenJDK 64-Bit Server VM (build 18.0.2+0, mixed mode, sharing).

[9] Compiled with clang, version 13.0.0 (clang-1300.0.29.30); `-O1` for model checking benchmarks (`-O2` was not optimized further for model checking [67]), `-O2` for simulation benchmarks.

Table 1 Simulation runtimes (ProB, generated Java, generated JavaScript, and generated C++ code) in seconds with number of operation calls (op calls), speed-up relative to ProB; models from [65] were re-benchmarked for ProB, Java, and C++ with a different device

		ProB	Java	JavaScript	C++
Lift (2×10^9 op calls)	Runtime	> 3600	5.85	13.86	0.07
	Speed-up	1	> 615.38	> 259.74	> 51 428.57
Traffic Light (1.8×10^9 op calls)	Runtime	> 3600	3.06	17.81	0.08
	Speed-up	1	> 1176.47	> 202.13	> 45 000
Sieve (1 op call, primes until 2 Million)	Runtime	49.83	2.86	21.94	4.65
	Speed-up	1	17.42	2.27	10.72
Scheduler (9.6×10^6 op calls)	Runtime	158.27	2.17	2.78	1.98
	Speed-up	1	72.94	56.93	79.93
Cruise Controller (Volvo, 136.1×10^6 op calls)	Runtime	> 3600	6.68	10.66	0.21
	Speed-up	1	> 538.92	> 337.71	> 17 142.86
CAN Bus (J. Colley, 15×10^6 op calls)	Runtime	199.66	1.61	1.65	0.61
	Speed-up	1	124.01	121.01	327.31
Train (ten routes) [1, 51] (940×10^3 op calls)	Runtime	45.16	2.41	3.54	1.66
	Speed-up	1	18.74	12.76	27.20
sort_m2_data1000 [59] (500.5×10^3 op calls)	Runtime	7.67	0.44	0.13	0.10
	Speed-up	1	17.43	59	76.7

JavaScript.[10] As explained in [65] and [67], those selected models range from small to large ones, covering various performance aspects. Due to the small number of states, we replaced the simulation benchmarks Lift, Traffic Light, and Sieve with the following machines for model checking: a Counter to one million, Landing Gear, NoTa, and N-Queens (with $N = 4$). As explained in [67], some models were rewritten to make B2Program applicable; for ProB, we benchmarked the original versions. Landing Gear was originally modeled by Ladenberger et al. [44], and then translated to classical B to make B2Program applicable (see Sect. 6). N-Queens also has few states, but computing transitions without constraint solving is very time-consuming. Compared to the earlier performance analyses, further optimizations were made in the generated Java, JavaScript, and C++ code. The complete benchmark set can be found in the B2Program repository.[11] Each benchmark is run five times on a MacBook Pro (16 GB RAM, Apple M1 Pro Chip with eight cores[12]) with a timeout of one hour and then the median runtime is taken.

Table 1 shows the simulation benchmarks comparing ProB, Java, JavaScript, and C++. Following the approach in [65], we execute operations in a long-running `while` loop. For ProB, we used the `-execute` command to just execute the first enabled transitions and avoid exploring the state space. Nevertheless, ProB always performs variant checking for `while` loops, which cannot be turned off.

Here, one can see that the generated JavaScript code outperforms ProB. For most benchmarks, JavaScript is one or two orders of magnitude faster than ProB. Sieve is a model with many set operations where JavaScript is less than one magnitude faster than ProB.[13] The slower runtime of ProB could be explained by the use of an interpreter implemented in Prolog. In contrast, the B syntax is compiled to JavaScript with B2Program. In addition, for the ARM processor used in the experiments, SICStus Prolog lacks the

[10] NodeJS 19.9.0.

[11] https://github.com/favu100/b2program.

[12] Six performance cores, two efficiency cores.

[13] This Sieve model is a slightly different, more low-level version compared of the one in Listing 1.

Table 2 Model checking runtimes (ProB, generated Java, generated JavaScript, and generated C++ code) in seconds with size of state space (states and transitions), speed-up relative to ProB, OP = Operation Reuse; Models from [67] were re-benchmarked for ProB, Java, and C++ with a different device

Counter		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(1 000 001 states,	Runtime	65.23	0.67	0.80	1.21	1.78	0.36	0.63
2 000 001 transitions)	Speed-up	1	97.36	81.54	53.91	36.65	181.19	103.54
Cruise Controller (Volvo,		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
1360 states,	Runtime	0.37	0.46	0.46	0.13	0.16	0.06	0.07
26 149 transitions)	Speed-up	1	0.80	0.80	2.85	2.31	6.17	5.29
CAN BUS (J.Colley,		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
132 599 states	Runtime	14.98	1.31	1.49	2.34	3.58	1.14	1.02
340 266 transitions)	Speed-up	1	11.44	10.05	6.4	4.18	13.14	14.69
Landing Gear [44]		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(131 328 states,	Runtime	24.58	4.23	4.86	8.87	11.26	6.07	5.82
884 369 transitions)	Speed-up	1	5.81	5.06	2.77	2.18	4.05	4.22
NoTa [57]		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(80 718 states,	Runtime	16.26	4.17	3.70	9.41	10.66	11.18	10.31
1 797 353 transitions)	Speed-up	1	3.90	4.39	1.73	1.53	1.45	1.58
Train [1, 51] (ten routes,		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
672 174 states,	Runtime	408.51	240.55	207.57	830.49	828.12	253.00	186.54
2 244 486 transitions)	Speed-up	1	1.70	1.97	0.49	0.49	1.61	2.19
sort_1000 [59]		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(500 501 states,	Runtime	183.23	373.73	37.76	> 3600	106.30	820.43	112.88
500 502 transitions)	Speed-up	1	0.49	4.85	< 0.05	1.72	0.22	1.62
N-Queens with N = 4		ProB OP	Java	Java + Cache	JavaScript	JavaScript + Cache	C++	C++ + Cache
(4 states,	Runtime	0.04	74.67	71.25	19.55	20.03	15.69	15.75
6 transitions)	Speed-up	1	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01

JIT compiler.[14] In our previous work [65], the JIT compiler was available and the performance gap between ProB and the generated Java code is less pronounced. Finally, ProB supports unlimited precision integers, while we used primitive integers for Java, JavaScript, and C++ here. Although JavaScript is an interpreted language, our new backend for B2Program performs very well: JavaScript and Java runtimes are usually within an order of magnitude and it seems that the JIT compiler in NodeJS optimizes effectively. As already discussed in prior work, C++ leads to a speedup compared to Java regarding simulation for all benchmarks except Sieve [65]. Compared to JavaScript, C++ is even faster for all simulation benchmarks we have considered (see Table 1). Especially in the simulation case, where we specialize the operations' input specifically, *clang*'s `-O2` optimization can optimize strongly [65].

Note, however, that Table 1 contains benchmarks for simulation or trace-replay, not for animation, i.e., we measure the performance of executing the model on long-running paths where operations parameters are provided explicitly. In animation — as in model checking — the tools need to compute all enabled transitions and present them to the user. To analyze the performance for this, we analyze the model checking performance of the generated JavaScript code. In prior work, model checking code generation was implemented in B2Program for Java and C++ [67]. In this work, we have extended B2Program's model checking code generation for JavaScript. For Java, JavaScript, and C++, we benchmarked both with and without caching. When activating B2Program's caching, the operations' guards and the invariant conjuncts are only computed if they contain variables that are changed by the operation that is executed to reach this state [67]. Although caching is not yet implemented in the animator of the dynamic export, this feature could be implemented later easily. For ProB we activated the operation reuse feature together with state compression [47] (`-p OPERATION_REUSE full` and `-p COMPRESSION TRUE`) which is an efficient caching strategy for animation

[14] https://sicstus.sics.se/download4.html.

and model checking. Invariant checking is also activated as it is displayed to the user of the dynamic export.

Note that we do not benchmark code generation and compilation time of B2Program. In the use case of this work, interactive validation documents are usually generated once, and can then be used by a domain expert. However, for the verification use case with model checking, B models might be compiled more frequently, e.g., when the encoding of the B model changes. Compilation for C++ might be significantly more time-consuming than model checking [67]. More details regarding code generation and compilation runtime with B2Program are discussed in [67].

The model checking results are shown in Table 2. Here, one can see that for most JavaScript benchmarks, we achieved runtimes within an order of magnitude as Java and C++. An exception here is Sort without caching, where JavaScript is around one order of magnitude or more slower than C++, and at least one order of magnitude slower than Java. For a few models, JavaScript is faster than Java or C++, for others it is the other way around (see Table 2). As already analyzed in [67], ProB's operation reuse can improve the performance up to the same order of magnitude as Java and C++, and thus also JavaScript for some models. For example, the generated JavaScript code has better performance than ProB for CAN BUS, Landing Gear, or NoTa, but not for Train, Sort (without caching), or N-Queens. The poor performance of JavaScript for Sort without caching is due to the invariant checking. The N-Queens example shows that ProB's constraint-solving capability can make it much faster than B2Program (also discussed in [67]). A similar effect appeared in the automotive case study in Sect. 6, where ProB can be up to three orders of magnitude faster at computing all enabled transitions presented to the user. For Light System, we only measured the computation of all enabled transitions for different states; due to the limitations presented before, the model is not yet feasible for model checking with B2Program. We plan to tackle this problem in the near future. Still, for all the case studies, the performance of B2Program was sufficient for interactive exploration; there were also no problems with memory usage, as they are about the same order of magnitude as Java. Note that SICStus Prolog lacks the JIT compiler for the ARM processor used here; in our previous model checking benchmarks [67] the JIT compiler was available and the results of ProB are slightly better (e.g., ProB is actually faster for Train than Java, Java + cache and C++, but slower than C++ + cache) but not fundamentally different.

8 Related work

In the following, we compare this work with existing tools that integrate domain experts in the software development process.

Requirements Automatic translation of natural language requirements makes it possible to involve domain experts more directly in the validation process. An example is the requirements language FRETish [30], which is supported by the tool FRET [29]. Using FRET, the domain expert can write FRETish requirements in natural language that are translated to linear temporal logic (LTL). To further improve communication between modeler and domain expert, FRET supports the visualization and simulation of the underlying LTL formulas. A similar approach is followed by the tool SPEAR [20]. In contrast, our work does not yet enable the domain expert to directly validate formal properties. Instead, the domain expert can run scenarios for certain properties, and inspect the behavior in a domain-specific visualization.

Other works support writing high-level domain-specific scenarios for execution on a formal model, e.g., Gherkin and Cucumber for Event-B to run scenarios using the ProB animator [21, 61]. This allows a domain expert to write scenarios in natural language, execute them, and check the behavior afterwards. As the base of communication, modelers and domain experts must agree on the events' meaning in natural language. Furthermore, the AValLa language was introduced to write domain-specific scenarios in ASMs, and run them using AsmetaV [12]. Another ASM tool is ASM2C++ which translates ASMs to C++, and AValLa scenarios to BDD code targeting the generated C++ code [8]. In our approach, the domain expert first creates scenarios by interacting with the domain-specific VisB visualization. More recently, the dynamic export also allows domain experts to write description text for each operation that is executed to describe the effect. Thus, our base of communication is the VisB visualization, and the import/export of scenarios with feedback (in the form of description text).

Documentation ProB Jupyter [27] provides a notebook interface for formal models (in B, Event-B, TLA+, etc.). It also supports generating HTML, LaTeX, and PDF documents from Jupyter notebooks. This way, it is also possible to generate validation documentation with explanatory texts. More recently, ProB Jupyter supports VisB domain-specific visualizations as used in this article.

The LaTeX mode [46] of ProB can be used to produce LaTeX documents and generate documentation with explanatory texts, visualizations and tables. It does not support VisB and domain-specific visualizations have to be created via LaTeX.

Visualizations This work has already outlined how important (domain-specific) visualizations are to validate a formal model.

There are more visualization tools for the B method like BMotionWeb [41], BMotionStudio [43], AnimB,[15] Brama

[15] http://wiki.event-b.org/index.php/AnimB.

[60], JeB [53], and the animation function [50] in ProB. A detailed comparison between these tools and VisB (together with SimB's interactive simulation) is described in [71] and [64]. An important novelty of our approach is that we create stand-alone artefacts for domain experts. However, B2Program used for the dynamic export only supports a subset of the B language.

State space projection was introduced by Ladenberger and Leuschel [42] to enable validation to focus on a subcomponent or particular aspect of a system. In future, we would like to incorporate projection diagrams into our approach.

PVSio-Web [70] is a tool for visualizing PVS models and creating prototypes, especially human-machine interfaces. This enables the user to assemble an interactive visualization for the model. In our approach, VisB visualizations are created manually, i.e., by creating an SVG image in an editor such as Inkspace, and by writing the VisB glue file. Similar to using VisB together with SimB's interactive simulation, PVSio-Web also supports simulation underneath.

There are also tools to create prototypes for VDM-SL models [55, 56]. Similar to our work, those works also allow domain-specific visualization, animation, simulation, and recording scenarios. In addition to validation by users and domain experts, the VDM-SL tools also incorporate UI designers as stakeholders.

Simulators JeB [53] supports animation, simulation and visualization by generating HTML with JavaScript from an Event-B model. The user can encode functions by hand to enable the execution of complex models. To ensure the reliability of the simulated traces, JeB's approach introduced the notion of fidelity.

In our approach, it is also possible to write additional code by hand. Compared to JeB, B2Program supports easy import and export of traces. While JeB translates Event-B models to JavaScript constructs, which are then run by an interpreter, B2Program translates B models nearly one-to-one to TypeScript classes.

This work also generates an interpreter for SimB's timed probabilistic simulation with user interaction. As discussed by Vu et al. [66], SimB is also related to simulation tools such as JeB, Uppaal [6], the co-simulation tool INTO-CPS [62], the ASM simulation tool AsmetaS [24, 25] in the Asmeta toolset [26], and the VDM simulation tool Overture [45]. With the implementation of SimB in this work, it is now possible to use SimB together with VisB as in ProB2-UI. Simulation scenarios can thus be exported by both modelers and domain experts, and shared between them.

OPEN/CÆSAR [23] is a language-independent, open software architecture for concurrent systems that allows verification, simulation, and testing. One of its features is an interactive simulator which works similarly to the animation feature in the operations view of our work. Enabled transitions are computed and shown to the user from which the user can choose one for execution. Additionally, we allow interaction with more realistic prototypes by combining domain-specific VisB visualization and timed probabilistic SimB simulation. In our work, the user can execute operations via the VisB visualization, which can also trigger a simulation in real-time. To extend OPEN/CÆSAR by another language, one must implement a C compiler for this language against OPEN/CÆSAR's interface. To extend B2Program, one would have to translate this language to B models compatible with B2Program. This principle is also applied in ProB, where some languages like TLA+ [31] or Alloy [40] are translated to B. (It is, however, also possible to provide the operational semantics as Prolog rules, as is done for CSP [11]).

Code generators Related code generators to B2Program are code generators for B [9, 15, 63], Event-B [14, 18, 22, 54, 59], ASM [7] and VDM [37]. Detailed comparisons have already been made in previous work by Vu et al. [65, 67]

Model checkers Our implementation and the associated performance analysis in this work resulted in an additional model checking tool as a by-product, generating JavaScript model checking code for a B model. Thus, model checkers such as ProB [48], TLC [32, 73], SPIN [34], pyB [72], LTSmin [38] are also related work. A detailed comparison of these model checkers with B2Program's model checking code generation is discussed by Vu et al. [67] In this work, we have achieved a satisfactory level of performance with JavaScript model checking code, reaching runtimes within an order of magnitude as Java for most benchmarks (see Sect. 7).

From the domain expert's view, only the animator is available via the HTML document, but not the model checker. We have made this decision as model checking is a technique a domain expert is usually not familiar with.

9 Conclusion and future work

In this work, we presented two solutions to improve the communication between modelers and domain experts by creating "interactive validation documents": (1) a (mostly) static export of a trace to an HTML file, and (2) a fully dynamic export of a classical B machine to an HTML document. While the static export works for all formalisms in ProB, the dynamic export only works for classical B machines supported by B2Program. The static export is suitable for analyzing a scenario or trace and allows the user to step through the saved trace and inspect the various states of the trace. In contrast, the dynamic export is suitable when domain experts have to animate or simulate traces, e.g., to modify existing traces, or to validate entire requirements.

Both approaches use domain-specific visualizations to help a domain expert reason about the formal model. For the dynamic export, we extended B2Program to generate HTML and JavaScript code while incorporating VisB visualizations. This makes it possible to interact with the model and check its behavior without the knowledge of the modeling language and its tools. Communication between modelers and domain experts is eased by features for importing/exporting scenarios and writing description texts. As new features for supporting the validation process, it is now possible to run (interactive) SimB simulations. A user or domain expert can now simulate scenarios with timing and probabilistic properties or evaluate the system's reaction to a user interaction. Overall, this work enables involving domain experts in the development and validation process more actively. These aspects have been demonstrated by two case studies: a light system model from the automotive domain, and a landing gear case study from the aviation domain.

Furthermore, we discussed the limitations of the dynamic export and analyzed the performance of the generated JavaScript code from B2Program. For most benchmarks in JavaScript, we achieved runtimes within an order of magnitude as Java and C++; a few models are faster in JavaScript, and for others, it is the other way around. We also encountered a benchmark where JavaScript is around one or more orders of magnitude slower than C++ and Java. Compared to ProB, the performance of simulation and trace replay seems to be significantly better. For animation and model checking, some models can be processed with JavaScript faster, while for others ProB achieves faster runtimes. With the operation caching feature in ProB, a strong performance boost could be achieved [47]. Furthermore, ProB is particularly efficient in models where constraint solving can be used well. Overall, the performance of all our case studies was good enough to interact with the model in dynamic export.

B2Program is available at:

https://github.com/favu100/b2program

Case studies are available at:

https://github.com/favu100/b2program/tree/master/visualizations[16]

In the future, one could support state diagrams, which are an important technique for domain-specific validation. To support a larger subset of SimB simulation, one could think about generating code for SimB instead of generating a SimB interpreter. Another possible future work is generating other application formats, such as standalone JavaFX applications.

[16] Also accessible via https://favu100.github.io/b2program/.

Acknowledgements We would like to thank anonymous reviewers for their constructive feedback.

Author Contribution Fabian Vu: Conceptualization, writing, original draft, review and editing, implementation (dynamic export), maintenance, material preparation, data collection, analysis; Christopher Happe: Conceptualization, implementation (dynamic export), data collection; Michael Leuschel: Implementation (static export), writing, review and editing, supervision, funding acquisition, project administration, material preparation, analysis.

Funding Open Access funding enabled and organized by Projekt DEAL. This research is part of the IVOIRE project funded by the "Deutsche Forschungsgemeinschaft" (DFG) and the Austrian Science Fund (FWF); grant # I 4744-N. Section 4 is part of the KI-LOK project funded by the "Bundesministerium für Wirtschaft und Energie"; grant # 19/21007E.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Hoare, A.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
3. Bagwell, P.: Ideal hash trees. Es Grands Champs **1195** (2001)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
5. Bendisposto, J., Geleßus, D., Jansing, Y., Leuschel, M., Pütz, A., Vu, F., Werth, M.: ProB2-UI: a Java-based user interface for ProB. In: Proceedings FMICS. LNCS, vol. 12863, pp. 193–201 (2021)
6. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL — a tool suite for automatic verification of real-time systems. In: Hybrid Systems III. LNCS, vol. 1066, pp. 232–243 (1996)
7. Bonfanti, S., Carissoni, M., Gargantini, A., Mashkoor, A.: Asm2C++: a tool for code generation from abstract state machines to Arduino. In: Proceedings NFM. LNCS, vol. 10227, pp. 295–301 (2017)
8. Bonfanti, S., Gargantini, A., Mashkoor, A.: Design and validation of a C++ code generator from Abstract State Machines specifications. J. Softw. Evol. Process **32** (2020)
9. Bonichon, R., Déharbe, D., Lecomte, T., Medeiros, V. Jr: LLVM-based code generation for B. In: Proceedings SBMF. LNCS, vol. 8941, pp. 1–16 (2014)
10. Boniol, F., Wiels, V.: The landing gear system case study. In: ABZ 2014: The Landing Gear Case Study. CCIS, vol. 433, pp. 1–18 (2014)

11. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Proceedings of Formal Methods 2005. LNCS, vol. 3582, pp. 221–236. Springer, Newcastle upon Tyne (2005)
12. Carioni, A., Gargantini, A., Riccobene, E., Scandurra, P.: A scenario-based validation language for ASMs. In: Proceedings ABZ. LNCS, vol. 5238, pp. 71–84 (2008)
13. Carlsson, M., Widen, J., Andersson, J., Andersson, S., Boortz, K., Nilsson, H., Sjöland, T.: SICStus Prolog user's manual. Swedish Institute of Computer Science Kista, (1988)
14. Cataño, N., Rivera, V.: EventB2Java: a code generator for Event-B. In: Proceedings NFM. LNCS, vol. 9690, pp. 166–171 (2016)
15. ClearSy: User and reference manuals. Aix-en-Provence, France (2016). http://www.atelierb.eu/
16. Dollé, D., Essamé, D., Falampin, J.: B dans le transport ferroviaire. L'expérience de Siemens Transportation Systems. Tech. Sci. Inform. **22**(1), 11–32 (2003)
17. Döring, L.: Feasibility and uses of a superset of B0 for embedded code-generation. Master's thesis, Heinrich-Heine-Universität Düsseldorf (2023)
18. Edmunds, A.: Templates for Event-B code generation. In: Proceedings ABZ. LNCS, vol. 8477, pp. 284–289 (2014)
19. Essamé, D., Dollé, D.: B in large-scale projects: the canarsie line CBTC experience. In: B 2007: Formal Specification and Development in B. LNCS, vol. 4355, pp. 252–254 (2006)
20. Fifarek, A.W., Wagner, L.G., Hoffman, J.A., Rodes, B.D., Aiello, M.A., Davis, J.A.: SpeAR v2.0: formalized past LTL specification and analysis of requirements. In: Proceedings NFM. LNCS, vol. 10227, pp. 420–426 (2017)
21. Fischer, T., Dghyam, D.: Formal model validation through acceptance tests. In: Proceedings RSSRail. LNCS, vol. 11495, pp. 159–169 (2019)
22. Fürst, A., Hoang, T.S., Basin, D.A., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Proceedings iFM. LNCS, vol. 8739, pp. 323–338 (2014)
23. Garavel, H.O.: An open software architecture for verification, simulation, and testing. In: Proceedings TACAS. LNCS, vol. 1384, pp. 68–84 (1998)
24. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based simulator for ASMs. In: Proceedings ASM Workshop (2007)
25. Gargantini, A., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univers. Comput. Sci. **14**(12), 1949–1983 (2008). https://doi.org/10.3217/jucs-014-12-1949
26. Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven language engineering: the ASMETA case study. In: Proceedings ICSEA, pp. 373–378 (2008). https://doi.org/10.1109/ICSEA.2008.62
27. Geleßus, D., Leuschel, M.: ProB and Jupyter for logic, set theory, theoretical computer science and formal methods. In: Proceedings ABZ. LNCS, vol. 12071, pp. 248–254 (2020)
28. Geleßus, D., Stock, S., Vu, F., Leuschel, M., Mashkoor, A.: Modeling and analysis of a safety-critical interactive system through validation obligations. In: Proceedings ABZ. LNCS, vol. 14010, pp. 284–302 (2023)
29. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Generation of formal requirements from structured natural language. In: Proceedings REFSQ. LNCS, vol. 12045, pp. 19–35 (2020)
30. Giannakopoulou, D., Pressburger, T., Mavridou, A., Schumann, J.: Automated formalization of structured natural language requirements. Inf. Softw. Technol. **137**, 106590 (2021). https://doi.org/10.1016/j.infsof.2021.106590
31. Hansen, D., Leuschel, M.: Translating TLA+ to B for validation with ProB. In: Proceedings iFM. LNCS, vol. 7321, pp. 24–38 (2012)
32. Hansen, D., Leuschel, M.: Translating B to TLA+ for validation with TLC. In: Proceedings ABZ. LNCS, vol. 8477, pp. 40–55 (2014)
33. Hansen, D., Leuschel, M., Körner, P., Krings, S., Naulin, T., Nayeri, N., Schneider, D., Skowron, F.: Validation and real-life demonstration of ETCS hybrid level 3 principles using a formal B model. Int. J. Softw. Tools Technol. Transf. **22**(3), 315–332 (2020). https://doi.org/10.1007/s10009-020-00551-6
34. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley, Reading (2011)
35. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system. In: Proceedings ABZ. LNCS, vol. 12071, pp. 281–301 (2020)
36. Institute of Electrical and Electronics Engineers: IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. IEEE, New York (1991). https://doi.org/10.1109/IEEESTD.1991.106963
37. Jørgensen, P.W.V., Larsen, M., Couto, L.D.: A code generation platform for VDM. In: Proceedings of the 12th Overture Workshop. School of Computing Science, Newcastle University, Technical Report CS-TR-1446, UK (2015)
38. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Proceedings TACAS. LNCS, vol. 9035, pp. 692–707 (2015)
39. Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., Leuschel, M.: Integrating formal specifications into applications: the ProB Java API. Form. Methods Syst. Des. **58**(1), 160–187 (2021). https://doi.org/10.1007/s10703-020-00351-3
40. Krings, S., Schmidt, J., Brings, C., Frappier, M., Leuschel, M.: A translation from Alloy to B. In: Proceedings ABZ. LNCS, vol. 10817, pp. 71–86 (2018)
41. Ladenberger, L.: Rapid creation of interactive formal prototypes for validating safety-critical systems. Ph.D. thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2016)
42. Ladenberger, L., Leuschel, M.: Mastering the visualization of larger state spaces with projection diagrams. In: Proceedings ICFEM. LNCS, vol. 9407, pp. 153–169 (2015). https://doi.org/10.1007/978-3-319-25423-4_10
43. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-motion studio. In: Proceedings FMICS. LNCS, vol. 5825, pp. 202–204 (2009)
44. Ladenberger, L., Hansen, D., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using ProB. Int. J. Softw. Tools Technol. Transf. **19**(2), 187–203 (2017). https://doi.org/10.1007/s10009-015-0395-9
45. Larsen, P., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The overture initiative: integrating tools for VDM. SIGSOFT Softw. Eng. Notes **35**, 1–6 (2010). https://doi.org/10.1145/1668862.1668864
46. Leuschel, M.: Formal model-based constraint solving and document generation. In: Proceedings SBMF. LNCS, vol. 10090, pp. 3–20 (2016). https://doi.org/10.1007/978-3-319-49815-7_1
47. Leuschel, M.: Operation caching and state compression for model checking of high-level models — how to have your cake and eat it. In: Proceedings iFM. LNCS, vol. 13274, pp. 129–145 (2022)
48. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Proceedings FME. LNCS, vol. 2805, pp. 855–874 (2003)
49. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008). https://doi.org/10.1007/s10009-007-0063-9
50. Leuschel, M., Samia, M., Bendisposto, J.: Easy graphical animation and formula visualisation for teaching B. In: The B Method: From Research to Teaching, pp. 17–32 (2008)

51. Leuschel, M., Bendisposto, J., Hansen, D.: Unlocking the mysteries of a formal model of an interlocking system. In: Proceedings Rodin Workshop (2014)
52. Leuschel, M., Mutz, M., Werth, M.: Modelling and validating an automotive system in classical B and Event-B. In: Proceedings ABZ. LNCS, vol. 12071, pp. 335–350 (2020)
53. Mashkoor, A., Yang, F., Jacquot, J.: Refinement-based validation of Event-B specifications. Softw. Syst. Model. **16**(3), 789–808 (2017). https://doi.org/10.1007/s10270-016-0514-4
54. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Proceedings SoICT, pp. 179–188. ACM ICPS (2011)
55. Oda, T., Yamamoto, Y., Nakakoji, K., Araki, K., Larsen, P.: VDM animation for a wider range of stakeholders. In: Proceedings Overture Workshop, pp. 18–32 (2015)
56. Oda, T., Akari, K., Yamamoto, Y., Nakakoji, K., Chang, H.M., Larsen, P.: Specifying abstract user interface in VDM-SL. In: Proceedings International Overture Workshop, pp. 5–20 (2021)
57. Oliver, I.: Experiences in using B and UML in industrial development. In: Proceedings B ’07. LNCS, vol. 4355, pp. 248–251 (2007). https://doi.org/10.1007/11955757_20
58. Parr, T.: StringTemplate website (2013). http://www.stringtemplate.org/. Accessed: 2022-07-05
59. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for Event-B. Int. J. Softw. Tools Technol. Transf. **19**(1), 31–52 (2017)
60. Servat, T.: BRAMA: a new graphic animation tool for B models. In: Proceedings B ’07. LNCS, vol. 4355, pp. 274–276 (2006)
61. Snook, C., Hoang, T.S., Dghaym, D., Fathabadi, A.S., Butler, M.: Domain-specific scenarios for refinement-based methods. J. Syst. Archit. **112**, 101833 (2021)
62. Thule, C., Lausdahl, K., Gomes, C., Meisl, G., Larsen, P.G.: Maestro: the INTO-CPS co-simulation framework. Simul. Model. Pract. Theory **92**, 45–61 (2019). https://doi.org/10.1016/j.simpat.2018.12.005
63. Voisinet, J.C.: JBTools: an experimental platform for the formal B method. In: Proceedings PPPJ ’02/IRE ’02, pp. 137–139 (2002)
64. Vu, F., Leuschel, M.: Validation of formal models by interactive simulation. In: Proceedings ABZ. LNCS, vol. 14010, pp. 59–69 (2023)
65. Vu, F., Hansen, D., Körner, P., Leuschel, M.: A multi-target code generator for high-level B. In: Proceedings iFM. LNCS, vol. 11918, pp. 456–473 (2019). https://doi.org/10.1007/978-3-030-34968-4_25
66. Vu, F., Leuschel, M., Mashkoor, A.: Validation of formal models by timed probabilistic simulation. In: Proceedings ABZ. LNCS, vol. 12709, pp. 81–96 (2021)
67. Vu, F., Brandt, D., Leuschel, M.: Model checking B models via high-level code generation. In: Proceedings ICFEM. LNCS, vol. 13478, pp. 334–351 (2022)
68. Vu, F., Happe, C., Leuschel, M.: Generating domain-specific interactive validation documents. In: Proceedings FMICS. LNCS, vol. 13487, pp. 32–49 (2022)
69. Warford, J.S.: The MVC Design Pattern, pp. 175–199. Vieweg+Teubner Verlag, Wiesbaden (2002). https://doi.org/10.1007/978-3-322-91603-7_9
70. Watson, N., Reeves, S., Masci, P.: Integrating user design and formal models within PVSio-web. In: Proceedings F-IDE, EPTCS, vol. 284, pp. 95–104 (2018). https://doi.org/10.4204/EPTCS.284.8
71. Werth, M., Leuschel, M.: VisB: a lightweight tool to visualize formal models with SVG graphics. In: Proceedings ABZ. LNCS, vol. 12071, pp. 260–265 (2020)
72. Witulski, J.: A Python B implementation — PyB A second tool-chain. Ph.D. thesis, Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2018)
73. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Proceedings CHARME. LNCS, vol. 1703, pp. 54–66 (1999)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.