Formal Methods in System Design (2021) 58:160–187
https://doi.org/10.1007/s10703-020-00351-3

Integrating formal specifications into applications: the ProB Java API

Philipp Körner[1] · **Jens Bendisposto**[1] · **Jannik Dunkelau**[1] · **Sebastian Krings**[1] · **Michael Leuschel**[1]

Received: 12 March 2020 / Accepted: 29 September 2020 / Published online: 21 October 2020

Abstract
The common formal methods workflow consists of formalising a model followed by applying model checking and proof techniques. Once an appropriate level of certainty is reached, code generators are used in order to gain executable code. In this paper, we propose a different approach: instead of generating code from formal models, it is also possible to embed a model checker or animator into applications in order to use the formal models themselves at runtime. We present a Java API to the ProB animator and model checker. We describe several case studies that use this API as enabling technology to interact with a formal specification at runtime.

Keywords Formal methods · Run-time execution · Embedding · ProB · ProB Java API

1 Introduction

When designing safety-critical software, the use of formal methods is highly recommended [18] to ensure correctness. This is often done by combining (manual and automatic) proof with model checking.

Once a formal model has been found to be correct, it is usually required to translate the model into a traditional, imperative programming language. Then, low-level formalisms are usually close enough that code can be generated easily. When using high-level formalisms

✉ Philipp Körner
p.koerner@hhu.de

Jens Bendisposto
jens.bendisposto@hhu.de

Jannik Dunkelau
jannik.dunkelau@hhu.de

Sebastian Krings
sebastian.krings@hhu.de

Michael Leuschel
leuschel@hhu.de

[1] Institut für Informatik, HHU Düsseldorf, Universitätsstr. 1, 40225 Düsseldorf, Germany

though, the model has to be gradually refined to an implementation level so that it only uses a restricted version of the specification language, disallowing high-level constructs which require, e.g., constraint solving techniques or unconstrained memory for execution. The alternative to code generation is manual implementation, which is known to be error-prone.

In this paper, we investigate another approach: we assume that a high-level specification is written to be *executable*, in the sense that a tool like an animator or model checker is able to compute all state transitions. Can we then implement a program interfacing with, e.g., a model checker that also simulates the environment and executes the model by choosing a traversing transition?

This paper is a mixture of a position, tool and application paper, and is structured as follows: in the remainder of this section, we briefly introduce two high-level specification languages, B and Event-B, as well as ProB, an animator and model checker for these languages. Afterwards, we present the enabling technology, the ProB Java API, which allows for fine-grained interaction with ProB in Sect. 2. Following, we evaluate our approach by implementing and discussing several new case studies based on the ProB Java API in Sect. 3, summarising its use in existing industrial applications and insights gained from implementation work. In Sect. 4, we distinguish an embedding of a formal specification in software from user-driven animation and revisit arguments concerning executability of specifications in the context of our approach and the B language. Then, related work in form of similar tools and other applications built by third parties on top of the ProB Java API are considered in Sect. 5. Next, we give an outlook about what kind of applications of the presented approach we may see in the future in Sect. 6, before drawing our conclusions in Sect. 7.

This article is based on our contribution to the 3rd world congress on formal methods (FM'19) [51]. It extends the original paper in the following ways:

- We give a more in-depth overview of the capabilities of the ProB Java API, and present some code snippets that show basic usage of its API in a Java application.
- We discuss two additional, previously unpublished case studies that show further use cases of the technology.
- The discussion in Sect. 5 is set into a more B-specific context, since the logical foundation is neither intended nor possible to execute entirely.
- We include a more thorough comparison with new experiments regarding code generation as presented in [89].
- We give a summary of known third-party tools that already use ProB Java API.
- We discuss the integration potential of executable specifications with AI as well as possible future use cases.

1.1 B, Event-B and ProB

Both B [4] and its successor Event-B [3] are state-based specification languages that allow for high levels of abstraction. They are based on Zermelo–Fraenkel set theory with the axiom of choice [28,29], using sets for data modelling. Further, they make use of generalised substitution for state modifications, and refinement calculus [5,6] to describe models at different levels of abstraction [13].

The highest level of abstraction includes, besides set theory, formulation of quantified formulae over arbitrary domains, functional composition and lambda expressions, as well as non-deterministic assignments.[1]

[1] Cf. https://www3.hhu.de/stups/prob/index.php/Summary_of_B_Syntax.

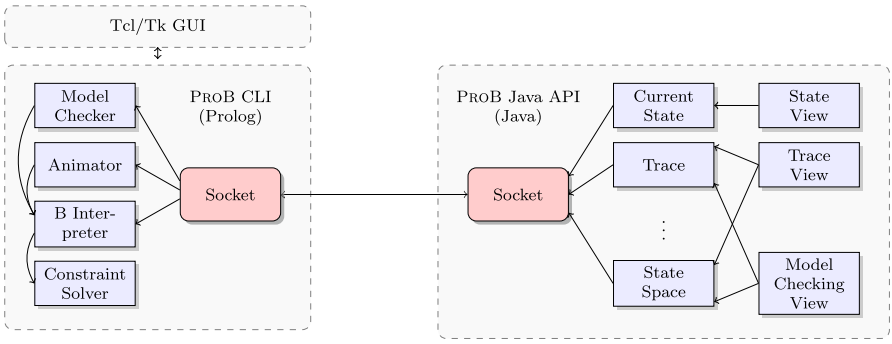

Fig. 1 Overview of the ProB ecosystem

In the following, we describe several projects that make use of ProB [61], an animator and model checker for both B and Event-B. Its core is developed mainly in SICStus Prolog [16], with some parts being implemented in C and Java, and makes use of co-routines and SICStus' CLP(FD) library [15]. Besides B, ProB offers support for several other formalisms as well, including TLA$^+$ [55] (via translation to B [37]), Z [68,83], CSP [11,44] and more. Hence, the approach discussed in this article is immediately applicable to languages other than B and Event-B.

2 ProB Java API

As ProB is written in Prolog, which admittedly is neither the most popular nor the easiest language to pick up, it is hard for formal method experts to adapt or extend the default validation capabilities of ProB.

Thus, a main design goal of the ProB Java API was to offer convenient access to the core features of ProB. It allows data exchange in both directions, allowing one to provide inputs to ProB and obtaining the outputs without having to parse streams or log files.

The ProB Java API also allows a fine-grained interaction with ProB, not just one-shot scenarios. Indeed, ProB provides a command-line interface, which can and has been used to develop several tools on top of it (e.g., DTVT [59], OLAF, CAVAL or SafeCap [46] for data validation or BTestBox [22] for test-case generation). However, these tools usually require just a one-shot interaction with ProB: ProB is asked to validate a formal model and ProB's outputs are translated to feedback provided to the user. Many tools require a more fine-grained interaction, with repeated calls to ProB depending on the results of earlier calls.

The source code of ProB Java API is available on GitHub [71]. For developers who want to build tools on top of the ProB Java API, we create releases of the tool as jar files than can be consumed using one of the build tools for the JVM such as Maven or Gradle. The artifacts are stored on Maven Central [73].

A general overview of the ProB Java API is given in Fig. 1. For each B model that is interacted with, an instance of the ProB CLI (command line interface) which actually loads the model is started in socket-mode. This means that the ProB CLI listens on a socket for commands to execute whitelisted Prolog code. The whitelist offers fine-grained access to ProB's constraint solving, animation and model checking capabilities as well as ProB's preferences and machine components.

Each command on the whitelist has a corresponding implementation in the ProB Java API. This offers an API that is fairly low-level and intended for ProB and ProB Java API developers. It is complemented by a high-level API that is built on top and abstracts away from ProB's internals in Prolog. The high-level API allows easy animation of the model, exploration of the state space, solving custom constraints over the variables in the state space, or registration of listeners subscribed to custom formulae which are notified once a new state is reached.

The State Space acts as the central interface to the ProB CLI. It is a representation of the underlying labelled transition system. Exploring the state space by executing operations adds transitions and newly encountered states. It allows animation of the model, evaluation of predicates in arbitrary states, extraction of states that match a given predicate, and, in general, execution of arbitrary ProB Java API commands.

The Model is an in-memory version of the loaded B machine. The ProB Java API offers convenient access to the contents of the specification. This includes invariants, variables, operations and their preconditions, etc. Upon that, it is possible to expand on loaded machines by adding further invariants or operations, resulting in a dynamically altered version with stricter semantics [19].

The Trace keeps track of the path throughout the state space starting from the initialisation of the machine. Traces behave like a browser history in the sense that they are append-only, but it is possible to "go back in time" and start a new fork. Executing an operation during animation automatically appends the successor state to the currently active trace.

The State objects are linked to their corresponding state space. They store outgoing transitions as well as map abstractions of variables and formulas to abstractions of values. For example, it is possible to retrieve the value of a given state variable but also to add expressions and predicates which are automatically evaluated in every state and are kept track of.

Value Translation is required to give a meaningful representation to the values of state variables. By default, ProB provides a string representation of each value to the ProB Java API. However, they can be translated into Java data structures as well: For example, B integers are translated into BigIntegers, B sets correspond to Java sets and sequences to Java lists. Naturally, this translation does not work for infinite sets. To avoid duplication of the entire state space in ProB and the ProB Java API, only up to 100 states are cached in Java. If a non-cached state is required, it is retrieved via a handle (a unique state ID) from the ProB CLI.

Trace Synchronisation is a tool that is provided by the ProB Java API. It allows coupling of multiple traces, even on different B models. One example is that a refined machine is synchronised with a more abstract version upon the shared operations, in order to ensure that it is a valid refinement. Another example is synchronisation of two entirely different machines that are two components in a system.

In the following, we want to demonstrate how the ProB Java API can be used within a Java application. The program loads a B model, performs a number of random steps and prints the value of each variable in the final state. The example is simplified to a minimum, i.e., we do not handle errors like missing files or syntax errors in the B model. Also, deadlocks will be ignored, i.e., if no operations can be executed, the animator will remain in the same state. Listing 1 shows the animation code, the full code is available on GitHub [72].

The entry point to the ProB Java API is the Api class. We use the Guice dependency injection framework [34], this means we can retrieve a fully configured Api class using a so-

```
public static void main(String[] args) throws Exception {
    String filename = args[0];
    int steps = Integer.parseInt(args[1]);

    Api api = Main.getInjector().getInstance(Api.class);

    // load model, initialize state space
    StateSpace stateSpace = api.b_load(args[0]);
    Trace trace = new Trace(stateSpace);

    // execute specified amount of transitions
     for (int i = 0; i < steps; i++) {
       trace = trace.anyOperation(null);
     }

    State state = trace.getCurrentState();

    // print current state
    state.getVariableValues(EXPAND).forEach(
      (k, v) -> {System.out.println(k+"=>"+v);}
    );

    stateSpace.kill();
    }
```

Listing 1 ProB Java API usage example

called Injector. The Api class has methods to load formal models for several formalisms, e.g. B, Event-B, Z, CSP, TLA$^+$ and a few more. All methods return a StateSpace object, that is as described the central interface to interact with the Prolog core. We can then create a Trace object and start to execute operations, in this example we perform some random operations. We could pass a filter (e.g. a list of operation names from which we want to choose the operations), but here we pass null which means, that we do not care which operations are used. Finally, we inspect the resulting state of the execution. Here, we print the values for each variable, but we could also inspect the invariant or evaluate arbitrary expressions or predicates.

3 Examples

In this section, we describe different use cases based on several examples. The first couple of examples we discuss are student projects implementing two well-known games: Pac-Man and Chess in Sects. 3.1 and 3.2, respectively. Furthermore, we present an application in Sect. 3.3 that does not represent the typical software development cycle, but rather shows how to use the API in a more creative way: it implements a web application that checks whether predicates (written in B or TLA$^+$) are valid and provides counter-examples when not. Afterwards, in Sect. 3.4, we show an experiment regarding domain specific languages on top of B. Finally, the approach found use in two more complex projects, namely a timetable planner for university courses, and a safety critical, industrial application for the ETCS Hybrid Level 3 concept, considered in Sects. 3.5 and 3.6, respectively.

For the four software prototypes, we use the state that is translated into Java data structures in order to provide an (interactive) visualisation. The other examples provide feedback to the user via the web-based front-end or uses a read-eval-print-loop (REPL) for user interaction.

Links to all code examples that are publicly available are given in the “Code availability” section at the end of the article.

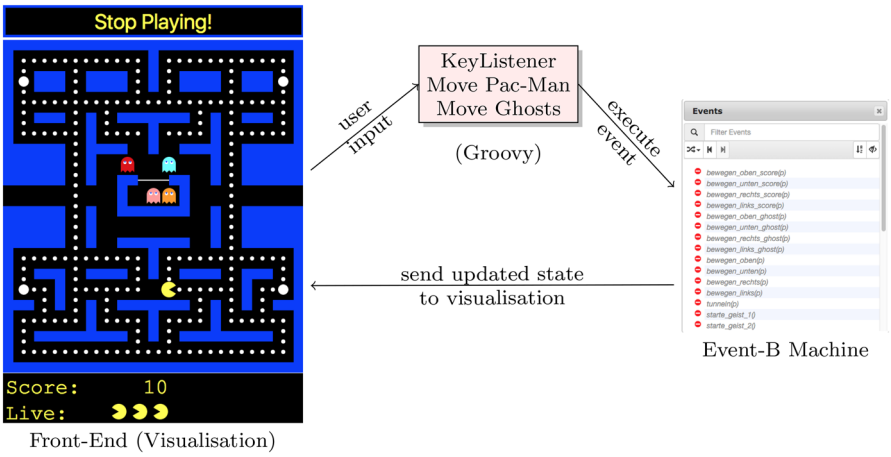

Fig. 2 Architecture of a Pac-Man game based on a formal model

3.1 Real-time animation: Pac-Man

Our first example application is based on a formal model of Pac-Man.

The formal model itself is written in Event-B. It specifies all valid positions on the board that the Pac-Man and the ghosts can be in. There are state transitions that describe valid moves, though in the model itself ghosts are allowed to turn around. The model also manages the duration and targets of super pills (so that ghosts may be eaten, but only once per pill), and encounters of the Pac-Man with pills and ghosts. Finally, it keeps tracks of the Pac-Man's lives and deadlocks the game once none of Pac-Man's lives are left. It is possible to play a turn-based version of Pac-Man in the animator.

Note that the model is non-deterministic in the sense that there are multiple available operations, one for each direction the Pac-Man and each ghost may move.

Additional to the model, we implemented an interface via ProB Java API that allows to play the game via traditional controls instead of executing transitions by clicking in the operation view. On the press of an arrow key, the following actions happen (Fig. 2):

- In the current state, it is evaluated whether the Pac-Man may move into that direction and the operation to move him is executed if allowed. Operations that result in eating a pill are preferred. This yields a new Trace object.
- For each ghost, it is evaluated whether enough time has passed to leave the monster pen. If so, the transition to move the ghost in a direction mandated by a heuristic is executed. New Trace objects are generated after moving each ghost and the movement operation is appended.
- It is verified whether the Pac-Man or some of the ghosts have to jump to the other side of the board via the tunnel. If the operation is enabled, it is executed.
- If available, operations that catch a ghost or the Pac-Man are executed.
- The GUI inspects the current state of the Trace and updates based on the new state values. The positions of the ghosts and the Pac-Man, the remaining pills, the score and the amount of remaining lives are extracted from the animation state.

For this kind of application, as the calculation of the next-state function is very fast, we did not encounter any performance issues when executing the model. We found that, even though

the visualisation is in Java, depending on the operating system and JDK implementation, the game can run smoothly or just below acceptable performance.[2] Yet, we find it especially note-worthy that it is indeed possible to create real-time applications that depend on user input based on formal models, as at least five events per tick are executed, one to move the Pac-Man and four to move the ghosts. Plain animation in ProB could not capture this, instead it would turn Pac-Man into a turn-based game.

3.1.1 Main contribution: real-time animation

The Pac-Man case study shows that our approach is feasible for real-time applications as long as the computation of successor states is not too complex. The application is able to timely react to user input, directly embedding the formal model in the application does not lead to a noticeable performance decline.

3.1.2 Lessons learned: non-determinism

The case study made obvious that it is hard to get the amount of non-determinism right. The formal model itself has to incorporate certain aspects non-deterministically, e.g. we have to take into account every key the player might press. Each time the state of the underlying model changes, ProB has to compute which events are enabled and the successor states they lead to.

However, as the player will only pick a single move out of all the possible ones, most of the events are never really executed and the computation work is discarded anyway. Due to the way ProB and the ProB Java API-based animation interact, we cannot simply let the user move first and then find out if the selected movement is actually valid as we would have to roll back changes to the state space. Simultaneously, the model has to be as deterministic as possible to allow automatic execution. As at least the ghosts are to be moved automatically, the computer controlled aspects of Pac-Man could be modelled deterministically in order to avoid ambiguity and to avoid having to implement how to decide between different options. Yet, we decided that the AI outside of the model should choose between different options, e.g., whether ghosts should turn a corner, resulting in a more general model with a higher amount of non-determinism. Finally, since a "tick" of the game, i.e., one movement of the Pac-Man and each ghost, is made up of not one but several operations that are tested and executed, this impacts performance manifold.

In summary, there is a tradeoff between determinism and execution speed that is both driven by the rules of the game as well as by design decision when modelling it. Further research is needed to find out where the sweet spot between non-determinism and determinism lies when modelling games, in particular, when we have to take into account overheads caused by the animation engine, interactivity, generality, as well as the communication between ProB and ProB Java API. Furthermore, it would be interesting to see if we can develop modelling approaches leading to an optimal tradeoff in general.

3.2 Predicting the future: chess

In the chess example, we have two use cases. Firstly, we want two (human) players to be able to play against each other. Secondly, a (simple) chess AI should be available to play against.

[2] On a Mac, it runs smoothly. On more powerful Linux PCs, it runs with stutters. We suspect that the socket communication is slower depending on the OS.

As with Pac-Man, we use the formal specification in order to specify the rules of the game. The model offers all valid moves as enabled actions, checkmate is encoded as an invariant violation. Then, we can use the vanilla ProB animator to play chess (preferably with an additional visualisation of the current state).

The more interesting part is that a basic AI of a computer-opponent is hard to specify but somewhat easy to implement. Thus, the AI was written in Java using the ProB Java API: we implemented the Minimax algorithm with alpha-beta pruning [50]. The calculated game tree has the current state at its root and its children are the successor states representing all valid turns by the AI. Their children again are their corresponding successor states where each state represents a turn by the human player and so forth. For termination, we limit the depth of the state space that should be explored, i.e. the amount of turns the AI is able to look ahead. Hence, this depth determines the AI's strength.

The Java side hereby is responsible for two things. It decides which child states need to be expanded and picks the most beneficial action for the AI opponent based on the explored game tree. Figure 3 visualises the execution. After the user's turn, the state space is explored, uncovering all possible courses the game could take. Then, the best action is chosen and the current chess state is updated accordingly. Note that the calculation of successor states happens on ProB side, as the game logic is fully implemented in B.

In the model, the board itself is represented via a square-centric approach: a total function maps each position on the board to either a chess piece or a special "empty" value. While a partial function or piece-centric approach have their individual advantages, this offers benefits for constraint solving, visualisation and easy identification of empty fields. Moving operations are split into two, one that moves a piece and one that additionally takes a piece of the enemy. Their preconditions share predicates for identifying combinations of position and chess piece, movement paths and whether a player is in check.

In order to assign a weight to each state, we use a more sophisticated evaluation function that only depends on a single state. It incorporates both the amount of pieces on the board and their positions and is also specified in B. Then, after checking states until a given depth, the turn suggested by Minimax is picked for the opponent. This strategy is very similar to bounded model checking [8], though execution is kept explicit instead of resorting to symbolic means. However, regarding this chess implementation we are not particularly concerned with violated invariants other than for identifying a checkmate state (which the AI accounts for). Instead, all possible outcomes are generated via execution of the model. Afterwards, a trace is chosen based on its Minimax value, eventually leading to an action that guarantees the most favourable outcome.

This case study offers worse results than Pac-Man from a performance perspective. Due to the state space explosion caused by the sheer amount of possible moves, generating all successor states as deep as required by a strong chess engine is infeasible. An implementation in, e.g., plain C or Java is orders of magnitudes faster. Modern chess engines usually make use of additional heuristics, and opening and end game databases in order to improve performance. Using our approach following a somewhat naive implementation, only a small part of the state space from a given board position can be generated in reasonable time, which results in the AI being a rather weak opponent.

3.2.1 Main contribution: game-driven model exploration

In this case study, we replaced the common exploration strategies of ProB (depth-first, breadth-first and random) by an exploration strategy based on the current state of a game. The Minimax algorithm is used to drive the model checker, with the aim of expanding the

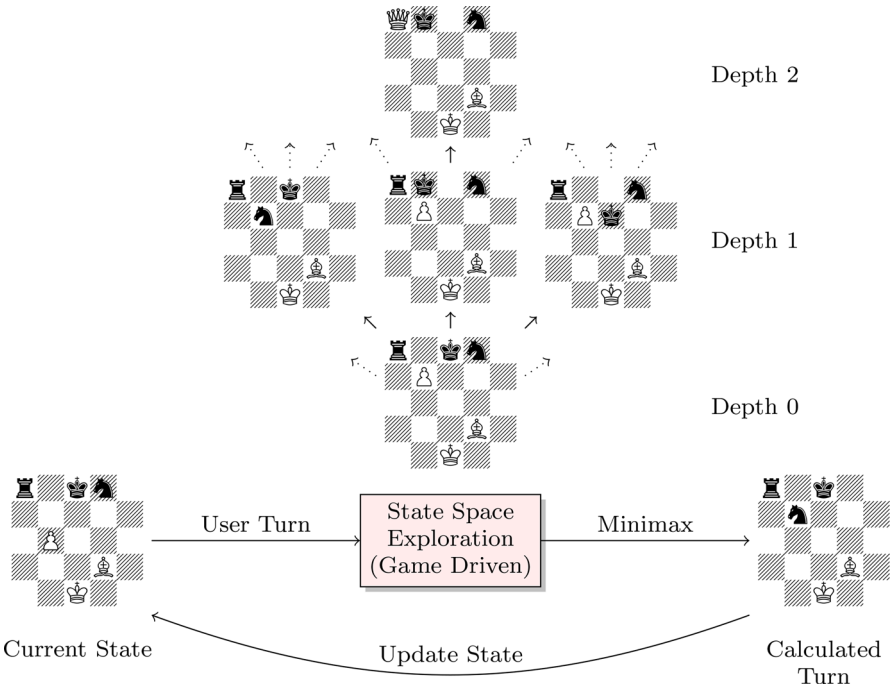

Fig. 3 Architecture of chess based on a formal model

most promising states, rather than exhaustively analysing the state space. Hence, we were able to implement a heuristic-based model checking approach.

Although ProB offers support for directed model checking [60] with a custom heuristic function already, our game-driven model exploration offers a huge advantage. Specifying an exploration heuristic in B is limited to the closed world of the calculated state. For each state, the heuristic provides a value after which it is sorted into a priority queue. It is not possible to argue about the heuristic values of, e.g., sister nodes in the search tree. By animating the model externally in ProB Java API however, we are able to do exactly that: comparing heuristic values of different nodes to decide which states do not need to be explored further by alpha-beta pruning.

In a way, this approach can be understood as a generalisation of directed model checking, as it is not restricted to searching for the common violations of interest in regular model checking scenarios (deadlocks, invariant violations, etc.). The search can, as in this example, be directed at a set of states fulfilling a certain set of criteria. While we here employed a checkmate as invariant violation, the desired criteria do not need to be formalizable in the first place (w.r.t. the state's closed world), but can also take meta information into account. Such meta information can consist of data collected over sister states, current path length, computation time needed for the state, historical data of current path, etc. Hence, it contributes highly customisable control over a highly formalisable set of operations, while not being restricted to pure model checking but allowing a wider range of analysis methods and other applications.

3.2.2 Lessons learned: model complexity

Fully encoding all possible moves on a chessboard has lead to a model that is very complex and features a very large state space. Even though our traversal strategy avoids exhaustively expanding it, debugging and partial exploration were extremely difficult:

- Errors such as incoherences with chess' movement rules sometimes only occurred for certain paths in the state space. For example, castling is only allowed if the king and the corresponding rook did not move until that point. For these cases, it is not enough to verify proper execution of operations in arbitrary states. Instead, the model has to be driven into a particular state (which includes more than just the positions on the board). To some extent, these traces had to be compiled manually.
- Once a target state was reached, it was often hard to understand why particular, complex predicates evaluated to true or false in that state. While ProB offers some debugging tools to do so, debugging B models is not as comfortable as it is for modern programming languages.
- It was hard to determine whether a bugfix covered the error in all states or just in the ones we debugged it in.
- Positions with a high number of possible moves and counter-moves take long to be evaluated by ProB, since the enabledness of all outgoing transitions is computed. In consequence, traversing them during debugging attempts slows down debugging as well.

Furthermore, the high complexity prevented our proof efforts. Further investigation into a refinement-based implementation of chess might help to overcome the difficulties.

3.3 ProB logic calculator

As an answer to a challenge proposed by Leslie Lamport [63] we implemented a logic calculator as a web application. The calculator accepts expressions and predicates in either B or TLA$^+$ and evaluates them, treating all free variables as being existentially quantified.

As an example, let us find a solution to a puzzle by Smullyan. The puzzle involves Knights and Knaves. While Knights always tell the truth, knaves always lie. We have three persons Gawain, Bors and Mordred and the following propositions:

1. Gawain says: "Bors is a knave or Mordred is a knave"
2. Bors says: "Gawain is a knight"

The translation to B is straightforward. Figure 4 shows the web-interface of the logic calculator with a solution to the puzzle.

3.3.1 Main contribution: web- and java integration

The logic calculator shows that embedding a formal methods toolchain into an application is possible with little effort. Its backend is written in about only 220 lines of Clojure [43] code on top of the constraint solving API that is provided by the ProB Java API. The application source code can be found on GitHub [84]. The logic calculator demonstrates how the ProB Java API constraint solver can be used from any language that runs on the JVM and is able to call Java. Furthermore, it shows that both desktop and web applications can be targeted.

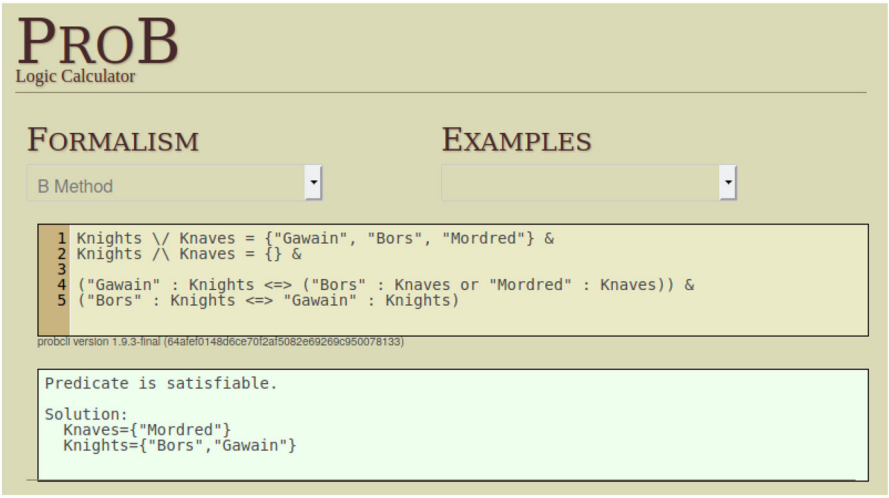

Fig. 4 ProB logic calculator http://eval-b.stups.uni-duesseldorf.de solving a Smullyan puzzle

3.3.2 Lessons learned: get communication right

The first version of the logic calculator was implemented as a PHP web page which called the ProB CLI via Common Gateway Interface. This had the drawback of startup time: a new ProB CLI was started after every evaluation request of the user, leading to a noticeable lag. The ProB Java API enabled us to develop a more flexible website, without noticeable startup time. The formulae are actually evaluated as the user types them.

In consequence, the ProB Java API enabled an easy embedding of formal methods technology into a web service. The logic calculator requires no installation effort, and can be run from any device with a browser. It can be useful for first experiments in the B language and to check out the capabilities of ProB's solver.

The recently developed kernel for Jupyter based on the ProB Java API [31] provides a more powerful notebook interface. It is an evolution of the logic calculator, enabling to mix B formulas with text and visualisations, but it requires more effort from the end user to set up. Here, the ProB Java API becomes essential: processing a computational notebook requires multiple calls to ProB, with dependence upon earlier results.

3.4 DSLs on top of B: lisb

The B language is rather inflexible, e.g., the original dialect only supports let and if-then-else constructs for *statements*. In the context of expressions, these features are not available. As an example, it is not possible to retrieve the absolute value of a number by writing `x := IF x > 0 THEN x ELSE -x END`.[3]

Another issue is that the B language does not offer a proper macro system. The only means to define B snippets and use them in different places is via C-preprocessor-like macros refered to as *definitions*. These definitions are however not satisfactory, e.g., operator precedences

[3] In recent versions of ProB, this is possible due to improvements discussed in Sect. 3.5.1.

```
user=> (eval (to-ast (b (= (* 2 :x) (+ 1 2 3)))))
{"x" 3}
```

Listing 2 Solving a predicate on a clojure REPL

are not always clear and variable identifiers may be captured by accident, which may result in erroneous replacements.

This combination of inflexibility and wonky definitions system lead us to work on lisb: lisb is an experiment that aims to leverage the syntactical flexibilities of a lisp-like language—in this case, Clojure. The key concept is that all B operators of the predicate sub-language (i.e., there is no support for state machines) are implemented in Clojure. Each operator is a pure function that generates a part of the AST (abstract syntax tree) that is used to solve the predicate that is formulated by the user. Then, several parts of the AST can be re-combined before it is sent to the ProB constraint solver.

To continue the example of the if-then-else expression earlier, one could write the absolute value function according to the re-writing rule given in [37]:

$$(\lambda t.(t \in \{\texttt{TRUE}\} \wedge (x > 0) | x) \cup \lambda t.(t \in \{\texttt{TRUE}\} \wedge \neg(x > 0) | -x))(\texttt{TRUE})$$

As this construct is not very readable, it might be preferable to write a function `myifte` *once* (using lisb) that *generates* this corresponding AST, and then call it via `(myifte (> x 0) x (- x))` instead.

Overall, this approach gives rise to new, flexible DSLs on top of B, as one can easily write pure functions that return a new AST, potentially combining many operator usages to complex instructions. At the same time, DSLs can allow users to handle, explore and work with the results. Another aim was to explore whether this gives a viable approach for the case study presented in Sect. 3.5.

Listing 2 shows how a simple predicate such as $2 * x = 1 + 2 + 3$ can be represented in lisb and solved by ProB. The form contained in `(b ...)` is rewritten by a macro into the code depicted as "Language Frontend" in Fig. 5. This code is then executed in order to create an intermediate representation, also as shown in Fig. 5. Finally, `to-ast` creates a ProB-specific AST, and `eval` evaluates it using the ProB Java API.

The main idea is that both the language frontend and tool-specific AST can be changed in order to become more independent of B. For example, predicates may be written in a syntax that is closer to other formalisms, e.g., Alloy or SMT, or more specific to a given problem. The intermediate representation only holds for mathematical information unrelated to any formalism. Finally, translations can be provided in order to use tools other than ProB, with the hope that, eventually, (most) predicates might be solved using Z3 [23] or other solvers.

A more involved example is given in Listing 3, which creates the mathematical constraints required to solve the well-known n-queens problem. How the constraints exactly describe the problem is not relevant here; what is interesting is that some expressions that are used repeatedly can be assigned to identifiers, such as the integer interval `width`. Each form in the code creates part of the intermediate representation and is combined in order to create the entire predicate. Also, the predicate can be instantiated with a size and a (partial) solution for the problem, returning a new intermediate representation in turn. It can be used to find a solution individually, that might in turn be re-used as input for other predicates. Alternatively, the returned predicate may be combined with other predicates as well.

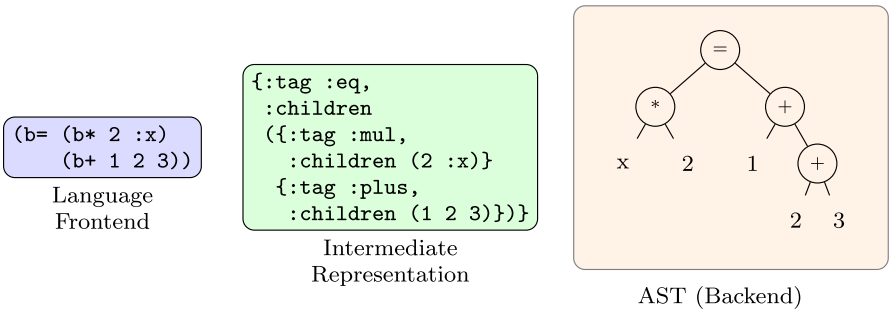

Fig. 5 Frontend, intermediate and backend representation of predicate in Listing 2

```
(defpred nqueens-p [size sol]
 (let [width (range 1 size)    ;; AST blocks
       q1pos (apply sol :q1)   ;; that are repeated
       q2pos (apply sol :q2)] ;; and can be reused
  (and (member? sol (>-> width width))
       (forall [:q1 :q2]
         (=> (and (member? :q1 width) (member? :q2 width)
                  (> :q2 :q1))
             (and (not= (+ q1pos (- :q2 :q1)) q2pos)
                  (not= (+ q1pos (- :q1 :q2)) q2pos)))))))
```

Listing 3 Definition of N-Queens in lisb

3.4.1 Main contribution: exploring domain specific languages

The ProB Java API and lisb integrate very nicely: under the hood, ProB ASTs are generated and send to ProB in order to find solutions. The aforementioned value translator then translates the results into Java data structures, which can be used in Clojure as well. In the other direction, it would be cumbersome to write a parser with a proper macro system oneself. Here, lisb takes the advantages of Clojure, or rather Lisp, and extends them into the (B-like) input language. This renders it very easy to create a domain specific language on top of B.

3.4.2 Lessons learned: predicates are not everything

Ultimately, lisb only covers the subset of expressions and predicates in B. While the initial state, where all formulas are evaluated, can be set-up individually, state transitions are not (yet) considered. Yet, one of the main advantages of B is not only the expressiveness of the language, but also the usage of a clearly defined state machine. Thus, for a real application, lisb was not sufficient, and more involved usage of the API were necessary.

3.5 ProB as a constraint solver: PlüS

PlüS [69] is an application for planning university timetables [79,80]. The goal is to show that it is possible for students to finish their studies in legal standard time for all courses or combinations of major and minor subjects. If a course or a combination is found to be infeasible, the smallest conflicting set of classes and time frames should be provided such that it can be fixed manually. This process is started from the current timetables. Complete

re-generation of timetables is avoided due to informal agreements, e.g., lecturers prefer given time slots or are unavailable on certain days.

A database stores information about all courses, e.g., for which subject they can be attributed, whether they are mandatory or if other courses are prerequisites. From this database, a B model is compiled. This is included in another B machine that allows checking for feasibility of a subject, move lectures etc. from one time-slot to another and to calculate the unsatisfiable core if applicable.

The formal model is the foundation for a GUI in JavaFX. The initial state is the initial timetable setup. Each course and combination can be checked individually, which triggers the state transition that checks feasibility. If the B model returns that there are conflicts, they are highlighted in the GUI. Then, the user can move courses to different time slots and re-calculate. This is done via drag-and-drop and, again, triggers the corresponding operation in the B machine.

If a course works out with the current scheduling, the state variable that represents the timetable is used to generate PDF files containing a default timetable that can be given to students, so they know in what semester they should attend which courses.

In this application, the interaction with ProB is hidden from the user, i.e., they do not need to know about formal methods, states and transitions. It is currently used by the University of Düsseldorf.

3.5.1 Main contribution: improving the B eco-system

PlüS was one of the earlier projects that used the ProB Java API extensively in the way presented. In particular, the value translator that translates B values into Java data structures, which is used in the other case studies, was created during the development of PlüS. Furthermore, certain shortcomings of B were identified: if-then-else statements are only available for substitutions, but not in the predicate and expression sub-language. Similarly, it is not possible to use `let`-like syntax to locally capture values for any identifier. These have been addressed in newer versions of ProB, which extend the syntax of B in these ways.

Lastly, it is hard to express function-like constructs that calculate values that can be used in predicates. B offers definitions, which offer a macro system similar to the C-preprocessor with all its shortcomings, e.g. shadowing of variable identifiers, which are unacceptable in a formal language. Currently, we work on a language extension for ProB that allows a more sophisticated construct to implement pure functions.

3.5.2 Lessons learned: model interaction

Interacting with the model can be quite cumbersome: in particular, feeding information from scratch into the model can be slow or very complex. Instead, it is easier to generate a large model containing all information.

Initially, the idea was to work on pure predicates without a state machine in order to find scheduling conflicts. However, the aforementioned shortcomings in the language resulted in large predicates with many repetitions that were hard to debug. We found that incorporating the information into a state machine with given operations for manipulation of the schedule is more sensible. Additionally, this offers a simple undo-feature by reverting the trace to an earlier state.

3.6 Real time animation: ETCS hybrid level 3 concept

We also used the ProB Java API in an industrial project, for a demonstrator of the ETCS (European Train Control System) HL3 (hybrid level 3) principles. HL3 is a novel approach to increase the capacity of the railway infrastructure, by allowing multiple trains to occupy the same track section. This is achieved by dividing the track sections into virtual subsections (VSS). While the status of the track sections is determined by existing wayside infrastructure (axle counters or track circuits), the status of the VSS is computed from train position reports.

In this application, the formal model was used as a component at runtime to control real trains in real time. This can be seen in the video presenting the technology at https://www.youtube.com/watch?v=FjKnugbmrP4&t=163, where in the lower center one can see the visualisation (using the ProB Java API) of the formal model. The visualisation shows that two trains occupy the same track section, but occupy disjoint virtual subsections.

The core of HL3 was written as a B model, managing the status of said virtual subsections. For the overall demonstrator, the HL3 model was interfaced with other real-world hardware components:

- an interlocking (IXL) which manages the signals and the status of the track sections,
- a Thales Radio Block Centre (RBC) which communicates with the trains and grants movement authorities
- and an Operation and Maintenance Server (OMS).

These three components fed information into the model via ProB Java API in order to drive the formal model.

The model itself is non-deterministic. Based on the inputs from the external sources, the corresponding operation is chosen. After updating the state of the model, the successor state is passed to a consumer in Java that in turn sends information to the IXL, OMS and RBC. The VBF application, comprising ProB, ProB Java API and the B formal model, performs well enough on a regular notebook computer for a real-life demonstration involving the management of actual trains on their VSS. More details about the model and the demonstrator can be found in [38,39].

The overall architecture of the VBF demonstrator is very similar to the Pac-Man example. The Pac-Man board can be seen equivalent to the railtrack topology, and the Pac-Man behaves similar to trains, as they move based on external input. Instead of only visualising the model state to the user, additionally the application reacts to it and communicates with other components.

3.6.1 Main contribution: application based on model alone

The ETCS case study fully relies on an embedded model rather than on code generation. By doing so, it has proven our approach to be both feasible and efficient in a real-world application. The overall development time was low when compared to manual or automated code generation. In addition, the formal model was very close to the HL3 natural language requirements. Changes to the requirements and model could be quickly carried out. Indeed, the use of our demonstrator has uncovered over 40 issues in the original HL3 principles paper, which were corrected in the official document along with our formal model. Of course, a fully refinement-based approach ending with code generation would be able to prove the system correctness and hence deliver a higher level of certainty than our approach does. However, we believe that for prototypes and demonstrators, a model-checked and well-tested specification

that is directly executed can beat non-formal software development by a wide margin in terms of development time and costs.

3.6.2 Lessons learned: full-stack debugging workflow

One important benefit of our approach was that we could store the formal model's behaviour in log files and later replay these traces in the ProB animator. This allowed us to analyse suspect behaviours, fix the HL3 specification and model, and then check that the corrected model solved the uncovered issues by replaying the trace again. That is, we automatically got record and replay capabilities of debuggers as in [65].

4 Discussion: should formal specifications be executable?

When thinking of executing formal specifications, one usually has animation or code generation (cf. [32,90]) in mind. We think that the term "execution" of formal specification is somewhat overloaded; its semantics differ when considering animation and code generation techniques. Finally, embedding gives a new dimension of this.

Thus, in this section, we will first consider differences between those three approaches concerning executability (Sect. 4.1). Afterwards, we take another look at the B language in particular (Sect. 4.2). Finally, we revisit arguments made in past discussions [30,35,41] whether specifications should be executable or not in the first place in the content of embedding (Sect. 4.3).

4.1 Executability

As mentioned above, executability of a formal specification can refer to several constructs depending on the context.

First, *animation* of a formal specification is an important means to quickly find errors by executing certain scenarios. This can either be done manually or even replaying a given trace automatically. Executing a longer trace by hand and verifying whether each encountered state is correct is very cumbersome and might be aided by state visualisations.

The most noteworthy feature about animation is that it is a means to develop, debug and reason about the correctness of a specification. Usually, the user interacts with the tool directly and events are chosen by hand, even ones that should be picked by the environment. In our case studies, that includes movement of the ghosts in Pac-Man, moving the chess pieces of the enemy and providing the input of signals, points, etc.

When considering what it means to *execute* a model in this context, we can characterise it as follows:

- Computation of transitions does not need to be efficient (but it is preferable).
- Constraint solving may fail (or rather: time out), but execution may still continue when manually providing values satisfying the constraints.
- Animation is used during development and covers a large part of the language.

Second, *code generators* usually are applied to (a subset of) the language that offers precise executable semantics. In the case of the B language, this is usually a small subset named B0, which does not include functions, relations, deferred sets, etc. (cf. Sect. 5.2 for more details). This approach has several advantages and drawbacks:

– Since B0 is very limited and very close to, e.g., a subset of C, such concrete specifications can immediately be translated to suitable low-level constructs. Then, computation is efficient even for the generated code, in particular when compared to approaches that try to emulate higher-level constructs (though may be less efficient compared to hand-written code).
– Constraint solving cannot fail, since the language (subset) cannot express any constraints.
– Development of the model (usually) is finished once code generators are applied.
– The generated code can directly interface with other components.
– The input language severely lacks abstractions and expressiveness.

Finally, *embedding* is a hybrid approach that tries to combine the best of both worlds:

– Computation must be efficient if the specification demands it, but can be inefficient for proof-of-concept implementation.
– Constraint solving must not fail, since execution would stop in this case.
– Embedding of the specification can be used during development of a prototype as well as be shipped as a finished product.
– The embedded specification can interface with other (existing) components.
– As with animation, large parts of the input language are "executable", yet usually *efficient execution* is required.

Overall, executability has different meanings, depending on the context. During animation, the characteristic question is "Can a solution be found automatically or be supplied?", for code generation, it is "Can the model be translated?", while for our approach, the relevant key question is "Can the specification be executed sufficiently efficiently?" In the following, we use the latter meaning of the word.

4.2 B as an executable language

For all intents and purposes, the B language is, foremost, a specification language. It was never intended to be executed; a software requirement document should be translated into an abstract model that works "at a more abstract level *execution is no longer possible"* [2] (emphasis in original). In fact, as the B language has its roots in first-order logic, abstract constraints are not even decidable in general. Only a small, implementable subset called B0 (which will be discussed in more detail in Sect. 5.2) has defined executable semantics. Why bother trying to execute a more abstract specification then?

During refinement in the traditional workflow, the abstract model is gradually transformed into a concrete model, that at some point is intended to be executed, or rather that code is generated from. Yet, in order to validate that the behaviour of the abstract model is correct in the first place, tools offering animation capabilities are required. Otherwise, errors or inconsistencies in the specification document can easily end up in the final software product. These tools usually rely on constraint solvers or user interaction in order to determine values for execution.

The point that justifies re-visiting the discussion presented in Sect. 4.3 is that the following inherent limitation holds: not the entirety of the B language can be executed at all or in reasonable time. In these cases, either an efficient implementation would be algorithmically (nearly) impossible, or some refinement is required in order to state the problem in a way that the constraint solver is able to execute it. As argued, the consequence for shipping such a model to be used as part of software during run-time is different from animation during creation of the model and reasoning about its correctness. Thus, not just *any* specification

can be used in a standalone tool. Instead, a certain level of concreteness is required, whereas higher levels of abstraction become *feasible*.

4.3 Should formal specifications be executable?

The famous article by Hayes and Jones [41] has led to quite a bit of controversy. It argues that formal specifications should not be executable and gives several counterarguments (CA):

CA1 Proof is more important than (finite) execution,
CA2 Forms of usable specifications are restricted,
CA3 Executable specifications tend to be over-specified,
CA4 Execution is inefficient.

This does not mean that we disagree with these arguments. In the context of the executability discussed in Sect. 4.2, these arguments offer valid points *against* our approach. In the following, we want to consider these arguments and give our reasoning why we deliberately go against this judgement and use a high-level specification language such as B.

Firstly (CA1), we find formal proof to be very important. However, we have observed that for most formal specifications, which are more involved and are written to be executable (in the sense of animation), it is very hard and cumbersome to discharge proof obligations. On the other hand, models written to be proven usually are not executable (again, in the sense of animation) either. Yet, proof should always be complemented by animation in order to verify that not only the model is consistent in itself, but also describes the desired behaviour. This is a challenge for executability (in the sense of both animation and embedding) that indeed needs to be addressed in the future, may it be by improving the constraint solver that works on the set-theoretical foundations of B, or by searching for new techniques for provers. We think that, currently, we cannot offer embedding of a fully proven specification that is sufficiently complex. Yet, trying to prove unsound specifications may result in a counterexample that quickly raises awareness of an error that may not be uncovered (quickly) by testing or model checking.

Secondly (CA2), as discussed in Sect. 4.2, B is a language that is very high-level and allows writing non-executable specifications (as one could encode a non-decidable problem in a single state transition). Instead of worrying about these issues, we try to provide an approach for specifications an animator can handle and execute (efficiently).

Surprisingly, most B and Event-B specifications can be animated with ProB. The major exceptions are mathematical models involving infinite domains, and some axiomatic specifications which require infinite models (e.g., algebraic specification of a stack). Also, sometimes users add complicated axioms to their model: this makes proof easier but animation more complicated. Thus, it can be necessary to separate proof axioms from animation axioms. In [21] we developed the prob-ignore pragma, to annotate proof axioms not necessary for animation. Animation configuration is kept in separate refinements of the proof models, which allows animators and provers to co-exist peacefully on the same development. Animation ensures no inconsistency in model, proof ensures we scale to all instances/topologies.

Thirdly (CA3), over-specification does not seem to be an issue for our use case. An example from [41] is a sorting algorithm. In B, this can be calculated by the constraint solver by purely specifying the *property* what it means for a sequence to be sorted. An example for a valid B predicate that can be solved by ProB in order to yield a sorted sequence is given in Fig. 6. Note that no concrete implementation is specified, as the problem is solved declaratively. Moreover, in the typical workflow of the B-Method, a concrete implementation

$$
\begin{aligned}
&input = [12, -3, 42, 7] \wedge && \text{(input sequence)}\\
&output \in 1..size(input) \rightarrow ran(input) \wedge && \text{(type of output)}\\
&\forall e \in ran(input) \cdot (card(input \rhd \{e\}) = card(output \rhd \{e\})) \wedge && \text{(keep elements)}\\
&\forall i \cdot 1 \leq i < size(input) \implies output(i) \leq output(i+1) && \text{(ordering)}
\end{aligned}
$$

Fig. 6 Sorting predicate

happens during refinement. Thus, the writer of the specification is usually *able to choose* the level of abstraction herself.

The last argument concerning performance (CA4) is carefully reviewed for each of our case studies individually in Sect. 3 and overall in Sect. 7. As different performance constraints are given on a case-by-case basis, it is hard to classify specifications that are suitable to be embedded.

5 Related work

There are several tools that are able to achieve part of the case studies presented, e.g., state visualisation tools and code generators that we will present in Sects. 5.1 and 5.2, respectively. Some additional applications that other researchers and industrial practitioners already built on top of the ProB Java API will be discussed briefly in Sect. 5.3. Finally, there are approaches that work very similar to ProB Java API. We will take a look at those in Sect. 5.4.

5.1 Visualisation

All the presented projects include a GUI which displays a visualisation of the current state. State visualisation by itself is a useful tool to understand the application state more easily and is often used during the development of a model, debugging, and also to explain it to a domain expert.

BMotionWeb [53,54] is a tool for state visualisation based on web technologies. It also builds upon the ProB Java API and allows simple interaction with the model. The chess example from Sect. 3.2 uses this tool both for visualisation and embedding the script that controls the AI. A heavy disadvantage however is the complex technology stack: BMotionWeb builds upon ProB Java API and uses Groovy, SVG, JavaScript and HTML5, where each component of the stack may go wrong, rendering development very cumbersome. Thus, a more simple successor was developed called VisB [92]. It also builds upon ProB Java API, but is easier to use and maintain.

State visualisation is not unique to the B formalisms: e.g., another tool that allows visualisations based on web technologies is WebASM [94], which works on top of CoreASM [24]. CoreASM is a tool that can be used to execute abstract state machines (ASM). Another advanced visualisation framework is PVSio-Web [91] for PVS. Also, for Event-B a series of other visualisation tools were developed, such as Brama, AnimB and JEB [93] which includes a JavaScript interpreter for B.

```
BSet<BInteger > _ic_set_0 = BSet<BInteger >();
for(BInteger _ic_x :
        (BSet<BInteger>::interval((BInteger(0)),
                                  (BInteger(2147483647))))) {
    if((_ic_x.less((BInteger(3)))).booleanValue()) {
        _ic_set_0 = _ic_set_0._union(BSet<BInteger >(_ic_x));
    }
    ...
}
```

Listing 4 Java code generated by B2PROGRAM

5.2 Code generation

A more traditional approach is to generate (low-level) code based on the specification. Translation tools usually cannot work on most constructs that high-level formalisms have to offer, e.g. calculation of an appropriate parameter for an operation, set comprehensions or solving quantifications usually require constraint solving techniques which are infeasible to generate.

A popular implementation-level subset of B is named B0 [1,20], from which translation into an imperative language is fairly straightforward. Many features of the B language are missing though, including many operators on functions, relations and sets as well as quantifications.

For B and Event-B, several code generators exist. One such code generator is C4B which is integrated in Atelier B [20]. It allows generation of C code from the implementation level subset of B (i.e. B0). However, refining a model of industrial size down to B0 is a notably cumbersome task to do. Another code generator that is capable to cope with a subset of B0 is b2llvm [10] that generates LLVM code. A notable toolset for Event-B is EB2ALL [64], which allows code generation to several languages including C and Java.

Another approach attempts code generation from a higher level of abstraction. Modern programming languages offer, e.g., sets and maps, which allows easy translation of B constructs such as sets, relations and functions. Supporting code generation for more constructs from B that are not included in B0 might make an approach using code generation more feasible. One such code generator is EventB2Java [17,75] that translates higher-level constructs. More recently, B2PROGRAM [89] was presented.

When translating aforementioned data structures, one loses an important property of the resulting program: execution is not necessarily possible in constant memory, i.e., without dynamic memory allocation. B0, on the other hand, is intended to be generate code suitable for, e.g., embedded systems, as failure to allocate memory is not handled as part of the formal method. This aligns with the approach we present in this article: the overall idea is that prototypes can be executed during early development stages.

However, there is one fundamental difference to code generation: performance often is drastically better due to access to PROB's constraint solver. Generated code must, e.g., enumerate and apply a filter predicate to all possible integer values. As an example, the set $\{x|x \in NAT \land x < 3\}$ is calculated by the code generated by B2PROGRAM shown in Listing 4. The code is far from optimal: since the range of 32-bit (signed) natural numbers is very large, computation of the set is very slow. While this approach is feasible if the domain is small enough, usually application of constraint solving techniques is far more preferable, especially on large or even unbounded domains.

Code generation is not exclusive to the B method: e.g., VDM specifications can be used to generate C++ or Java code [49]. In particular, higher-level constructs such as set comprehensions are handled by this translation as well. Moreover, when targetting Java, this code generator can also translate pre- and postconditions to JML (Java Modelling Language) annotations, that allow optional checks of correctness of the system realisation [88].

5.3 Other tools

A variety of third-party tools have been developed using the ProB Java API, highlighting that it can be used as good way to build tools.

- The HRemo tool for invariant discovery, where ProB is used to produce traces with undesirable states, fed to the machine learning system HR. HRemo is described in detail in chapter 4 of [76] and used, e.g., in [36].
- The VTG (Vulnerability Test Generator) system [77] is based on an Event-B model of the JavaCard bytecode operations. VTG then creates mutants of those JavaCard operations using the ProB Java API, and then uses ProB to generate test traces to exercise those mutants. Note that the generation of the mutants with ProB Java API was orders of magnitude faster than the original code within Rodin.
- CODA [12] is a refinement-based framework for modelling component-based embedded systems. The animation and simulation features were implemented via access to the ProB Java API.
- Cucumber Event-B https://github.com/tofische/cucumber-event-b is a tool to execute test scenarios described in the Gherkin language for Event-B models. It is used for high-level assurance tests [26].
- Meeduse http://vasco.imag.fr/tools/meeduse/ is a tool for domain specific languages building upon EMF (Eclipse Modelling Framework). The domain specific languages are translated to B and the operational semantics realised with ProB. Meeduse has been applied, e.g., to develop a domain specific language for the railway domain [45].
- The VDM interpreter of the Overture tool [56] was integrated with ProB in order to execute implicit VDM specifications [58]. In particular, ProB's constraint solving capabilities are used to find solutions for parts of the specification, that are not in the executable subset of the VDM specification language. This integration makes use of an early version of the ProB Java API. Due to lessons learned from other projects and performance improvements, especially concerning record types (as they were used in PlüS, presented in Sect. 3.5), the current ProB Java API could render similar integrations with other tools much easier.

Other tools using ProB Java API exist, e.g., the data validation tool Rubin developed in collaboration between Thales and the STUPS group.

5.4 Other approaches

Another formal specification language is part of the Vienna Development Method (VDM) [48]. A well-known tool for VDM is Overture [56], which implements an interpreter in Java. In [66], an extension to the VDM language and Overture was presented. It allows execution of Java code from VDM specifications and, in turn, to control the interpreter to evaluate expressions in the current state. The goal is to add visualisation of the current state to the model and to integrate models with legacy systems, as we did, e.g., in Sect. 3.6.

An application that can also be understood as "execution" of a formal model is co-simulation [33]. Formalisms usually differ in their application area: e.g., B is a formalism used to model discrete events, whereas behaviour regarding continuous time is hard or impossible to express. Using interfaces such as FMI [9], one can combine several models in different formalisms. A co-simulation orchestration engine, such as Maestro [85], usually manages the passage of time and synchronises the execution of all models. Such a component that is orchestrated can be either be a *tool-wrapping FMU* (functional mock-up unit) or a *generated FMU*. Tool-wrapping FMUs implement the FMI in a way that tool exposes the behaviour of the model to the interface, such that a high-level model can be used. An example is the Overture FMU extension [86]. Generated FMUs usually stem from a model and are exported by a tool. Then, the generated C code represents a dynamic system and implements the interface of the standard. Again, the Overture is able to generate such FMUs [7].

Built on top of FMI, INTO-CPS [57] focuses on co-simulation with pragmatic integration of current industrial-strength tools. It also offers, amongst others, model checking, hard- and software-in-the-loop simulations. INTO-CPS also provides usage different levels of abstraction of the models [87], and gives rise to visualisation techniques such as augmented reality. One future endeavour is to create models of the physical world as a "digital twin", in order to simulate complex cyber-physical systems [27].

Another approach [67] is interesting as well: in his thesis, Nummenmaa executes several example runs on probabilistic specification of games. The idea is to leverage non-determinism in order to simulate and analyse game design. For this, the DisCo method [47] is utilised. Yet, in these simulations, the model does not interact with an environment.

6 A look into the crystal ball—potential for the future

In the previous sections, we have presented several applications that already use the ProB Java API and discussed the circumstances, under which we deem such an approach reasonable. Now, we want to take a look into the crystal ball and discuss some potential use cases that we did not implement yet, but seem very promising. We discuss integrations with more sophisticated artificial intelligence than the one used in the Pac-Man or Chess case studies and link to existing research on that topic in Sect. 6.1. Finally, we will name some other future tools that ProB Java API allows us to implement in Sect. 6.3.

6.1 Integration potential with artificial intelligence

As already mentioned in the discussion of the chess case study in Sect. 3.2, the approach of defining custom AI to control ProB's exploration strategy corresponds to the notion of directed model checking. Using AI heuristics for directed model checking is already a researched approach in the community, for instance utilising AI planning heuristics [52], Monte-Carlo Tree Search [70], or relaxation techniques for heuristic finding [82]. However, the integration with executable specifications allows for a more general notion of directed model exploration and analysis.

As the ProB Java API offers the possibility to simulate modified copies of a model in-memory, it is also possible to introduce a self-repairing model checking routine. Combining regular model checking techniques with a constraint-based repair method as outlined in [78] would allow a ProB Java API based controller to alter the model once a violation is found by applying generated repair suggestions. The model checking could then continue on the

repaired model to search for possible further violations. This can be done for multiple possible repair suggestions, allowing to directly discard faulty ones automatically or present the user a set of viable options once all states were explored.

On another note, a component-wise integration as proposed in [40] is also easily possible with ProB Java API. The idea is to compose independent executable specification components (ES-only components) with AI-only components, where both, the ES and AI components only address particular subgoals of the overall system. The refinements in this approach are done by extending the system with further components or by splitting components into smaller ones. The splitting step allows for substitution of a subgoal of the original component for an implementation of the respective other type, i.e., splitting an ES component into three subcomponents where one is implemented as AI component. Vice-versa, (partially) replacing AI components with ES components allows for a more provable system. Hence, applying this approach to an initial AI-only system might increase the provable guarantees of the system at least for certain subgoals while not having to switch to a fully formal workflow.

6.2 Tool-wrapping FMU

The B language itself has no notion of time: all events are discrete and happen instantaneously. Yet, often time constraints are important and are expressed by state variables. One strategy is to add a time variable that is only incremented by certain events. The drawback is that this way, the state space usually becomes infinite and, thus, exhaustive explicit-state model checking is impossible. Another way to model time is to maintain a collection of deadline timers that count down instead. This is method is described and used in [14,38,62,74].

The ProB Java API allows users to superimpose any notion of time on their model, whatever modelling strategy is used—or manage time outside of the B model. Thus, it is feasible to implement the FMI standard as an individual tool-wrapping FMU (which, however, requires the development of C glue code via Java's native interface). It would also be sensible to explore how continuous behaviour can be modelled and verified as well, making use of hybrid automata [42].

6.3 Future use cases

Embedding a formal model directly into applications has several benefits that might aid enabling future use cases.

First, the model is closer to the actual hardware. This allows it to be included in real-time simulations of the system, including all components and the actual (rather than a modelled) environment. This also allows usage of formal models for hardware-in-the-loop tests, which are common for instance in the automotive industry (cf. [25,81]). Having the model included in full system-level tests should help remedy some concerns regarding the loss of fully formal proof.

Second, as part of a regular application, formal models can easily be accessed programmatically. This enables new kinds of analyses not readily available in current formal methods toolchains. For instance, the ProB Java API can be used to define, execute and analyse test case as well as user usage scenarios. This is especially handy, when formal modelling in used together with specifications including classical use case definitions. Ultimately, the ProB Java API can be used to connect formal models with frameworks such as JUnit and thus enables a tighter integration of formal methods and non-formal development. In particular,

it might be easier to formulate test cases in the sense that they can be expressed in a way that is closer to the specification or more involved than, say, an LTL formula.

7 Conclusions

In this paper, we presented the ProB Java API, which offers an easy to use interface to the ProB animator and model checker. The ProB Java API renders it possible to write applications that interact with a formal model at runtime, offering declarative programming, rapid prototyping and easy debugging. Furthermore, we embedded formal models into actual applications and investigated this approach via five different case studies. We also considered counterarguments regarding executable specifications and re-evaluated them given the gained experiences.

Overall, we can draw the following conclusions:

- We think that specifications can and should indeed be executable, as it allows verification of an interpretation or an implementation against the specification. Given a suitable high-level specification language, many counterarguments such as over-specification do not hold. With a tool as presented in Sect. 2 or in [66], it is possible and (often) viable to use that specification as a library in an application, allowing embedment of declarative programming into traditional, imperative programming languages.
- Development of complex components is significantly eased by the level of abstractions provided by a high-level specification language, such as B. Integration with existing code, written in other programming languages or running on different machines, is very useful. When adapting the formal model, changes can immediately be evaluated via a test scenario in the context of an entire application. In contrast, adapting a traditional implementation is more cumbersome and more prone to introducing new, unrelated bugs. Tool support such as model checking or animation proved to be invaluable to uncover errors early on which may otherwise have gone unnoticed for a longer time.
- The main concern for real-life applications, as already stated in 1989 by [41], is performance. Low-level applications written in traditional imperative, functional or even logical programming languages can be orders of magnitudes faster because they can work at lower levels of abstraction. Hence, for many time-critical applications the execution of formal specifications is not the way to go yet. However, as long as performance requirements are reasonable (e.g., if data sets are rather small), utilising formal models at runtime allows us to quickly deploy complex applications that can make use of the ecosystem associated with formal methods, from proof to animation and model checking.
- The presented case studies clearly show that the integration of formal models in a typical software development life cycle is possible. Yet, since the entirety of the API can be accessed, the ProB Java API allows for applications that are way more involved and may prove to be the foundation for game changers concerning use cases and accessibility of formal methods. We think that this approach is just scratching the surface of what is possible, especially regarding the integration with AI components, and we are excited to see what academic and industrial usages may emerge.

Acknowledgements We thank Christoph Heinzen and David Geleßus for authoring and improving the presented Pac-Man application, as well as Philip Höfges for the chess model, AI and GUI. Additionally, we want to thank the many people who were involved in the development of both ProB and ProB Java API, the Slot Tool and the ETCS Hybrid Level 3 case study.

Funding Open Access funding enabled and organized by Projekt DEAL. The HL3 case study in Sect. 3.6 was funded by Thales.

Code availability All code of ProB Java API and our public case studies is available on GitHub: ProB *Java API Maven artifacts source code* https://github.com/hhu-stups/prob2_kernel ProB *Java API Maven artifacts* https://search.maven.org/artifact/de.hhu.stups/de.prob2.kernel *API example* https://github.com/hhu-stups/executable_spec_example *Pac-Man plug-in* https://github.com/pkoerner/EventBPacman-Plugin *Chess* https://github.com/pkoerner/b-chess-example *Logic Calculator* https://github.com/hhu-stups/prob-logic-calculator *lisb* https://github.com/pkoerner/lisb *PlüS* https://github.com/plues/plues ProB *Jupyter Kernel* https://gitlab.cs.uni-duesseldorf.de/general/stups/prob2-jupyter-kernel The code implementing the HL3 case study is confidential and cannot be disclosed here.

Compliance with ethical standards

Conflicts of interest The authors declare that they have no conflict of interest.

References

1. Abrial JR (1996) The B-book: assigning programs to meanings. Cambridge University Press, Cambridge
2. Abrial JR (2006) Formal methods in industry: achievements, problems, future. In: Proceedings of the 28th international conference on software engineering, pp 761–768
3. Abrial JR (2010) Modeling in Event-B: system and software engineering, 1st edn. Cambridge University Press, Cambridge
4. Abrial JR, Lee MK, Neilson D, Scharbach P, Sørensen IH (1991) The B-method. In: Proceedings VDM, LNCS, vol 552. Springer, pp 398–405
5. Back R (1981) On correct refinement of programs. J Comput Syst Sci 23(1):49–68
6. Back RJ, Wright J (2012) Refinement calculus: a systematic introduction. Springer, Berlin
7. Bandur V, Tran-Jørgensen PW, Hasanagic M, Lausdahl K (2017) Code-generating VDM for embedded devices. In: Proceedings of the 15th overture workshop, School of Computing Science technical report series, vol 1513. School of Computing Science, University of Newcastle upon Tyne, pp 1–15
8. Biere A, Cimatti A, Clarke EM, Strichman O, Zhu Y et al (2003) Bounded model checking. Adv Comput 58(11):117–148
9. Blochwitz T, Otter M, Akesson J, Arnold M, Clauss C, Elmqvist H, Friedrich M, Junghanns A, Mauss J, Neumerkel D et al (2012) Functional mockup interface 2.0: the standard for tool independent exchange of simulation models. In: Proceedings MODELICA, 076. Linköping University Electronic Press, pp 173–184
10. Bonichon R, Déharbe D, Lecomte T, Medeiros V (2014) LLVM-based code generation for B. In: Proceedings SBMF, LNCS, vol 8941. Springer, pp 1–16
11. Butler M, Leuschel M (2005) Combining CSP and B for specification and property verification. In: Proceedings FM, LNCS, vol. 3582. Springer, pp 221–236
12. Butler MJ, Colley J, Edmunds A, Snook CF, Evans N, Grant N, Marshall H (2013) Modelling and refinement in CODA. In: Proceedings refine, vol 115. Open Publishing Association, pp 36–51
13. Cansell D, Méry D (2012) Foundations of the B method. Comput Inform 22(3–4):221–256
14. Cansell D, Méry D, Rehm J (2007) Time constraint patterns for event B development. In: Proceedings B, LNCS, vol 4355. Springer, pp 140–154
15. Carlsson M, Ottosson G, Carlson B (1997) An open-ended finite domain constraint solver. In: Proceedings PLILP, LNCS, vol 1292. Springer, pp 191–206

16. Carlsson M, Widen J, Andersson J, Andersson S, Boortz K, Nilsson H, Sjöland T (1988) SICStus prolog user's manual, vol 3. Swedish Institute of Computer Science, Kista
17. Cataño N, Rivera V (2016) EventB2Java: a code generator for event-B. In: Proceedings NFM, LNCS, vol 9690. Springer, pp 166–171
18. CENELEC: railway applications—communication, signalling and processing systems—software for railway control and protection systems. Tech. Rep. EN50128, European Standard (2011)
19. Clark J, Bendisposto J, Hallerstede S, Hanse D, Leuschel M (2016) Generating event-B specifications from algorithm descriptions. In: Proceedings ABZ, LNCS, vol 9675. Springer, pp 183–197
20. ClearSy: Atelier B, user and reference manuals. Aix-en-Provence, France (2016). http://www.atelierb.eu/
21. Comptier M, Leuschel M, Mejia L, Perez JM, Mutz M (2019) Property-based modelling and validation of a CBTC zone controller in Event-B. In: Proceedings RSSRail, LNCS, vol 11495. Springer, pp 202–212
22. de Azevedo Oliveira D, Medeiros V, Déharbe D, Musicante MA (2019) BTestBox: a tool for testing B translators and coverage of B models. In: Proceedings TAP, LNCS, vol 11823. Springer, pp 83–92
23. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Proceedings TACAS, LNCS, vol 4963. Springer, pp 337–340
24. Farahbod R, Gervasi V, Glässer U (2007) CoreASM: an extensible ASM execution engine. Fundamenta Informaticae 77(1–2):71–103
25. Fathy HK, Filipi ZS, Hagena J, Stein JL (2006) Review of hardware-in-the-loop simulation and its prospects in the automotive area. In: Modeling and simulation for military applications, vol 6228. SPIE
26. Fischer T, Dghaym D (2019) Formal model validation through acceptance tests. In: Proceedings RSSRail, LNCS, vol 11495. Springer, pp 159–169
27. Fitzgerald J, Larsen PG, Pierce K (2019) Multi-modelling and co-simulation in the engineering of cyber-physical systems: towards the digital twin. In: From software engineering to formal methods and tools, and back, LNCS, vol 11865. Springer, pp 40–55
28. Fraenkel A (1922) Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. Mathematische Annalen 86(3):230–237
29. Fraenkel AA, Bar-Hillel Y, Levy A (1973) Foundations of set theory, vol 67. Elsevier, Amsterdam
30. Fuchs NE (1992) Specifications are (preferably) executable. Softw Eng J 7(5):323–334
31. Gelessus D, Leuschel M (2020) ProB and Jupyter for logic, set theory, theoretical computer science and formal methods. In: Proceedings ABZ 2020, LNCS, vol 12071. Springer, pp 248–254 (2020)
32. Ghezzi C, Kennerer RA (1991) Executing formal specifications: the ASTRAL to TRIO translation approach. In: Proceedings TAV. ACM, pp 112–122
33. Gomes C, Thule C, Broman D, Larsen PG, Vangheluwe H (2017) Co-simulation: state of the art. arXiv:1702.00686v1
34. Google Guice Repository. https://github.com/google/guice. Accessed 27 Feb 2020
35. Gravell A, Henderson P (1996) Executing formal specifications need not be harmful. Softw Eng J 11(2):104–110
36. Grov G, Ireland A, Llano MT (2012) Refinement plans for informed formal design. In: Proceedings ABZ, LNCS, vol 7316. Springer, pp 208–222
37. Hansen D, Leuschel M (2012) Translating TLA+ to B for validation with ProB. In: Proceedings IFM, LNCS, vol 7321. Springer, pp 24–38
38. Hansen D, Leuschel M, Körner P, Krings S, Naulin T, Nayeri N, Schneider D, Skowron F (2020) Validation and real-life demonstration of ETCS hybrid level 3 principles using a formal B model. STTT 22:315–332
39. Hansen D, Leuschel M, Schneider D, Krings S, Körner P, Naulin T, Nayeri N, Skowron F (2018) Using a formal B model at runtime in a demonstration of the ETCS hybrid level 3 concept with real trains. In: Proceedings ABZ, LNCS, vol 10817. Springer, pp 292–306
40. Harel D, Marron A, Rosenfeld A, Vardi M, Weiss G (2019) Labor division with movable walls: composing executable specifications with machine learning and search (Blue Sky Idea). In: Proceedings AAAI, vol 33. Association for the Advancement of Artificial Intelligence, pp 9770–9774
41. Hayes IJ, Jones CB (1989) Specifications are not (necessarily) executable. Softw Eng J 4(6):330–339
42. Henzinger TA (2000) The theory of hybrid automata. In: Verification of digital and hybrid systems. Springer, pp 265–292
43. Hickey R (2020) A history of Clojure. In: Proceedings of the ACM on programming languages, vol 4 (HOPL), pp 1–46
44. Hoare CAR (1978) Communicating sequential processes. In: The origin of concurrent programming. Springer, pp 413–443
45. Idani A, Ledru Y, Wakrime AA, Ayed RB, Bon P (2019) Towards a tool-based domain specific approach for railway systems modeling and validation. In: Proceedings RSSRail, LNCS, vol 11495. Springer, pp 23–40

46. Iliasov A, Lopatkin I, Romanovsky A (2013) The SafeCap platform for modelling railway safety and capacity. In: Proceedings SAFECOMP, *LNCS*, vol 8153. Springer, pp 130–137
47. Jarvinen H, Kurki-Suonio R, Sakkinen M, Systa K (1990) Object-oriented specification of reactive systems. In: Proceedings ICSE. IEEE, pp 63–71
48. Jones CB (1990) Systematic software development using VDM, vol 2. Prentice-Hall, Upper Saddle River
49. Jørgensen PWV, Larsen M, Couto LDMD (2015) A code generation platform for VDM. In: Proceedings of the 12th overture workshop, School of Computing Science Technical report series, vol 1446. School of Computing Science, University of Newcastle upon Tyne, pp 21–35
50. Knuth DE, Moore RW (1975) An analysis of alpha-beta pruning. Artif Intell 6(4):293–326
51. Körner P, Bendisposto J, Dunkelau J, Krings S, Leuschel M (2019) Embedding high-level formal specifications into applications. In: Proceedings FM, LNCS, vol 11800. Springer, pp 519–535
52. Kupferschmid S, Hoffmann J, Dierks H, Behrmann G (2006) Adapting an AI planning heuristic for directed model checking. In: Proceedings SPIN, LNCS, vol 3925. Springer, pp 35–52
53. Ladenberger L (2017) Rapid creation of interactive formal prototypes for validating safety-critical systems. Ph.D. thesis, HHU Düsseldorf
54. Ladenberger L, Leuschel M (2016) BMotionWeb: a tool for rapid creation of formal prototypes. In: Software engineering and formal methods, LNCS, vol 9763. Springer, pp 403–417
55. Lamport L (2002) Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley Longman Publishing Co., Inc, Boston
56. Larsen PG, Battle N, Ferreira M, Fitzgerald J, Lausdahl K, Verhoef M (2010) The overture initiative integrating tools for VDM. ACM SIGSOFT Softw Eng Notes 35(1):1–6
57. Larsen PG, Fitzgerald J, Woodcock J, Fritzson P, Brauer J, Kleijn C, Lecomte T, Pfeil M, Green O, Basagiannis S et al (2016) Integrated tool chain for model-based design of cyber-physical systems: the INTO-CPS project. In: Proceedings CPS Data. IEEE, pp 1–6
58. Lausdahl K, Ishikawa H, Larsen PG (2015) Interpreting implicit VDM specifications using ProB. In: Proceedings of the 12th overture workshop, School of Computing Science Technical Report Series, vol 1446. School of Computing Science, University of Newcastle upon Tyne, pp 6–20
59. Lecomte T, Burdy L, Leuschel M (2012) Formally checking large data sets in the railways. arXiv:1210.6815
60. Leuschel M, Bendisposto J (2011) Directed model checking for B: an evaluation and new techniques. In: Proceedings SBMF, LNCS, vol 6527. Springer, pp 1–16
61. Leuschel M, Butler M (2003) ProB: a model checker for B. In: Proceedings FME, LNCS, vol 2805. Springer, pp 855–874
62. Leuschel M, Mutz M, Werth M (2020) Modelling and validating an automotive system in classical B and event-B. In: Proceedings ABZ, LNCS, vol 12071. Springer, pp 335–350
63. Logic Calculators. https://web.archive.org/web/20120418155039/http://research.microsoft.com/en-us/um/people/lamport/tla/logic-calculators.html. Accessed 27 Feb 2020
64. Méry D, Singh NK (2011) Automatic code generation from event-B models. In: Proceedings SoICT. ACM, pp 179–188
65. Narayanasamy S, Pokam G, Calder B (2005) Bugnet: continuously recording program execution for deterministic replay debugging. In: ACM SIGARCH computer architecture news, vol 33. IEEE Computer Society, pp 284–295
66. Nielsen CB, Lausdahl K, Larsen PG (2012) Combining VDM with executable code. In: Proceedings ABZ, *LNCS*, vol 7316. Springer, pp 266–279
67. Nummenmaa T (2013) Executable formal specifications in game development: design, validation and evolution. Ph.D. thesis, University of Tampere
68. Plagge D, Leuschel M (2007) Validating Z specifications using the ProB animator and model checker. In: Proceedings IFM, LNCS, vol 4591. Springer, pp 480–500
69. PlüS. https://plues.github.io/en/index/. Accessed 27 Feb 2020
70. Poulding S, Feldt R (2015) Heuristic model checking using a monte-carlo tree search algorithm. In: Proceedings GECCO. ACM, pp 1359–1366
71. ProB Java API Source Code. https://github.com/hhu-stups/prob2_kernel. Accessed 11 Mar 2020
72. ProB Java API Example Source Code. https://github.com/hhu-stups/executable_spec_example. Accessed 11 Mar 2020
73. ProB Maven Artifacts. https://search.maven.org/artifact/de.hhu.stups/de.prob2.kernel. Accessed 11 Mar 2020
74. Rehm J, Cansell D (2007) Proved development of the real-time properties of the IEEE 1394 root contention protocol with the event B method. In: Proceedings ISoLA, Revue des Nouvelles Technologies de l'Information, vol RNTI-SM-1. Cépaduès-Éditions, pp 179–190
75. Rivera V, Cataño N, Wahls T, Rueda C (2017) Code generation for event-B. STTT 19(1):31–52

76. Rodriguez MTL (2013) Invariant discovery and refinement plans for formal modelling in Event-B. Ph.D. thesis, Heriot-Watt University, UK
77. Savary A, Lanet JL, Frappier M, Razafindralambo T, Dolhen J (2012) VTG—vulnerability test generator, a plug-in for rodin. In: Workshop deploy 2012. Fontainebleau, France
78. Schmidt J, Krings S, Leuschel M (2018) Repair and generation of formal models using synthesis. In: Proceedings IFM, LNCS, vol 11023. Springer, pp 346–366
79. Schneider D (2017) Constraint modelling and data validation using formal specification languages. Ph.D. thesis, Heinrich-Heine-Universität Düsseldorf
80. Schneider D, Leuschel M, Witt T (2018) Model-based problem solving for university timetable validation and improvement. In: Formal aspects of computing, pp 545–569
81. Short M, Pont MJ (2008) Assessment of high-integrity embedded automotive control systems using hardware in the loop simulation. J Syst Softw 81(7):1163–1183
82. Smaus JG, Hoffmann J (2009) Relaxation refinement: a new method to generate heuristic functions. In: Postproceedings MOCHART 2008, LNAI, vol 5348. Springer, pp 147–165
83. Spivey JM, Abrial J (1992) The Z notation. Prentice Hall Hemel Hempstead, Englewood Cliffs
84. The ProB Logic Calculator. https://github.com/hhu-stups/prob-logic-calculator. Accessed 10 July 2020
85. Thule C, Lausdahl K, Gomes C, Meisl G, Larsen PG (2019) Maestro: the INTO-CPS co-simulation framework. Simul Model Pract Theory 92:45–61
86. Thule C, Lausdahl K, Larsen PG (2018) Overture FMU: export VDM-RT models as tool-wrapper FMUs. In: Proceedings of the 16th overture workshop, School of Computing Science Technical Report Series, vol 1524. School of Computing Science, University of Newcastle upon Tyne, pp 23–38
87. Thule C, Nilsson R (2016) Considering abstraction levels on a case study. In: The 14th overture workshop: towards analytical tool chains, vol 4. The Electronics and Computer Engineering. Aarhus University, Department of Engineering, pp 16–31
88. Tran-Jørgensen PW, Larsen PG, Leavens GT (2018) Automated translation of VDM to JML-annotated Java. STTT 20(2):211–235
89. Vu F, Hansen D, Körner P, Leuschel M (2019) A multi-target code generator for high-level B. In: Proceedings iFM, LNCS, vol 11918. Springer, pp 456–473
90. Wahls T, Leavens GT, Baker AL (2000) Executing formal specifications with concurrent constraint programming. Autom Softw Eng 7(4):315–343
91. Watson N, Reeves S, Masci P (2018) Integrating user design and formal models within PVSio-Web. In: Proceedings F-IDE, vol 284. Open Publishing Association, pp 95–104
92. Werth M, Leuschel M (2020) VisB: a lightweight tool to visualize formal models with SVG graphics. In: Proceedings ABZ 2020, LNCS, vol 12071. Springer, pp 260–265
93. Yang F, Jacquot J, Souquières J (2013) JeB: safe simulation of event-B models in javascript. Proc APSEC 2013:571–576
94. Zenzaro S, Gervasi V, Soldani J (2014) WebASM: an abstract state machine execution environment for the web. In: Proceedings ABZ, LNCS, vol 8477. Springer, pp 216–221