Embedding High-Level Formal Specifications into Applications

Philipp Körner, Jens Bendisposto, Jannik Dunkelau,
Sebastian Krings, Michael Leuschel

The final authenticated version is available online at
https://doi.org/10.1007/978-3-030-30942-8_31.

Embedding High-Level Formal Specifications into Applications

Philipp Körner ✉[0000−0001−7256−9560], Jens Bendisposto[0000−0001−5914−1092], Jannik Dunkelau[0000−0003−0819−5554], Sebastian Krings[0000−0001−6712−9798], Michael Leuschel[0000−0002−4595−1518]

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf, Germany
p.koerner@hhu.de
{jens.bendisposto,jannik.dunkelau,sebastian.krings,michael.leuschel}@hhu.de

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 the enabling technology ProB 2.0, a Java API to the ProB animator and model checker. We describe several case studies that use ProB 2.0 to interact with a formal specification at runtime.

1 Introduction

When designing safety-critical software, the use of formal methods is highly recommended [13] 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 required to translate the model into a traditional programming language. Low-level formalisms are usually close enough that code can be generated easily. When using high-level formalisms 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: in the following, 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 ProB 2.0, a Java API for interaction with ProB in Section 2. Following, we evaluate our approach by implementing and discussing several new case studies based on ProB 2.0 in Section 3, summarising its use in existing industrial applications and insights gained from implementation work. Finally, we argue for using formal models as runtime artefacts and discuss similar approaches in Section 4.

1.1 B, Event-B and ProB

Both B [3] and its successor, Event-B [2], 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 [17,18], using sets for data modelling. Further, they make use of generalised substitution for state modifications, and of refinement calculus [4,5] to describe models at different levels of abstraction [9].

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].

In the following, we describe several projects that make use of ProB [33], an animator and model checker for both B and Event-B. Its core is developed mainly in SICStus Prolog [11], with some parts being implemented in C and Java, and makes use of co-routines and SICStus' CLP(FD) library [10]. Besides B, ProB offers support for several other formalisms as well, including TLA$^+$ [30] (via translation to B [22]), Z [42,38], CSP [25,8] and more. Hence, the approach discussed in this article is immediately applicable to languages other than B and Event-B.

2 ProB 2.0

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 use anything but the default animation and model checking capabilities. Thus, a main design goal of ProB 2.0 was to offer access to the API of ProB via a scripting language that allows easy embedding of domain specific languages (DSLs). For this, we picked and embraced Groovy, a dynamic programming language running on the JVM, which is (almost) a superset of Java. ProB 2.0 is available on GitHub[2].

A general overview of ProB 2.0 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.

[1] Cf. `https://www3.hhu.de/stups/prob/index.php/Summary_of_B_Syntax`
[2] `https://github.com/bendisposto/prob2`

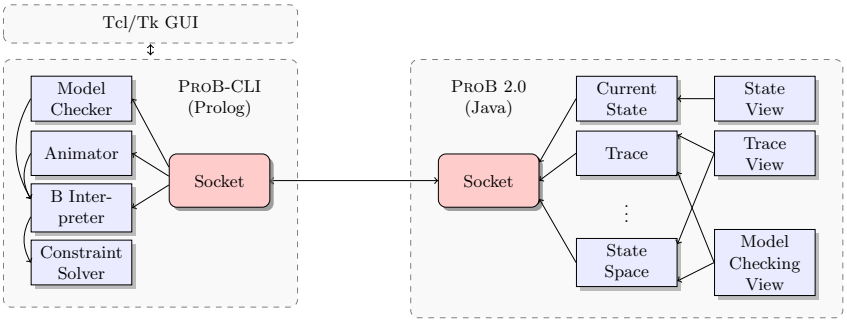

Fig. 1: Overview of the ProB Ecosystem

Each command on the whitelist has a corresponding implementation in ProB 2.0. This offers an API that is fairly low-level and intended for ProB and ProB 2.0 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 2.0 commands.

The Model is an in-memory version of the loaded B machine. ProB 2.0 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 [14].

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 ProB 2.0. 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 ProB 2.0, 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 ProB 2.0. It allows coupling of multiple traces, even on different machines. 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.

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. Additionally, 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.

In all four examples we use the state that is translated into Java data structures in order to provide an (interactive) visualisation.

3.1 Real-Time Animation: Pac-Man

Our first example application is based on a formal model of Pac-Man[3].

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 a plugin in ProB 2.0 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:

[3] Available at: `https://github.com/pkoerner/EventBPacman-Plugin`

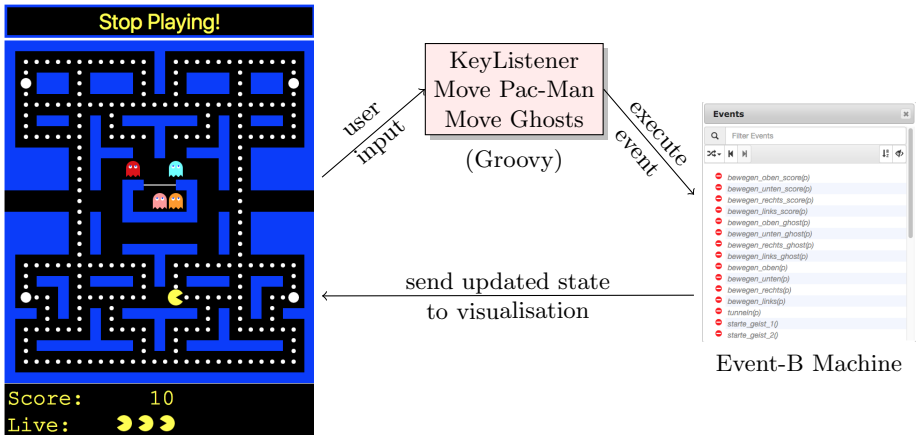

Fig. 2: Architecture of a Pac-Man Game Based on a Formal Model

- 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 for each ghost and are extended by the next movement operation.
- 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[4]. 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.

[4] 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.

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 reasonably 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.

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. Simultaneously, the model has to be as deterministic as possible. As at least the ghosts are to be moved automatically, the computer controlled aspects of Pac-Man have to be modelled deterministically in order to avoid ambiguity and to avoid having to implement how to decide between different options.

3.2 Predicting the Future: Chess

In the chess example[5], 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.

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 an AI is hard to specify but somewhat easy to implement. Thus, the AI was written in Java using ProB 2.0: we implemented a Minimax algorithm with alpha-beta pruning [27]. The calculated 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. Fig. 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 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.

[5] Available at: `https://github.com/pkoerner/b-chess-example`

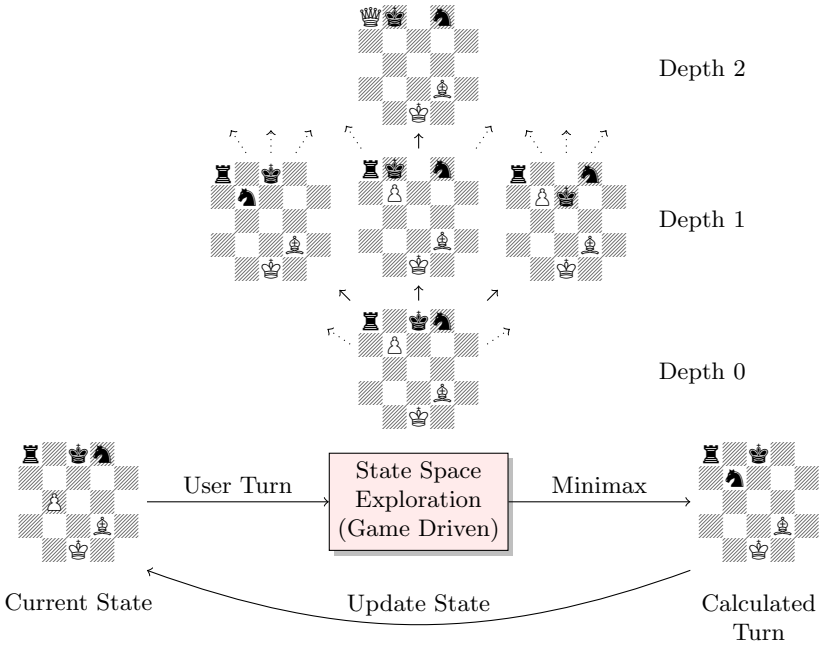

Fig. 3: Architecture of Chess Based on a Formal Model

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 [6], 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. 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.

Main Contribution: Game-driven Model Exploration In this case study, we replaced the common exploration strategies of ProB (breadth-first, depth-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 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 [32] 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 2.0 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.

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. 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 as a Constraint Solver: PlüS

PlüS[6] is an application for planning university timetables [40,41]. 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 timeframes 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 timeslots 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 required to be completed beforehand. 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 timeslots and re-calculate. This is done via drag-and-drop and, again, triggers the corresponding operation in the B machine.

[6] Available at: `https://plues.github.io/en/index/`

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.

Main Contribution: Improving the B Eco-System PlüS was one of the earlier projects that used ProB 2.0 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 sublanguage. 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.

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.

3.4 Real Time Animation: ETCS Hybrid Level 3 Concept

We also used ProB 2.0 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.

Fig. 4: Screenshot from a video of Deutsche Bahn `https://www.youtube.com/watch?v=FjKnugbmrP4` showing a formal Event-B model in action

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 Figure 4, where in the lower center one can see the ProB 2.0 visualisation of the formal model. The visualisation shows that two trains occupy the same occupied track section, but occupy disjoint virtual subsections. The ETCS hybrid level 3 principles were independently used as a case study for ABZ 2018 [23]. Due to page constraints, we can only give a high-level overview here. More details can be found in [23].

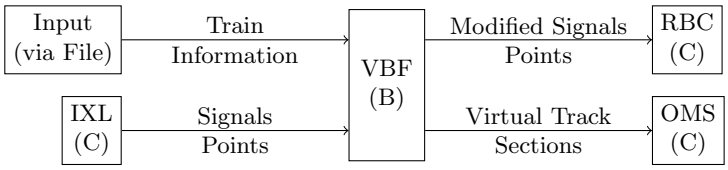

Fig. 5: Architecture of the HL3 Prototype

When considering railroad tracks, they are usually divided in subsections. On the boundaries, usually there are sensors like axle counters in order to determine whether a train is occupying the corresponding subsection. The main idea is that by using a Virtual Block Function (VBF), the capacity of existing railroad networks can be increased. The VBF subdivides the tracks into virtual subsections without having to install further sensors or other hardware onto the tracks. As main part of the demonstration, the VBF was written as a B model, managing the status of said virtual subsections.

For the overall demonstrator, the VBF was interfaced with other 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 feed information into the model via ProB 2.0 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 2.0 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.

Though we cannot disclose the model and the application itself, 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.

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.

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. I.e., we automatically got record and replay capabilities of debuggers as in [35].

4 Discussion and Related Work

When thinking of executing formal specifications, one usually has animation or code generation (cf. [20,44]) in mind. Yet, we will first discuss whether the approach itself makes sense: in the past, there has been thorough discussion [24,19,21] whether specifications should be executable in the first place.

4.1 Soundness of Approach

Some argue [24] that formal specifications should not be executable for several reasons:

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

We deliberately go against this judgement and use a formal specification language, B, that has powerful tooling such as animators, model checkers, PO generators, etc.

This does not mean that we disagree with these arguments; firstly, 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, it is very hard and cumbersome to discharge any proof obligation. On the other hand, models written to be proven usually are not executable 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.

Secondly, 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, we embrace specifications an animator can handle and execute.

Thirdly, over-specification does not seem to be an issue for our use case. An example from [24] 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 in a declarative manner.

In the typical workflow of the B-Method, a concrete implementation happens during refinement. Thus, the writer of the specification is usually *able to choose* the level of abstraction herself.

The last argument concerning performance is carefully reviewed for each of our case studies individually in Section 3 and overall in Section 5.

Being able to execute specifications also gives rise to techniques such as animation and model checking. Both of them have proven to be vital during creation and debugging of a formal specification, as errors can be caught early on.

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

Fig. 6: Sorting Predicate

4.2 Animation

As mentioned above, 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 in an automated manner. Executing a longer trace by hand and verifying each encountered state is correct is very cumbersome and might be aided by state visualisations.

In contrast to the approach we presented earlier, the user interacts with the formal model directly. This also means that all events have been chosen by hand, even ones that should be picked by the environment. In our examples, that includes movement of the ghosts in Pac-Man, moving the chess pieces of the enemy and providing the input of signals, points, etc.

Another approach [37] executes several example runs on probabilistic models. Yet, it is not possible to let an environment interact with the model itself. On the other hand, this approach encourages non-deterministic models.

4.3 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 [29,28] is a tool for state visualisation based on web technologies. It also builds upon the ProB 2.0 API and allows simple interaction with the model. The chess example from Section 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 2.0 and uses Groovy, SVG, JavaScript and HTML5, where each component of the stack may go wrong, rendering development very cumbersome.

State visualisation is not unique to the B formalisms: e.g., another tool that allows visualisations based on web technologies is WebASM [45], which works on top of CoreASM [16]. CoreASM is a tool that can be used to execute abstract state machines (ASM).

4.4 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 [1,15], 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 [15]. 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 [7] that generates LLVM code. A notable toolset Event-B is EB2ALL [34], which allows code generation to several languages including C and Java.

Currently, we explore which level of abstraction is required to render it feasible to generate code from higher-level specifications [43]. Supporting more constructs from B that are not included in B0 might make an approach using code generation more feasible. This work is aligned with EventB2Java [12,39] that also translates higher-level constructs.

4.5 Other Approaches

Another formal specification language is part of the Vienna Development Method (VDM) [26]. A well-known tool for VDM is Overture [31], which implements an interpreter in Java. In [36], 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 Section 3.4.

5 Conclusions

In this paper, we presented ProB 2.0, which offers a Java API to the ProB model checker. ProB 2.0 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 four 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 Section 2 or in [36], 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 a high-level specification language, such as B, can provide. Integration with existing code, that may be written in other programming languages or running on different machines, is very powerful. In particular, when re-iterating on the formal model, changes can immediately be re-evaluated by a test scenario in the context of an entire application. Otherwise, an implementation has to be changed as well, which allows introduction of new 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 [24], 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 eco-system associated with formal methods, from proof to animation and model checking.

Acknowledgement. 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 2.0, the Slot Tool and the ETCS Hybrid Level 3 case study.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 1st edition, 2010.
3. J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen. The B-method. In *VDM '91 Formal Software Development Methods*, volume 552 of *LNCS*. Springer, 1991.
4. R. Back. On correct refinement of programs. *Journal of Computer and System Sciences*, 23(1):49–68, 1981.

5. R.-J. Back and J. Wright. *Refinement calculus: a systematic introduction*. Springer Science & Business Media, 2012.
6. A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11):117–148, 2003.
7. R. Bonichon, D. Déharbe, T. Lecomte, and V. Medeiros. LLVM-Based Code Generation for B. In *Formal Methods: Foundations and Applications*, volume 8941 of *LNCS*, pages 1–16. Springer, 2014.
8. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *International Symposium on Formal Methods*, volume 3582 of *LNCS*, pages 221–236. Springer, 2005.
9. D. Cansell and D. Méry. Foundations of the B method. *Computing and informatics*, 22(3-4):221–256, 2012.
10. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
11. M. Carlsson, J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. *SICStus Prolog user's manual*, volume 3. Swedish Institute of Computer Science Kista, Sweden, 1988.
12. N. Cataño and V. Rivera. EventB2Java: A Code Generator for Event-B. In *NASA Formal Methods*, volume 9690 of *LNCS*, pages 166–171. Springer, 2016.
13. CENELEC. Railway Applications – Communication, signalling and processing systems – Software for railway control and protection systems. Technical Report EN50128, European Standard, 2011.
14. J. Clark, J. Bendisposto, S. Hallerstede, D. Hansen, and M. Leuschel. Generating Event-B Specifications from Algorithm Descriptions. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 9675 of *LNCS*, pages 183–197. Springer, 2016.
15. ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2016. Available at `http://www.atelierb.eu/`.
16. R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1-2):71–103, 2007.
17. A. Fraenkel. Zu den Grundlagen der Cantor-Zermeloschen Mengenlehre. *Mathematische Annalen*, 86(3):230–237, 1922.
18. A. A. Fraenkel, Y. Bar-Hillel, and A. Levy. *Foundations of set theory*, volume 67. Elsevier, 1973.
19. N. E. Fuchs. Specifications are (preferably) executable. *Software engineering journal*, 7(5):323–334, 1992.
20. C. Ghezzi and R. A. Kennerer. Executing Formal Specifications: The ASTRAL to TRIO Translation Approach. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 112–122. ACM, 1991.
21. A. Gravell and P. Henderson. Executing formal specifications need not be harmful. *Software engineering journal*, 11(2):104–110, 1996.
22. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Integrated Formal Methods*, volume 7321 of *LNCS*, pages 24–38. Springer, 2012.
23. D. Hansen, M. Leuschel, D. Schneider, S. Krings, P. Körner, T. Naulin, N. Nayeri, and F. Skowron. Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 10817 of *LNCS*, pages 292–306. Springer, 2018.
24. I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–339, 1989.

25. C. A. R. Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.
26. C. B. Jones. *Systematic software development using VDM*, volume 2. Citeseer, 1990.
27. D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.
28. L. Ladenberger. *Rapid creation of interactive formal prototypes for validating safety-critical systems*. PhD thesis, 2017.
29. L. Ladenberger and M. Leuschel. BMotionWeb: A Tool for Rapid Creation of Formal Prototypes. In *Software Engineering and Formal Methods*, volume 9763 of *LNCS*, pages 403–417. Springer, 2016.
30. L. Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.
31. P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl, and M. Verhoef. The overture initiative integrating tools for VDM. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–6, 2010.
32. M. Leuschel and J. Bendisposto. Directed Model Checking for B: An Evaluation and New Techniques. In *Formal Methods: Foundations and Applications*, volume 6527 of *LNCS*, pages 1–16. Springer, 2011.
33. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, volume 2805 of *LNCS*. Springer, 2003.
34. D. Méry and N. K. Singh. Automatic code generation from event-B models. In *Proceedings SoICT*, pages 179–188. ACM, 2011.
35. S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ACM SIGARCH Computer Architecture News*, volume 33, pages 284–295. IEEE Computer Society, 2005.
36. C. B. Nielsen, K. Lausdahl, and P. G. Larsen. Combining VDM with executable code. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 7316 of *LNCS*, pages 266–279. Springer, 2012.
37. T. Nummenmaa. *Executable formal specifications in game development: Design, validation and evolution*. PhD thesis, 2013.
38. D. Plagge and M. Leuschel. Validating Z specifications using the ProB animator and model checker. In *Integrated Formal Methods*, volume 4591 of *LNCS*, pages 480–500. Springer, 2007.
39. V. Rivera, N. Cataño, T. Wahls, and C. Rueda. Code generation for Event-B. *STTT*, 19(1):31–52, 2017.
40. D. Schneider. *Constraint Modelling and Data Validation Using Formal Specification Languages*. PhD thesis, Heinrich-Heine-Universität Düsseldorf, 2017.
41. D. Schneider, M. Leuschel, and T. Witt. Model-based problem solving for university timetable validation and improvement. *Formal Aspects of Computing*, pages 545–569, 2018.
42. J. M. Spivey and J. Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.
43. F. Vu. A High-Level Code Generator for Safety Critical B Models. Bachelor's thesis, Heinrich Heine Universität Düsseldorf, August 2018.
44. T. Wahls, G. T. Leavens, and A. L. Baker. Executing formal specifications with concurrent constraint programming. *Automated Software Engineering*, 7(4):315–343, 2000.
45. S. Zenzaro, V. Gervasi, and J. Soldani. WebASM: an abstract state machine execution environment for the web. In *Abstract State Machines, Alloy, B, VDM, and Z*, volume 8477 of *LNCS*, pages 216–221. Springer, 2014.