Towards Practical Partial Order Reduction for High-Level Formalisms

Philipp Körner, Michael Leuschel

Towards Practical Partial Order Reduction for High-Level Formalisms

Philipp Körner, Michael Leuschel

Heinrich Heine University, Universitätsstraße 1, 40225 Düsseldorf, Germany
{p.koerner,leuschel}@hhu.de

Abstract. Partial order reduction (POR) has considerable potential to reduce the state space during model checking by exploiting independence between transitions. This potential remains, however, largely unfulfilled for high-level formalisms such as B or TLA$^+$. In this article, we report on our experiments regarding POR: We empirically assess that our current implementation of POR in ProB does not have any impact for a vast majority of B machines. We then analyse why POR fails to achieve reductions and identify minimal examples without reduction that make use of high-level constructs in B, and provide several new ideas to make POR pay off for more complex formal models. A proof-of-concept implementation then yields two orders of magnitude reduction in the state space for a particularly challenging case study, a railway interlocking model that escaped our POR techniques thus far.

Keywords: B-Method · Partial Order Reduction· Model Checking · Analysis.

1 Introduction

Partial order reduction (POR) [18,32,38] is a technique to tackle the state space explosion problem in model checking [12]: Instead of executing all interleavings of independent behaviour, only one is explored in the best case. In *low-level formalisms*, such as Petri nets or Promela, and in process algebras like CSP or mCRL2, POR is known to reduce the state space by several orders of magnitudes [7,17,19,25].

In contrast, the application of POR to *high-level formalisms* like TLA$^+$ [26] or B [1,2] has been disappointing thus far. Attempts at using POR for TLA$^+$ using TLC [39] were not successful and abandoned[1]. POR has also been implemented for B using the ample set approach within ProB [13,14,15]. While considerable reduction can be obtained for some specifications, the technique does not seem beneficial for real-life examples. Another attempt of using POR for B was made using LTSmin together with ProB [6,25]. It uses ProB to solve predicates and calculate the next states while POR is provided by LTSmin. LTSmin's

[1] Private communication from Stephan Merz to Michael Leuschel at Schloß Dagstuhl; see also the presentation by Kuppe [21].

```
1  MACHINE NoReduction
2  VARIABLES xx, locked
3  INVARIANT xx ∈ POW(1..2) ∧ locked ∈ 𝔹
4  INITIALISATION xx := ∅ || locked := ⊥
5  OPERATIONS
6  add(yy) = SELECT locked = ⊥ ∧ yy ∈ 1..2 ∧ yy ∉ xx
7            THEN xx := xx ∪ {yy} END;
8  lock    = SELECT locked = ⊥ THEN locked := ⊤ END;
9  unlock  = SELECT locked = ⊤ THEN locked := ⊥ END
10 END
```

Listing 1: Adding a Value Into a Set — No Reduction

approach to POR is based on the stubborn set theory [38] and works well for *low-level* formalisms. Compared to ProB's approach in [13,14,15], the approach of LTSmin is more fine-grained (wrt. guards), yet rarely achieves (mostly slightly) better reduction for B models[2]. Overall, POR rarely seems worth the effort for practical B models.

This article re-visits the implementation of POR in ProB: first, we evaluate its effectiveness in Section 3. The main insight we gained is that static analysis of a model (before model checking) often does not determine a precise enough independence relation. The techniques described in the rest of the paper focus on POR for deadlock checking (as effectiveness is already low and LTL model checking requires even more constraints): Many B models contain operations drawing a parameter from a known finite set; such operations are treated as a unit and, thus, independence between certain instances cannot be captured. We propose to *unroll* such operations by replacing them with a new operation for each parameter (Section 4). Additionally, operations that access a shared set variable usually only interact with a small subset of its elements. We discuss benefits and drawbacks of a constraint-based analysis as well as encoding sets to SAT variables before applying a syntactical analysis (Section 5).

As an example, the model in Listing 1 can (automatically) be re-written to an equivalent model depicted in Listing 2 by *unrolling* the `add` operation and encoding the set `xx` as booleans. The former model yields no state space reduction using ProB's POR, whereas the latter one does. Though some specifications may require additional re-writes or more involved analysis techniques, the combination of these two techniques allows state space reduction by POR on large, real-world models. In Section 6, we share key insights based on a grand challenge we set ourselves, a large model with many real-world features whose state space should be significantly reduced using POR, yet escaped our approach so far. With the techniques above, the expected reduction occurs.

[2] Already the results in Section 4.3 and Table 3 of [24] for POR were unsatisfying. Other techniques of LTSmin were very effective, however.

```
MACHINE HasReduction
VARIABLES xx_1, xx_2, locked
INVARIANT xx_1 ∈ 𝔹 ∧ xx_2 ∈ 𝔹 ∧ locked ∈ 𝔹
INITIALISATION xx_1 ≔ ⊥ ‖ xx_2 ≔ ⊥ ‖ locked ≔ ⊥
OPERATIONS
add_1  = SELECT locked = ⊥ ∧ xx_1 = ⊥ THEN xx_1 ≔ ⊤ END;
add_2  = SELECT locked = ⊥ ∧ xx_2 = ⊥ THEN xx_2 ≔ ⊤ END;
lock   = SELECT locked = ⊥ THEN locked ≔ ⊤ END;
unlock = SELECT locked = ⊤ THEN locked ≔ ⊥ END
END
```

Listing 2: Unrolled and SAT Encoded Version of Listing 1 — POR is Successful

2 Background

The B-Method [1] and its successor Event-B [2] are methodologies that rely on a correct-by-construction approach, i.e., an abstract specification is proven correct and is iteratively refined as more details are added. Proofs accompany all refinement steps, linking each iteration to the ones before.

Both B and Event-B have seen particular use in the railway industry [9]. While the former focuses on software development, the latter is designed for modelling systems. Event-B is most commonly used via the Rodin toolset [3], and exported proof information can be used for model checking [5]. B and Event-B are very expressive, encompassing first-order logic with (higher-order) sets, sequences, functions, relations and records. Both formalisms are state-based with (possibly non-deterministic) initial assignments of constants and state variables, and guarded transitions (named operations in B and events in Event-B)[3] yielding successor states. A state of a B model is composed of values for all the constants and variables of the model.

While we study both B and Event-B models, we will use the term operation to denote both B operations and Event-B events. Small examples of a B specification are given in the motivating example in Listings 1 and 2. B machines might include additional clauses such as the `CONSTANTS` clause (that declares identifiers of constants similar to the `VARIABLES` clause), the `PROPERTIES` clause (constraining the constants) or the `SET` clause (that contains, e.g., enumerated sets). While the following concepts of operation and operation instance are related, it is important to distinguish between them:

Notation. *An* operation *is the name of a guarded substitution (aka statement) that may be parameterised. E.g.,* `add` *or* `lock` *in Listing 1 are operations. The guarded substitution is also called the* body *of the operation.*
An operation along with values for all its parameters is called an operation instance. *E.g.,* `add(1)` *is an operation instance. Another one is* `add(2)`.

An operation instance is thus a transition label.

[3] Or actions in TLA+.

ProB [28,29] is an animator, model checker and constraint solver for the B language. It is written in SICStus Prolog [10] and its constraint-solving backend makes use of coroutines and the CLP(FD) library [11]. Alternative backends are available via translations to SAT and SMT: the work of Plagge and Leuschel [34] uses the Kodkod [37] library to translate B to SAT, while the works of Krings, Schmidt and Leuschel [20,35] translate B to SMT for using Z3 [30] as a solver.

Partial Order Reduction (POR) [4,32,33] is a model checking technique that only explores a subset of the state space. POR is considered to be appealing because, for n independent operation instances, one has to explore (in the best case) only a single ordering rather than $n!$ many. Thus, exponential reductions are possible in concurrent systems that synchronise on few events. While the underlying idea seems simple, the conditions to ensure correctness are intricate[4].

POR exploits *independent* operation instances: Two operation instances are independent, if they can be performed in any order without changing the resulting state. This is visualised in Fig. 1: If α and β are independent and simultaneously enabled in the original state space, this implies that β can be executed after α and vice-versa, and the resulting states are identical. In short, this is the case if the operation instances commute and do not disable each other.

Below, we will give a more formal definition. Note, as is usual when presenting POR, we assume that operation instances are deterministic, i.e., given an operation instance α and a state s there is at most one successor state s' such that $s \xrightarrow{\alpha} s'$.[5]

Notation (Enabling Predicate)**.** *For an operation e, we define en_e to be its enabling predicate (its guard) that is evaluated over a state s.*

Definition 1 (Independence). *Two operation instances α and β are independent, if the following constraint holds. Otherwise, they are dependent.*
$\forall s, s_1, s_2 : en_\alpha(s) \wedge en_\beta(s) \wedge s \xrightarrow{\alpha} s_1 \wedge s \xrightarrow{\beta} s_2 \implies \exists s' : en_\beta(s_1) \wedge en_\alpha(s_2) \wedge s_1 \xrightarrow{\beta} s' \wedge s_2 \xrightarrow{\alpha} s'$

The operation instance `lock` depends on `add(1)` (and vice versa, as the independence relation is symmetric), because performing `lock` may (and will) disable `add(1)`. The operation instance `add(1)` is independent of `add(2)`.
Usually, one approximates the independence relation during static analysis before model checking based on operations. Two operations are independent if all respective operation instances are independent. As an example, the operations `add` and `unlock` are independent of each other because they write different variables (and the read in the guard of `add` of `unlock` is not conflicting)[6].

[4] For example, an error in a twenty-year-old algorithm was recently discovered [36].

[5] For Event-B it is straightforward to lift all non-determinism into parameters. In Classical B this is more difficult; but the formalisation of independence with non-determinism would make the presentation overly complex and detract from the main points of the article.

[6] More precisely, all operation instances of `add` are independent of `unlock` because they can never be enabled at the same time.

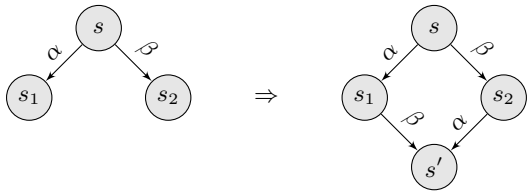


Fig. 1: Visualisation of the Operation Independence Definition

The Ample Set Approach As the POR implementation in ProB relies on the ample set approach[7], we introduce it more formally. For this article, it is not necessary to understand *why* POR works in detail, but only *what information* is required.

By $op(\alpha)$ we denote the operation associated with an operation instance α. We also define the enabled operations in a state s by $enabled(s) = \{op(\alpha) \mid \exists s' : s \xrightarrow{\alpha} s'\}$.

An ample set is a subset of enabled operations in a state (referred to as s in the following formulas) that are considered by model checking. In other words, all operation instances for operations not contained in the ample set are ignored. For example, in Figure 1, we could choose $ample(s) = \{op(\alpha)\}$ and thus ignore β in s. To reach a sound reduction of the state space, one requires the following conditions to hold (taken from [15]):

(A 1) **Emptiness Condition:** $ample(s) = \varnothing \Leftrightarrow enabled(s) = \varnothing$

(A 2) **Dependence Condition:** Along every finite path in the original state space starting at s, an operation dependent on $ample(s)$ cannot appear before some operation $e \in ample(s)$ is executed.

The conditions (A 1) and (A 2) suffice for deadlock checking; LTL model checking (which is used for invariant checking) has additional conditions (stutter and cycle), yet those are out of scope for this paper. In ProB's implementation, two local criteria are used instead of (A 2). They have been proven correct in [14,15]:

(A 2.1) **Direct Dependence Condition:** Any (ignored) operation $e \in enabled(s) \setminus ample(s)$ is independent of all operations in $ample(s)$.

(A 2.2) **Enabling Dependence Condition:** Any (disabled) operation $e \in Events \setminus enabled(s)$ that depends on some operation $f \in ample(s)$ *and* is possibly co-enabled with f may not become enabled by execution of operations $e' \notin ample(s)$.

Two operations are considered to be possibly co-enabled if there exists a state s in which both guards are satisfied. Note that such a state may not be reachable.

Thus, in practice, the independence relation, an enabling relation and a "may be co-enabled" relation between operations are approximated during a static analysis phase (which we will refer to as *POR analysis*).

[7] The implementation in LTSmin uses stubborn sets. There is not much difference concerning our argument as the analysis must extract mostly the same information.

3 Experiments and Results

In order to evaluate the impact of ProB's partial order reduction, we use a collection of B and Event-B specifications [23] and compare the state space sizes with and without applying POR. We consider 1894 B machines with at least two operations in order to have an opportunity for independent events to occur. The set of machines and produced results can be found on GitHub[8].

All machines were model checked for 30 minutes (per configuration) with 2 GB of RAM on a single CPU core of an Intel E5-2697v2 (Ivy Bridge EP) running at 2.70 GHz. A nightly version of ProB 1.11.0 was used (commit `1b6f14bbd533c2459b1ce675eb57ab24fee89caa`).

For **deadlock checking**, we excluded 519 machines that time out with and without POR; 17 machines that time out only with POR; and 25 machines ($\tilde{4}$ % of machines with timeout) only timed out using the vanilla baseline implementation. We assume some reduction occurred for these 25 machines. 3 machines are included due to some other error. Thus, 1330 machines are subject to this analysis. 1121 are deadlock-free, and 209 contain a deadlock.

The original and reduced state space sizes are given in Fig. 2a and Fig. 2b. Data points on the diagonal correspond to cases where no reduction occurs, while data points below the diagonal correspond to a reduction due to POR. In the right figure (Fig. 2b) data points can also be found above the diagonal, meaning that model checking with POR did find the deadlock later than without POR.

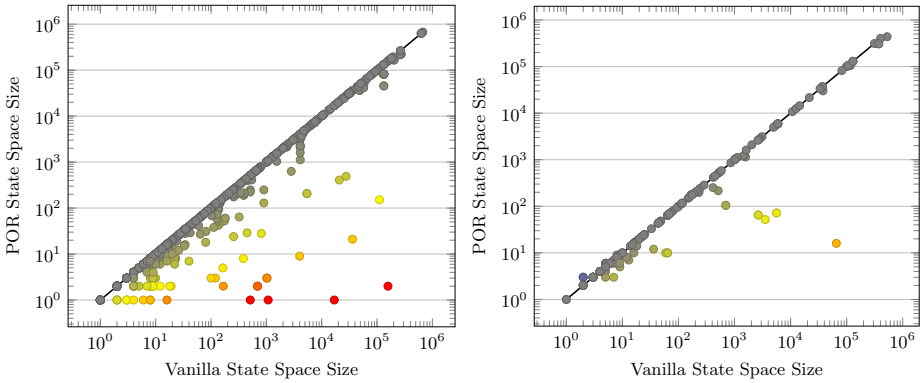
Of the 1121 deadlock-free machines, only 191 (17 %) showed some reduction with POR. On average the reduced state space has 54 % of the original size (i.e., a reduction of 46 %) for these 191 machines. The median is 56 % of the original size. Similar, of 209 machines containing deadlocks, we can observe 36 (17.2 %) with a reduced state space and 9 with a larger one (as discussed earlier).

Thus, even when adding the 25 machines with timeout when not using POR above, we have less than 20 % of models where POR reduces the state space.

For **invariant checking**, we can analyse 1385 machines after excluding 452 where both model checking algorithms time out, 2 machines that only time out with POR and 55 machines that only time out without POR. Again, we assume some reduction for the latter cases (around 11 % of all machines featuring any timeout). Further, we exclude 55 additional machines due to other errors. This leaves 1331 machines to analyse here, of which 1169 machines preserve the invariant.

The (reduced) state space sizes are visualised in Fig. 3a. Of these, we can observe a state space reduction in 37 machines (3.2 %). On average, the reduced state space has 76 % of the original size (i.e., a reduction of 24 %) for these 37 machines. The median is 86 % of the original size. Unexpectedly, a single outlier lies above the diagonal, i.e., yields a larger state space with POR. This is a machine that acts as a test for ProB's randomisation library, and hence the state space can change with each run. Even when assuming that all 55 machines with timeout produce a reduction, we have a reduction in less than 10 % of cases.

[8] https://github.com/hhu-stups/specifications/tree/por-experiments



(a) Deadlock-Free Models (b) Deadlock-Containing Models

Fig. 2: (Reduced) State Space Sizes for Deadlock Checking

Of 162 machines with invariant violations (Fig. 3b), we observe 30 machines (18.5 %) with a reduced state space and 18 with a larger one.

Threats to Validity Many machines time out and are excluded, though they might exhibit better reduction in reality. However, from our sample, we can also observe the trend that smaller machines exhibit state space reductions more often (cf. Figs. 2a and 3a). Indeed, our findings in Section 4 and Section 5 suggest that constructs to structure larger machines *hinder* POR.

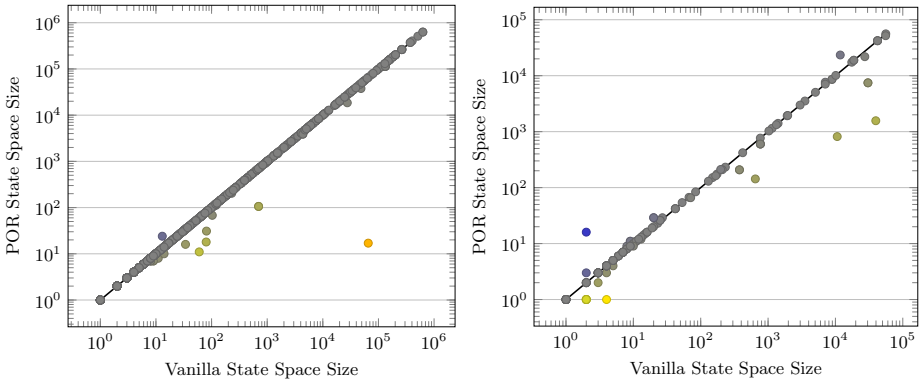
Further, the set of machines may not be representative, as it includes many examples from literature, small machines used for teaching, different versions or instantiations of the same machine, etc., and not larger, confidential machines from industry. From our experience, POR does not work well for these machines. The bias may even be *towards* machines well-suited for POR, as several models meant for testing the POR implementation are included.

4 Idiom 1: Parameterised Operations

ProB's partial order reduction and the POR analysis identifies operations by their name. However, there may be several operation instances, i.e., combinations of a name and concrete parameter values. A trivial example is part of Listing 1.

From a high-level point of view, this machine has three operations where only `add` and `lock` can be enabled simultaneously but are dependent. Thus, the state space cannot be reduced. Yet, the operation instances `add(1)` and `add(2)` satisfy exactly our definition of independence (Fig. 1), as `add(1)` and `add(2)` commute (see Fig. 4)!

In this example, the independence of some operation instances within the same operation is not exploited. In many cases, certain operation instances of *one* operation are independent of certain operation instances of *another* operation. An example is described based on our grand challenge in Section 6.2.



(a) Invariant-Preserving Models (b) Invariant-Violating Models

Fig. 3: (Reduced) State Space Sizes for Invariant Checking

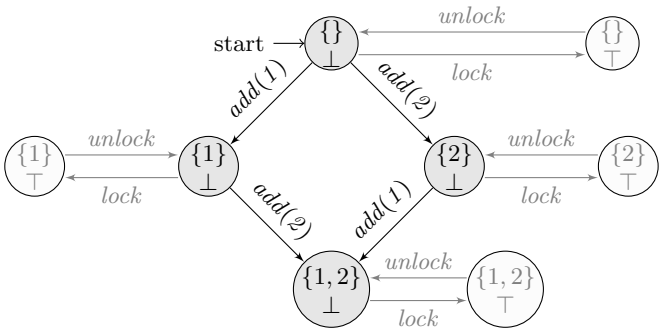


Fig. 4: State space of the machine in Listing 1. Each state consists of the set `xx` (at the top) and the boolean `locked` (at the bottom). The commutativity of the *add* operation instances is highlighted.

4.1 Solution: Unrolling of Operations

The example above has one important property: for the considered operation `add`, we can statically determine a finite set of possible values for the parameters (i.e., either $yy = 1$ or $yy = 2$). In this case we can replace the *operation* with all its *operation instances*, by hardwiring the parameter values. For the example above, this gives rise to two operations `add_1` and `add_2` in Listing 3.

Advantage: Necessary Preprocessing This technique is the bare minimum to locate independence between operations that share at least one variable. Thus, it is the foundation for the techniques below.

Drawback: Infinite Sets This unrolling technique is not always applicable given that parameter choices for all states have to be considered. Indeed, the calculation of all possible parameter values may be expensive and yield a large or infinite number of values (due to an overapproximation by the static analysis).

```
1 OPERATIONS
2 add_1  = SELECT locked = ⊥ ∧ 1 ∉ xx THEN xx ≔ xx ∪ {1} END;
3 add_2  = SELECT locked = ⊥ ∧ 2 ∉ xx THEN xx ≔ xx ∪ {2} END;
4 lock   = SELECT locked = ⊥ THEN locked ≔ ⊤ END;
5 unlock = SELECT locked = ⊤ THEN locked ≔ ⊥ END
```

Listing 3: Unrolled `add` Operation

Drawback: Multiple Evaluations While unrolling an operation may be suitable for POR analysis, it duplicates the majority of sub-expressions. Each operation is considered individually in ProB, and shared sub-expressions have to be re-evaluated which results in a slow-down during model checking.

Below, we assume that all operation instances are unrolled. Thus, there is no difference between the concepts of operation and operation instance and their independence. In case an operation cannot be unrolled, it is retained as-is and syntactic independence can still be determined.

5 Idiom 2: Usage of Compound Values (Sets, etc.)

With the simple unrolling technique above, we have established that the POR analysis could now in principle spot the independence between operation instances. In practice, the POR analysis in ProB will, however, *not* determine the independence if two operations write to the same variable.

For performance reasons, the POR analysis focuses mostly on syntactic aspects in order to yield a fast approximation[9]. It considers the (**a**ction) **r**ead and **w**rite sets of two operations (AR_1, AR_2, R_1, R_2, W_1 and W_2). A variable is contained in the action read set AR of an operation, iff the substitution reads it; in the read set R iff the guard or the substitution reads it; and in the write set W iff the variable is written to. The POR analysis then follows the flowchart depicted in Fig. 5, where only the disabling analysis uses semantic aspects.

If we re-consider the operations in Listing 3, we can observe that both `add_1` and `add_2` write to the same variable `xx`. Obviously, the intersection of the two write sets $W_1 \cap W_2$ is not empty and a syntactic POR analysis yields that the two operations are (race) dependent. Yet, set union is associative and commutative and the operations *should* be classified as independent because $(xx \cup \{1\}) \cup \{2\} = (xx \cup \{2\}) \cup \{1\}$.

5.1 Solution 1: Constraint-Based POR Analysis

Since the original syntactic approach depicted in Fig. 5 does not suffice, we added a new constraint-based semantic approach. Instead of syntactically classifying a

[9] Which is precise enough for some formalisms (at least using LTSmin's POR), but not for others [25].

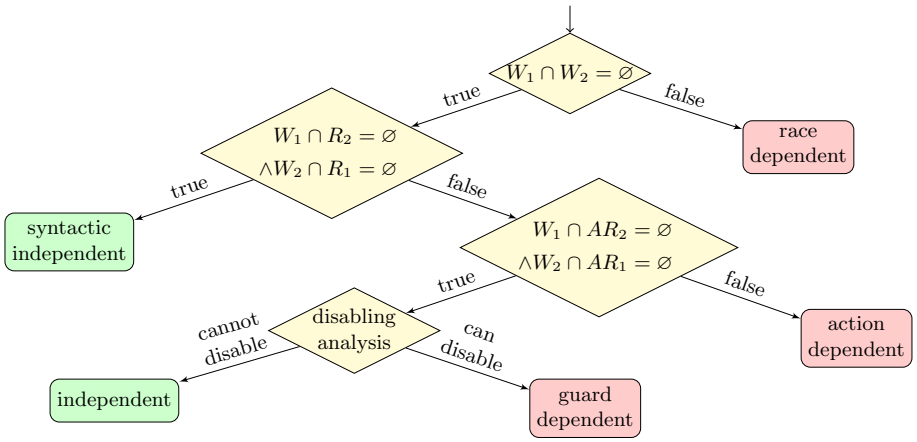


Fig. 5: Syntactically Determining the Independence Relation of Two Operations

pair of operations as race or action dependent (see Fig. 5), we use a constraint solver (ProB, Kodkod or Z3) during the POR analysis. Below, we present how we determine operations to be independent by considering non-disabling and commutativity constraints separately (see Def. 1). Further, in order to be able to check (A 2.2) on the fly, we also use constraints to determine which other operations may (not) be enabled by a specific operation. Finally, again for (A 2.2), one also has to determine which operations may be co-enabled. For the overall approach, we use the notion of before-after predicates and enabling predicates:

Notation (Before-After Predicate)**.** *For an operation instance e, we define $BA_e(s, s')$ to be the before-after predicate. It is a conjunction of the guard of operation $op(e)$ and the predicate whose solutions s' form the successor states of s using e.*

As an example, the before-after predicate for the operation `add_1` is[10]:

$$BA_{add_1}(s, s') \equiv \underbrace{locked = \bot \wedge 1 \notin xx}_{en_{add_1}(s)} \wedge \underbrace{xx' = xx \cup \{1\} \wedge locked' = locked}_{\text{substitution of } add_1}$$

Before-after predicates do not exist for all operations, e.g., those containing a WHILE-loop.

Non-Disabling Constraint. Independent operations must not disable each other and commute. The constraint below checks whether operation α can disable the operation β. The conjunct *Info* might contain additional information, such as the values of constants, proven theorems or (parts of) the state invariant. Also

[10] We will directly refer to the state variables by their name; e.g., xx is part of state s, and xx' is a variable of s'.

note that the states s and s' may not be reachable in the state space, and, thus the following computes a (safe) approximation of disabling:

$$\exists s, s'.(\mathit{Info} \wedge en_\beta(s) \wedge BA_\alpha(s, s') \wedge \neg en_\beta(s'))$$

For example, to check whether `add_1` may disable `add_2`, we have to consider the constraint:

$$\begin{aligned}\exists s, s'.(\mathit{Info} \wedge & \underbrace{\mathit{locked} = \bot \wedge 2 \in xx}_{en_{add_2}(s)} \\ & \wedge \underbrace{\mathit{locked} = \bot \wedge 1 \notin xx \wedge xx' = xx \cup \{1\} \wedge \mathit{locked}' = \mathit{locked}}_{BA_{add_1}(s,s')} \\ & \wedge \underbrace{\neg(\mathit{locked}' = \bot \wedge 2 \in xx')}_{\neg en_{add_2}(s')})\end{aligned}$$

As this constraint is a contradiction, we can conclude that `add_1` cannot disable `add_2` (and, analogously, vice versa). This does not suffice for independence, and we have to continue to check the commutativity of the operations (see below). However, `lock` can (and will) disable `add_1` and the operations cannot be independent. The same holds for `lock` and `add_2`.

Commuting Constraint. The next constraint below encodes counter examples to commutativity in Def. 1. egain, if a solution is found, a timeout occurs or unknown is returned by the solver, we conclude that the operations might be non-commuting and thus dependent:

$$\exists s, s_1, s_2, s_3, s_4.(\mathit{Info} \wedge BA_\alpha(s, s_1) \wedge BA_\beta(s, s_2) \wedge BA_\alpha(s_2, s_3) \wedge BA_\beta(s_1, s_4) \wedge s_3 \neq s_4)$$

E.g., to find that `add_1` and `add_2` commute, the following constraint is used:

$$\begin{aligned}\exists s, s_1, s_2, s_3, s_4.(& \underbrace{\mathit{locked} = \bot \wedge 1 \notin xx \wedge xx_1 = xx \cup \{1\} \wedge \mathit{locked}_1 = \mathit{locked}}_{BA_{add_1}(s,s_1)} \\ & \wedge \underbrace{\mathit{locked} = \bot \wedge 2 \notin xx \wedge xx_2 = xx \cup \{2\} \wedge \mathit{locked}_2 = \mathit{locked}}_{BA_{add_2}(s,s_2)} \\ & \wedge \underbrace{\mathit{locked}_2 = \bot \wedge 1 \notin xx_2 \wedge xx_3 = xx_2 \cup \{1\} \wedge \mathit{locked}_3 = \mathit{locked}_2}_{BA_{add_1}(s_2,s_3)} \\ & \wedge \underbrace{\mathit{locked}_1 = \bot \wedge 2 \notin xx_1 \wedge xx_4 = xx_1 \cup \{2\} \wedge \mathit{locked}_4 = \mathit{locked}_1}_{BA_{add_2}(s_1,s_4)} \\ & \wedge \underbrace{\neg(xx_3 = xx_4 \wedge \mathit{locked}_3 = \mathit{locked}_4)}_{s_3 \neq s_4})\end{aligned}$$

Due to the associativity and commutativity of the set union, the two operations will commute. Further, as they do not disable each other, the constraint can be

found to be unsatisfiable. Hence, we know for certain that for all states Def. 1 holds and the operations are independent of each other.
Non-Enabling Constraint. For condition (A 2.2), we also have to know which operations can *enable* each other. In order to determine whether operation α can enable β, we need a constraint similar to the non-disabling constraint:

$$\exists s, s'.(\mathit{Info} \wedge \neg en_\beta(s) \wedge BA_\alpha(s, s') \wedge en_\beta(s'))$$

As an example, `add_1` cannot enable `add_2` and vice versa. However, both these operations can be enabled by `unlock`.
Co-Enabledness Constraint. Again, for condition (A 2.2), we need to know which operations are potentially co-enabled. The constraint below is true if the operations α and β are co-enabled in some state:

$$\exists s.(\mathit{Info} \wedge en_\alpha(s) \wedge en_\beta(s))$$

For example, `add_1` and `add_2` are both enabled in the initial state. However, `lock` and `unlock` are never co-enabled as their guards form a contradiction.
Advantage: Precision Overall, such a constraint-based analysis is very precise and, in an optimal world, would obtain all necessary information for POR.
Drawback: Required Information In practice, (proven) invariants often are important to determine independence (i.e., they should be part of the *Info* predicate above). E.g., if $x > 0 \Rightarrow x = y$ is known, we can infer that the guards $x > 0$ and $y \leq 0$ are mutually exclusive. However, adding conjuncts to the *Info* predicate can also make a constraint solver time out. We were not able to find a heuristic that selects additional information for the solver and consistently succeeds for more complex models.
Drawback: Analysis Overhead For many constraints the solvers time out, which vastly increases the POR analysis time. We found that for many models, such an analysis surpasses the actual model checking time for the full state space. The issue is further discussed regarding the interlocking example in Section 6.2.
Drawback: Instability of Solver Integrations ProB's own constraint solver does not perform well in finding unsatisfiability of the commuting constraints. Other integrated solvers on the other hand, i.e., Kodkod and Z3 fit extraordinarily well. However, for some constraints Kodkod and Z3 will occupy all available memory (including swap space), leading to crashes during POR analysis.

5.2 Solution 2: SAT Encoding of Finite Sets

While the constraint-based approach above works well for smaller models, the blow up of analysis time renders it less favourable for larger ones. Thus, we have implemented a prototype[11] that aims to expose syntactic independence by automatically re-writing finite set variables (as well as finite relations) into a series of boolean variables. This is technique often refered to as "bit blasting", or "data refinement" in the context of modelling and refinement. It also is used in

[11] Available at: https://github.com/JanRossbach/fset

Kodkod's translation to SAT, and similar re-writes are required when encoding such a model in lower-level formalisms, such as Promela. In Listing 2, an example encoding is given for the machine in Listing 3.

One can see that the (original) set variable `xx` can contain at most two values that can be determined statically (i.e., 1 and 2). Then, the original set `xx` is replaced by a group of boolean variables, here `xx_1` (that equals `TRUE` iff $1 \in xx$) and `xx_2` (that equals `TRUE` iff $2 \in xx$). Finally, a membership check is a comparison with `TRUE` (or `FALSE` for non-membership, e.g., in the guard of `add_1`), and the set union with a singleton set just sets the according boolean to true (e.g., in the body of `add_1`). Most operators concerning sets, functions and relations can be re-written (though some translations are rather involved [37], and are omitted here).

Advantage: Faster Analysis The POR analysis yields a pretty precise result even if the original, fast syntactical analysis in Fig. 5 is re-used. For example, `add_1` reads and writes only `xx_1` and does not require `xx_2`, and vice versa for `add_2`, resulting in independent operations on a syntactical level. Further, as the behaviour of the machine is not altered, one could also verify that this is a valid refinement in order to ensure correctness.

Drawback: Performance There are several aspects of performance overheads to consider here: first, the translation itself requires some time, especially if all operations are unrolled and if complicated invariants are used. For larger models, our prototype of the translation may take several minutes. Second, the translated model does not perform as well during model checking with ProB, and may be several times slower. Thus, a sensible option would be to use the translated model for POR analysis only and map the results to the original model.

Drawback: Translatable Subset Unfortunately, not all operators in the B language have a straightforward mapping to a SAT encoding. As a fallback, one may re-calculate the original set by combining all boolean values it is spliced into. Yet, in these instances, one loses all syntactic independence again.

6 Case Study & Challenge: Railway Interlocking System

In his book on Event-B [2], Abrial presents a model[12] of a railway interlocking system. The role of an interlocking is to safely operate signals and points within an area of the train network. This means that the interlocking controller has to ensure that trains do not collide and that points are not moved while a train is driving over them.

In this section, we investigate the impact of the POR analysis techniques we presented above with this interlocking system by Abrial [2, Chapter 17] (cf. Listing 4). Although it is an academic model intended for teaching, we chose it because *(i)* it shares several features with real-world models, *(ii)* while SAT-based approaches are able to verify small to medium-sized interlockings [8,31], the verification of larger interlockings is still an active research area and challenge, *(iii)* applying ProB's *POR yields no state space reduction*, *(iv)* it requires

[12] https://github.com/pkoerner/train-por/blob/main/Train_1_beebook_TLC.mch

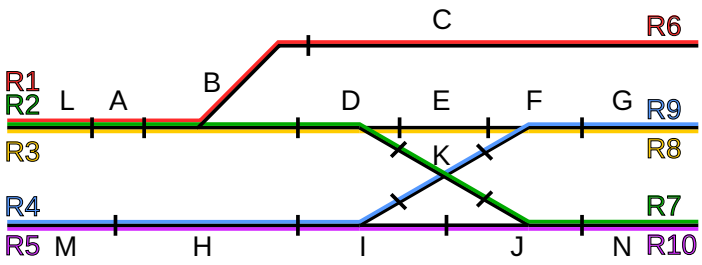


Fig. 6: Example interlocking track layout based on page 524 of [2] with 5 signals, 5 points, one crossing and 14 tracks segments

vast resources for model checking — its state space for the simple topology from Fig. 6 consists of 61 648 077 states and invariant checking with ProB would take about six days (based on estimates [22] — without distributed model checking, the process ran out of memory and crashed), *(v)* one can identify that partial-order reduction is in principle possible because the `route_freeing` operation is independent of all other operations. One can hand-code this insight into the model [27] by forcing this operation (`route_freeing`) to be taken as soon as it is enabled[13], thereby reducing the state space to 672 174 states. Our challenge for the last years has been to identify why our current approach fails and to obtain this two-order of magnitude reduction by (an improved) POR.

6.1 Interlocking Model Overview

The rail network is divided into individual *blocks*; the blocks in Fig. 6 are named A – N. The interlocking allows trains to follow a fixed number of statically determined *routes* through the network. Fig. 6 contains 10 routes, named R1 – R10. For example, route R1 goes through blocks L, A, B, C, while route R2 goes through L, A, B, D, E, F, G and route R6 is the reversed route of R1, going through C, B, A, L (analogously for R7 – R10).

The model also contains the following constants and variables: `fst` and `lst` are functions that map a route to its first and last block, respectively. `nxt` is a function that — given a route — returns a function mapping a block to its successor. `rtbl` is a relation storing the routes for each block. `resbl` (reserved blocks) `resrt` (reserved routes) and `rsrtbl` (blocks reserved for routes) store information about reservations. `OCC` keeps track of blocks that are occupied. `frm` stores which routes are formed on the physical track (`TRK`). `LBT` maps a route to the last block of the train.

Operations are usually called within a certain order: first, a route has to be reserved (`route_reservation`) and the points need to be positioned to match the route (`point_positionning`). Then, these points are locked as the route is formed (`route_formation`). On formed routes, trains may enter and leave blocks

[13] https://github.com/pkoerner/train-por/blob/main/Train_1_beebook_tlc_POR.mch

```
MACHINE Train_1_beebook_TLC
SETS BLOCKS={A,B,C,D,E,F,G,H,I,J,K,L,M,N};
     ROUTES={R1,R2,R3,R4,R5,R6,R7,R8,R9,R10}
CONSTANTS fst, lst, nxt, rtbl
VARIABLES LBT, TRK, frm, OCC, resbl, resrt, rsrtbl
INITIALISATION
resrt ≔ ∅ ‖ resbl ≔ ∅ ‖ rsrtbl ≔ ∅ ‖ OCC ≔ ∅ ‖ TRK ≔ ∅ ‖
frm   ≔ ∅ ‖ LBT ≔ ∅
OPERATIONS
route_reservation(r) =
  SELECT r ∉ resrt ∧ (rtbl⁻¹)[{r}] ∩ resbl = ∅
  THEN resrt ≔ resrt ∪ {r} ‖
       rsrtbl ≔ rsrtbl ∪ (rtbl ▷ {r}) ‖
       resbl ≔ resbl ∪ (rtbl⁻¹)[{r}] END;
route_freeing(r)
  SELECT r ∈ resrt \ ran(rsrtbl)
  THEN resrt ≔ resrt \ {r} ‖ frm ≔ frm \ {r} END;
FRONT_MOVE_1(r) =
  SELECT r ∈ frm ∧ fst(r) ∈ resbl \ OCC ∧ rsrtbl(fst(r)) = r
  THEN OCC ≔ OCC ∪ {fst(r)} ‖ LBT ≔ LBT ∪ {fst(r)} END;
FRONT_MOVE_2(b) =
  SELECT b ∈ OCC ∧ b ∈ dom(TRK) ∧ TRK(b) ∉ OCC
  THEN OCC ≔ OCC ∪ {TRK(b)} END;
BACK_MOVE_1(B) =
  SELECT b ∈ LBT ∧ b ∉ dom(TRK)
  THEN OCC ≔ OCC \ {b} ‖ rsrtbl ≔ {b} ⩤ rsrtbl ‖
       resbl ≔ resbl \ {b} ‖ LBT ≔ LBT \ {b} END;
BACK_MOVE_2(b) =
  SELECT b ∈ LBT ∧ b ∈ dom(TRK) ∧ TRK(b) ∈ OCC
  THEN OCC ≔ OCC \ {b} ‖ rsrtbl ≔ {b} ⩤ rsrtbl ‖
       resbl ≔ resbl \ {b} ‖ LBT ≔ LBT \ {b} ∪ {TRK(b)} END;
point_positionning(r) =
  SELECT r ∈ resrt \ frm
  THEN TRK ≔ ((dom(nxt(r)) ⩤ TRK)
              ⩥ ran(nxt(r))) ∪ nxt(r) END;
route_formation(r) =
  SELECT r ∈ resrt \ frm ∧
        (rsrtbl⁻¹)[{r}] ◁ nxt(r) = (rsrtbl⁻¹)[{r}] ◁ TRK
  THEN frm ≔ frm ∪ {r} END
END
```

Listing 4: Grand Challenge: Abrial's Interlocking System (Excerpt)

in the corresponding order (via the operations `FRONT_MOVE_1`, `FRONT_MOVE_2`, `BACK_MOVE_1` and `BACK_MOVE_2`). Once a train finishes its route, the route is freed again (`route_freeing`).

Since only some routes share blocks, several routes can be reserved, formed and several trains may be on the tracks at the same time. For example, route R1 does not share any block with route R4 or R5. On the other hand, route R3 and R4 both include the blocks F and G.

6.2 Insights

Operation Unrolling As previously mentioned, this is the key technique for the POR analysis that avoids re-writing the POR implementation itself. In our case study, one can unroll all operations, as parameters are either one of the ten routes or fourteen blocks. Then, the unrolled model has 92 operations. If the operations were not unrolled, one could not exploit that some pairs of routes do not overlap (and the corresponding operation instances are, thus, *independent*). One consequence is that the POR analysis cannot infer the independence of, e.g., the route reservation of the disjoint routes R1 and R5. Another consequence is that, e.g., `route_reservation` and `route_formation` are overapproximated as dependent, even though *some* pairs of routes do not overlap (and the corresponding operation instances are, thus, *independent*).

Constrained-Based Analysis The constraint-based approach is able to yield a precise independence analysis. This, however, comes with a cost: if operations *are* dependent on each other, solvers usually time out rather than returning a counterexample or unknown. As many operations do not commute (or may enable or disable each other), this drastically increases POR analysis time. As 4186 (unordered) pairs of operations exist, a full analysis that checks the non-disabling, commutativity (for independence) as well as non-enabling and co-enabledness constraints (for (A 2.2)) takes several hours even on modern hardware due to the amount of timeouts. Finally, even though the obtained information was pretty precise, we did not achieve any reduction with this approach. The POR analysis was not able to determine that a crucial pair of operations cannot be co-enabled (cf. (A 2.2)), and was not precise enough concerning the enabling relation. In particular, for the same parameter route `R`, the operation instance `route_freeing(R)` may disable both `point_positionning(R)` and `route_formation(R)` and, thus, is not independent of them. However, the operations are never enabled at the same time. If this co-enabledness was disproven, the reduction would occur as expected.

SAT Encoding Finally, the SAT encoding of the original model[14] *in combination with* the constraint-based analysis yielded the most precise POR analysis results. In consequence, the technique also allowed the POR algorithm to achieve the same reduction as the hand-written version. Analysis and model checking takes about 30 minutes (1881 seconds) and requires 5048 MB of memory. In comparison, the hand-written version without ProB's POR takes around 7 minutes (397 seconds) and uses 2038 MB of memory. The faster runtime is due to the overhead of the POR as well as the less efficient encoding of the refinement. Reasons for the additional memory usage include a larger refined model and larger states, storage of POR analysis results, etc.

7 Conclusions and Future Work

In this paper, we have identified two idioms in B and Event-B — operation abstraction by parameters and usage of high-level data types — that often hinder

[14] https://github.com/pkoerner/train-por/blob/main/train_auto4.mch

the POR analysis and, henceforth, successful state space reduction. Certainly, there are further patterns that may be uncovered in the future. Thus, our main conclusion is that the usage of high-level constructs prevalent in B are indeed the root cause for our previous unsatisfying experiences with POR and, thus, deeper analysis is required.

We have described three techniques in Sections 4 and 5, (i.e. unrolling of operations, constraint-based POR analysis of operations based on before-after predicates and/or a precise SAT encoding of finite set variables). Individually, each technique is no universal remedy and brings its own drawbacks to the table. In combination, however, one can exploit their individual advantages and, indeed, we were able to match the two order of magnitude state space reduction of the hand-written version for deadlock checking of the interlocking case study.

Related work is dynamic POR [16] which is especially useful for model checking of concurrent software systems, where possible parameter values are drawn from large or infinite sets such as integer values. It avoids static analysis altogether, tracks information dynamically during execution traces and backtracks later if alternative paths that need to be explored are identified. One main benefit is that one does not need to keep the entire state space in memory but only the execution that is currently considered. While this is quite different from our approach, it still requires precise information on the dependence relation and, thus, cannot yield better reduction alone. Yet, evaluating the dependency relation *lazily* — i.e., considering only combinations of operation instances which are actually encountered — can help where our improvements in Sections 4 and 5 currently fail, i.e., when parameters are drawn from infinite sets or when sets are statically unbounded.

The constraint-based analysis still has room for improvement: for one, there might be useful heuristics for similar operation pairs to avoid timeouts. If missing information was made more transparent to the user, one might also assist the POR analysis by providing (proven) theorems. Yet, our implementation of SAT encoding is not mature enough for large-scale benchmarking. In the future, we aim to evaluate our new approach in the large.

Finally, the focus of this study lies on deadlock checking — invariant or LTL model checking may require different or additional techniques. In particular, it is often hard to prove that operations preserve the invariant (which is required for operations to be stutter events, which in turn is required for successful reduction during LTL model checking). Thus, work in this direction might benefit from integrating provers to obtain information about invariants that are guaranteed to be preserved by individual operations.

Acknowledgement. The authors thank the anonymous referees for their feedback, Joshua Schmidt for his patience and relentless work on the Z3 interface and Jan Roßbach for his implementation of the SAT encoding of finite sets. Computational infrastructure and support were provided by the Centre for Information and Media Technology at Heinrich Heine University Düsseldorf.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. Software Tools for Technology Transfer **12**(6), 447–466 (2010)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
5. Bendisposto, J., Leuschel, M.: Proof assisted model checking for B. In: Proceedings ICFEM (International Conference on Formal Engineering Methods). Lecture Notes in Computer Science, vol. 5885, pp. 504–520. Springer (2009)
6. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: Proceedings CAV (International Conference on Computer Aided Verification). Lecture Notes in Computer Science, vol. 6174, pp. 354–359. Springer (2010)
7. Bønneland, F.M., Jensen, P.G., Larsen, K.G., Muñiz, M., Srba, J.: Partial Order Reduction for Reachability Games. In: Proceedings CONCUR (International Conference on Concurrency Theory). LIPIcs, vol. 140, pp. 23:1–23:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019)
8. Borälv, A.: Interlocking Design Automation Using Prover Trident. In: Proceedings FM (International Symposium on Formal Methods). Lecture Notes in Computer Science, vol. 10951, pp. 653–656. Springer (2018)
9. Butler, M., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L.F., Voisin, L.: The First Twenty-Five Years of Industrial Use of the B-Method. In: Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems). Lecture Notes in Computer Science, vol. 12327, pp. 189–209. Springer (2020)
10. Carlsson, M., Mildner, P.: SICStus Prolog—the first 25 years. Theory and Practice of Logic Programming **12**, 35–66 (2012)
11. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Proceedings PLILP (International Symposium on Programming Language Implementation and Logic Programming). Lecture Notes in Computer Science, vol. 1292, pp. 191–206. Springer (1997)
12. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
13. Dobrikov, I., Leuschel, M.: Optimising the ProB Model Checker for B using Partial Order Reduction. In: Proceedings SEFM (International Conference on Software Engineering and Formal Methods). Lecture Notes in Computer Science, vol. 8702, pp. 220–234. Springer (2014)
14. Dobrikov, I., Leuschel, M.: Optimising the ProB model checker for B using partial order reduction. Formal Aspects of Computing **28**(2), 295–323 (2016)
15. Dobrikov, I.M.: Improving Explicit-State Model Checking for B and Event-B. Ph.D. thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2017)
16. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings POPL (Symposium on Principles of Programming Languages). pp. 110–121. ACM (2005)
17. Gibson-Robinson, T., Hansen, H., Roscoe, A.W., Wang, X.: Practical partial order reduction for CSP. In: Proceedings NFM (NASA Formal Methods Symposium). Lecture Notes in Computer Science, vol. 9058, pp. 188–203. Springer (2015)

18. Godefroid, P.: Using Partial Orders to Improve Automatic Verification Methods. In: Proceedings CAV (International Conference on Computer Aided Verification). Lecture Notes in Computer Science, vol. 531, pp. 176–185. Springer (1990)
19. Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997)
20. Krings, S., Leuschel, M.: SMT solvers for validation of B and Event-B models. In: Proceedings IFM (International Conference on Integrated Formal Methods). Lecture Notes in Computer Science, vol. 9681, pp. 361–375. Springer (2016)
21. Kuppe, M.A.: Let TLA+ RiSE. RiSE group all-hands meeting (August 2018)
22. Körner, P., Bendisposto, J.: Distributed Model Checking Using ProB. In: Proceedings NFM (NASA Formal Methods Symposium). Lecture Notes in Computer Science, vol. 10811, pp. 244–260. Springer (2018)
23. Körner, P., Leuschel, M., Dunkelau, J.: Towards a Shared Specification Repository. In: Proceedings ABZ (International Conference on Rigorous State-Based Methods). Lecture Notes in Computer Science, vol. 12071, pp. 266–271. Springer (2020)
24. Körner, P., Meijer, J., Leuschel, M.: State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin. In: Proceedings iFM (International Conference on integrated Formal Methods). Lecture Notes in Computer Science, vol. 11023, pp. 275–295. Springer (2018)
25. Laarman, A., Pater, E., Van De Pol, J., Hansen, H.: Guard-based partial-order reduction. Software Tools for Technology Transfer **18**(4), 427–448 (2016)
26. Lamport, L.: Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley (2002)
27. Leuschel, M., Bendisposto, J., Hansen, D.: Unlocking the Mysteries of a Formal Model of an Interlocking System. In: Proceedings Rodin Workshop 2014 (2014)
28. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Proceedings FME (International Symposium of Formal Methods Europe). Lecture Notes in Computer Science, vol. 2805, pp. 855–874. Springer (2003)
29. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Software Tools for Technology Transfer **10**(2), 185–203 (2008)
30. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings TACAS (International Conference on Tools and Algorithms for the Construction and Analysis of Systems). Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
31. Parillaud, C., Fonteneau, Y., Belmonte, F.: Interlocking Formal Verification at Alstom Signalling. In: Proceedings RSSRail (International Conference on Reliability, Safety, and Security of Railway Systems). Lecture Notes in Computer Science, vol. 11495, pp. 215–225. Springer (2019)
32. Peled, D.: All from one, one for all: on model checking using representatives. In: Proceedings CAV (International Conference on Computer Aided Verification). Lecture Notes in Computer Science, vol. 697, pp. 409–423. Springer (1993)
33. Peled, D.: Combining partial order reductions with on-the-fly model-checking. In: Proceedings CAV (International Conference on Computer Aided Verification). Lecture Notes in Computer Science, vol. 818, pp. 377–390. Springer (1994)
34. Plagge, D., Leuschel, M.: Validating B, Z and TLA+ using ProB and Kodkod. In: Proceedings FM (International Symposium on Formal Methods). Lecture Notes in Computer Science, vol. 7436, pp. 372–386. Springer (2012)
35. Schmidt, J., Leuschel, M.: Improving SMT Solver Integrations for the Validation of B and Event-B Models. In: Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems). Lecture Notes in Computer Science, vol. 12863, pp. 107–125. Springer (2021)

36. Siegel, S.F.: What's Wrong with On-the-Fly Partial Order Reduction. In: Proceedings CAV (International Conference on Computer Aided Verification). Lecture Notes in Computer Science, vol. 11562, pp. 478–495. Springer (2019)
37. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Proceedings TACAS (International Conference on Tools and Algorithms for the Construction and Analysis of Systems). Lecture Notes in Computer Science, vol. 4424, pp. 632–647. Springer (2007)
38. Valmari, A.: Stubborn sets for reduced state space generation. In: Proceedings ICATPN (International Conference on Application and Theory of Petri Nets). Lecture Notes in Computer Science, vol. 483, pp. 491–515. Springer (1989)
39. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA+ specifications. In: Proceedings CHARME (Advanced Research Working Conference on Correct Hardware Design and Verification Methods). Lecture Notes in Computer Science, vol. 1703, pp. 54–66. Springer (1999)