Optimising the ProB Model Checker for B using Partial Order Reduction*

Ivaylo Dobrikov, Michael Leuschel

Institut für Informatik, Heinrich-Heine Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf, Germany
{dobrikov,leuschel}@cs.uni-duesseldorf.de

Abstract. Partial order reduction has been very successful at combatting the state explosion problem [4], [9] for lower-level formalisms, but has thus far made hardly any impact for model checking higher-level formalisms such as B, Z or TLA$^+$. This paper attempts to remedy this issue in the context of the increasing importance of Event-B, with its much more fine-grained events and thus increased potential for event-independence and partial order reduction. This paper provides a detailed description of a partial order reduction in ProB. The technique is evaluated on a variety of models. Additionally, the implementation of the method is discussed, which contains new constraint-based analyses.

Key words: Model Checking, Partial Order Reduction, Static Analysis, Event-B.

1 Introduction

ProB [14] is a toolset for validating systems formalised in B, Event-B, CSP, TLA+ and Z. Initially developed for B, ProB comprises an animator, a model checker, and a refinement checker. Using the ProB model checker for consistency checking of B and Event-B models is a convenient way of searching for errors in the model. In contrast to interactive theorem provers, model checking performs tasks like invariant and deadlock freedom checking automatically.

B offers a variety of data structures and B models are often infinite state. Making such a B machine manageable for model checking requires setting bounds on the types of the variables. However, even systems with finite types can have very large state spaces. Therefore, applying various optimisation techniques is essential for practical model checking of B or Event-B specifications.

Partial order reduction reduces the state space by taking advantage of independence between actions. The reduction relies on choosing only a subset of all enabled actions in each reachable state of the state space. In the process of choosing such a subset, certain requirements have to be satisfied so that no new error states (deadlocks) are introduced and no important executions for the verification of the underlying system are pruned. There are several theories [8], [11], [19]

* This research is being carried out as part of the DFG funded research project GEPAVAS.

ensuring the soundness of such a type of reduction. Our implementation of partial order reduction uses the ample set theory which is suggested as a method for partial order reduction in [4], [8], [9].

Our optimisation uses a static analysis for determining the relations between each pair of operations or events in a B or Event-B machine, respectively. The static analysis is executed prior to the model checking and is based on both syntactic and new constraint-based analyses. These analyses are used for discovering the mutual influences of actions inside the model. In this paper we present an implementation of partial order reduction in the standard ProB model checker [14] for the formalisms B [1] and Event-B [2]. In addition, we evaluate the implementation on several case study models, and discuss the implementation and its limitations. For practical reasons, we will concentrate our review of the implementation of partial order reduction on Event-B only.

Indeed, Event-B events are much more fine-grained than typical operations in classical B (e.g. an if-then-else is decomposed into two separate events in Event-B). As such, the potential for finding independent events and partial order reduction is greater. Our intuition is that the more fine-grained nature of events in Event-B should dramatically increase the potential for partial order reduction.

In the next section, we give a brief overview of the Event-B formalism and consistency checking algorithm in ProB, as well as basic definitions and notation are introduced. In Section 3, we discuss and define formally relations between events that are relevant for this work. Section 4 presents the method and the algorithm. The evaluation and the discussion of the implementation are given in Section 5. The related work is outlined in Section 6. Finally, we discuss future improvements and features for the reduced state space search, and draw the conclusions of our work.

2 Preliminaries

Event-B. Event-B is a formal language for modelling and analysing of hardware and software systems. The formal development of a system in Event-B is a state-based approach using two types of components for the description of the system: contexts and machines.

The machines represent the dynamic part of the model and each machine is comprised primarily of variables, invariants, and events. The variables are typecast and constrained by the invariants. The variables determine the states of the machine. In turn, the states of the machine are related to each other by means of the events. Each event consists of two main parts: guards and actions. Formally, an event can be described as follows:

$$\textbf{event } e \ = \ \textbf{any } t \textbf{ where } G(x,t) \textbf{ then } S(x,t,x') \textbf{ end}$$

In the definition above, x and x' stand for the evaluation of the variables before and after the execution of the event e, respectively. The parameters t in the **any** clause are typecast and restricted in the enabling predicate $G(x,t)$ of the event. The enabling predicate of an event e will be often denoted as the guard of e.

The actions part $S(x, t, x')$ of an event is composed of a number of assignments to state variables. When the event is executed, all assignments in $S(x, t, x')$ are completed simultaneously. All non-assigned variables remain unaltered.

The event e is said to be enabled in a particular state s of the machine if $G(x, t)$ holds for the current evaluation of the variables of s. Otherwise, we say that the event e is disabled in s.

Notation and Basic Definitions. When we talk about enabled events in a particular state s, we mean all events whose enabling predicates hold in s. The set of all events that are enabled in a state s will be denoted by $enabled(s)$.

By definition, an event in Event-B may have parameters and non-deterministic assignments. Thus, in some state s an event e can have several representations, i.e. there is more than one successor state s' such that $s \xrightarrow{e} s'$. In that case, we say that e is a non-deterministic event. For simplicity, from now on we will assume that each event is *deterministic*. However, the optimisation in this work has been implemented for the general case where non-determinism is present.

An event is called a *stutter* event if it preserves the truth value of each atomic proposition of the property being checked. By property we mean an LTL formula or invariant of an Event-B machine. Formally, an event e is stuttering w.r.t. a property ϕ if for each transition $s \xrightarrow{e} s'$ it is fulfilled that for each atomic proposition p of ϕ either $s \models p$ and $s' \models p$ or $s \not\models p$ and $s' \not\models p$.

The implementation of the partial order reduction technique presented in this work is realised by the ample set theory. The reduction of the state space happens by choosing a subset of $enabled(s)$ in each state s. These subsets we will denote by $ample(s)$. In the context of partial order reduction, a state s is then said to be *fully expanded* if $ample(s) = enabled(s)$.

The Consistency Checking Algorithm. Since the main contribution of this work is the optimisation of the consistency checking algorithm for Event-B and B, we will give a quick overview of it (Algorithm 1).

The pseudo code in Algorithm 1 describes a graph traversal algorithm for exhaustive error search in a directed transition system. All unexplored nodes in the state space are stored in a standard queue data structure *Queue* while running the consistency check for the particular Event-B machine. By popping unexplored states from the front or the end of the queue a depth-first search or a breadth-first search through *Graph* can be simulated, respectively. A mixed depth-first/breadth-first search can be simulated by a randomised popping from the front and end of the queue. This is the standard search strategy in ProB.

Once an unexplored state has been chosen from the queue, it will be checked for errors by the function *error* (line 4). An error state, for example, can be a state that violates the invariant of the machine or that has no outgoing transitions.

If no error has been found in the current state, then it will be expanded. In this context, expansion means that all events from the current machine will be applied to the current state. Each event whose enabling predicate $G(x, t)$ holds for the current variables' evaluation will be executed and a possible new successor state will be generated. Subsequently, a new transition will be added to the state space (line 8) if not already present in *Graph*, and a new state *succ*

Algorithm 1: Consistency Checking

```
1  Queue := {root} ; Visited := {}; Graph := {};
2  while Queue is not empty do
3      state := get_state(Queue)
4      if error(state) then
5          return counter-example trace in Graph from root to state
6      else
7          for all succ,evt such that state -evt-> succ do        the code to
8              Graph := Graph ∪ {state -evt-> succ};              be optimised
9              if succ ∉ Visited then
10                 push_to_front(succ, Queue);
11                 Visited := Visited ∪ {succ}
12             end if
13         end for
14
15     end if
16 end while
17 return ok
```

will be adjoined to the queue (line 10) if not already visited. The algorithm runs as long as the queue is non-empty and no error state has been found.

Since the way of adding transitions to the state space will become slightly different in order to apply partial order reduction, the most relevant part of Algorithm 1 for this paper is thus the pseudo code in lines 7-13.

3 Event Relations

Finding out how the events of an Event-B machine are related to each other is a key step for applying partial order reduction. The simplest approach just analyses the syntactic structure. For this, we first need to determine the *read* and *write* sets for each event. For an event e, we denote by $read(e)$ the set of the variables that are read by e, and by $write(e)$ the set of the variables that are written by e. With $read_G(e)$ and $read_S(e)$ we will denote the sets of the variables that are read in the guard and in the actions part of the event e, respectively. To simplify the presentation we assume that each event is deterministic.

Introducing Independence. The most important event relation is independence. Formally, one can define independence between two events as follows:

Definition 1 (Independence).
Two events e_1 and e_2 are independent if for any state s with $e_1, e_2 \in enabled(s)$ it is satisfied that the executions $s \xrightarrow{e_1} s_1 \xrightarrow{e_2} s'$ and $s \xrightarrow{e_2} s_2 \xrightarrow{e_1} s''$ are feasible in the state space (enabledness), and additionally $s' = s''$ (commutativity).

Two events e_1 and e_2 are said to be *syntactically independent* if the following three conditions are satisfied:

(SI 1) The read set of e_1 is disjoint to the write set of e_2 ($read(e_1) \cap write(e_2) = \varnothing$).
(SI 2) The write set of e_1 is disjoint to the read set of e_2 ($write(e_1) \cap read(e_2) = \varnothing$).
(SI 3) The write sets of e_1 and e_2 are disjoint ($write(e_1) \cap write(e_2) = \varnothing$).

Two syntactically independent events are independent by means of Definition 1 since no event can affect the guard of the other one (enabledness) and additionally the read and write sets of each of both events are disjoint to the write set of the other one (commutativity).

On the other hand, syntactical independence is obviously a quite coarse concept: two events of an Event-B machine can be independent even if some of the conditions (SI 1) - (SI 3) are violated. Take for example the following two events:

Example 1 (Event Dependency).

event e_1 =
  when
    $x \in \mathbb{N}$
  then
    $y := y + 1$
  end

event e_2 =
  when
    $z \geq 1 \wedge z \leq 10$
  then
    $x := z \parallel z := z + 1$
  end

Apparently, e_1 and e_2 are not syntactically independent as (SI 1) is violated ($read(e_1) \cap write(e_2) = \{x\}$). However, e_2 cannot affect the guard of e_1 because e_2 can assign to x only values between 1 and 10, and e_1 is enabled when x is a natural number. Since additionally $write(e_1) \cap read(e_2) = \varnothing$, it follows that the *enabledness* condition for independence for e_1 and e_2 is fulfilled. Further, no variable written by the one event will be read in the actions part of the other event and the write sets of e_1 and e_2 are disjoint. Thus, both events cannot interfere each other and herewith the *commutativity* condition for independence is fulfilled for e_1 and e_2. Hence, e_1 and e_2 are indeed independent events.

Since partial order reduction takes advantage of the independence between events, it is important to determine independence as accurately as possible. The higher the degree of independence in a system, the higher is the chance to reduce its state space significantly. This motivates the following, more precise approach to determine independence by using the ProB's constraint solving facilities.

Refining the Dependency Relation. We use the constraint solver to find feasible sequences of events for the analysed Event-B model. First, we define a procedure stating a Prolog predicate in ProB used for testing whether a given sequence of events is feasible. This will form the basis of our analysis.

Definition 2 (The *test_path* procedure).
*For a given Event-B machine M, let Φ and Ψ be B predicates for M, and $e_1, \ldots, e_n$ events of M. Then, we define **test_path** as follows:*

$$test\_path(\Phi, \langle e_1, \ldots, e_n \rangle, \Psi) = \begin{cases} true & \text{if there is an execution } s \xrightarrow{e_1} \ldots \xrightarrow{e_n} s' \\ & \text{such that } s \models \Phi \text{ and } s' \models \Psi \\ false & \text{otherwise} \end{cases}$$

The predicates Φ and Ψ are used in order to constrain the search for possible test paths for M. If, for example, Φ and Ψ are both equal to the truth value $TRUE$ then *test_path* will return *true* if the given sequence of events is possible from some valid state of M.

We can now refine our definition of independence. We introduce the binary relation $Dependent_M \subseteq Events_M \times Events_M$ which is intended to comprise all dependent pairs of events of a given Event-B machine M. Two events e_1 and e_2 will be denoted as dependent if $(e_1, e_2) \in Dependent_M$, otherwise they are considered to be independent. The dependency relation is defined as follows:

$$Dependent_M := \{(e, e') \mid (e, e') \in Events_M \times Events_M \wedge dependent(e, e')\},$$

where M is the observed Event-B machine, $Events_M$ is the set of events of M and *dependent* is the procedure showed in Algorithm 2.

Algorithm 2: Determining Events' Dependency

```
1  procedure boolean dependent(e1, e2)
2      if write(e1) ∩ write(e2) ≠ ∅ then
3          return true                    /* events are race dependent */
4      else if (read(e1) ∩ write(e2) = ∅ ∧ write(e1) ∩ read(e2) = ∅) then
5          return false     /* events are syntactically independent */
6      else
7          return
8          (read_S(e1) ∩ write(e2) = ∅ ∧ write(e1) ∩ read_S(e2) = ∅) ⇒
9            ((read_G(e1) ∩ write(e2) ≠ ∅ ∧ test_path(G_e1 ∧ G_e2, · -e2-> ·, ¬G_e1))
10         ∨ (write(e1) ∩ read_G(e2) ≠ ∅ ∧ test_path(G_e2 ∧ G_e1, · -e1-> ·, ¬G_e2))
11     end if
```

The procedure *dependent* presents a refined strategy for determining the dependency between two events. The **else** branch in Algorithm 2 will be executed if at least one of the two events modifies a variable that is read by the other one. In order to test whether two events are independent, we need to check the two independence conditions *enabledness* and *commutativity*. The test for dependency is expressed by means of the predicate in lines 8-10. We are interested mainly in the case when the predicate evaluates to $false$. This is clearly fulfilled when the left side of the implication holds and the right side evaluates to $false$. In case the premise of the implication

$$(read_S(e_1) \cap write(e_2) = \varnothing \wedge write(e_1) \cap read_S(e_2) = \varnothing)$$

is satisfied, then it is assured that both events cannot affect each other (at this point we know that the write sets of e_1 and e_2 are disjoint) and thus the commutativity condition for independence is satisfied in case the events cannot disable each other. Once we know that e_1 and e_2 cannot interfere, we need to check the enabledness condition. The enabledness condition is tested by the two disjunction arguments in lines 9 and 10. If at least one of the arguments is fulfilled, we have deduced that e_1 and e_2 are indeed dependent. Otherwise, we have proven that e_1 and e_2 are independent.

Checking whether the events can disable one other is realised by means of the *test_path* procedure. If, for example, e_2 assigns a variable that is read in the guard G_{e_1} of e_1 (i.e. if $read_G(e_1) \cap write(e_2) \neq \varnothing$) then we can further check whether e_2 eventually can disable e_1. This can be additionally examined by searching for a possible transition $s \xrightarrow{e_2} s'$ such that e_1 and e_2 are enabled in s ($s \models G_{e_1} \wedge G_{e_2}$) and e_1 disabled in s' ($s' \models \neg G_{e_1}$). The call for this case is then $test\_path(G_{e_1} \wedge G_{e_2}, \cdot \xrightarrow{e_2} \cdot, \neg G_{e_1})$. If the result of the call is *true* then we have found a case in which e_2 can disable e_1 and thus inferred that e_1 and e_2 are dependent. Otherwise, we have shown that the enabling condition of e_1 cannot be affected by the execution of e_2.

The Enabling Relation. In addition to the independence of events, we are also interested in the particular way events may influence each other. Concretely, if event e_1 modifies some variables in the guard of event e_2 we are asking in which way the effect of e_1 may affect the guard of e_2. In that case, the possible direct influences of e_1 to e_2 can be *enabling* and *disabling*. The enabling relation is the residual relation needed for applying the optimisation technique in this work.

In the next section we are interested whether events can be enabled after the successively execution of a number of certain events. We will retain the enabling information between events in terms of a directed edge graph, defined as follows:

Definition 3 (Enable Graph). *An enable graph for an Event-B machine M is a directed edge graph $\mathtt{EnableGraph}_M = (V, E)$, where $V = Events_M$ are the vertices and $E = \{e_1 \mapsto e_2 \mid e_1, e_2 \in Events_M \ \wedge \ can\_enable(e_1, e_2)\}$ the edges of $\mathtt{EnableGraph}_M$.*

In Definition 3, $e_1 \mapsto e_2$ means that e_1 can enable e_2, while *can_enable* constitutes a procedure which returns $false$ when $write(e_1) \cap read_G(e_2) = \varnothing$, otherwise tests if e_1 can enable e_2 by means of the *test_path* procedure. The call of *test_path* for testing whether e_1 may enable e_2 is then $test\_path(G_{e_1} \wedge \neg G_{e_2}, \cdot \xrightarrow{e_1} \cdot, G_{e_2})$.

4 Algorithm

In this section we introduce the theory of partial order reduction and the algorithm for the expansion of states by using the ample set method. The reduction of the original state space using ample sets is realised by choosing of a subset of all enabled events in each state.

The Ample Set Requirements. There are four requirements that should be satisfied by each ample set to make the reduction of the state space sound:

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

(A 2) Dependency Condition
Along every finite execution in the original state space starting in s, an event dependent on $ample(s)$ cannot appear before some event $e \in ample(s)$ is executed.

(A 3) Stutter Condition
If $ample(s) \subsetneq enabled(s)$ then every $e \in ample(s)$ has to be a stutter event.

(A 4) Cycle Condition
For any cycle C in the reduced state space, if a state in C has an enabled event e, then there exists a state s in C such that $e \in ample(s)$.

The Need of Local Criteria for (A 2). We are interested in how efficiently each of the requirements can be checked. For a state s, the conditions (A 1) and (A 3) can be checked by examining the events in $ample(s)$. In contrast to conditions (A 1) and (A 3), condition (A 2) is a global property which requires for $ample(s)$ the examination of all possible executions (in the original state space) starting in s. A straightforward checking of (A 2) will demand the exploration of the original state space. Local criteria thus need to be given for (A 2) that facilitate an efficient computation of the condition.

For our implementation, we define the following two local conditions (which will replace (A 2)), where M is the observed Event-B machine, $Events_M$ the set of events in M, and s a state in the original state space:

(A 2.1) Direct Dependency Condition
Any event $e \in enabled(s) \setminus ample(s)$ is independent of $ample(s)$.

(A 2.2) Enabling Dependency Condition
Any event $e \in Events_M \setminus enabled(s)$ that depends on $ample(s)$ may not become enabled through the activities of events $e' \notin ample(s)$.

The following theorem states that (A 2.1) and (A 2.2) are sufficient local criteria for (A 2). The proof of Theorem 1 can be examined in [10].

Theorem 1 (Sufficient Local Criteria for (A 2)).
Given a state s in the original state space. If $ample(s)$ is computed with respect to the local criteria (A 2.1) and (A 2.2), then $ample(s)$ satisfies (A 2) for all execution fragments in the original state space starting in state s.

Computing *ample(s)*. We can now present our algorithm for computing an ample set satisfying (A 1) through (A 3). The procedure *ComputeAmpleSet* in Algorithm 3 gets as argument a set of events.$Dependent_M$ and $EnableGraph_M$ are the dependent relation and the enable graph computed for the corresponding Event-B machine M, respectively (see Algorithm 2 and Definition 3). The procedure *ComputeAmpleSet* uses the *DependencySet* procedure for computing a set S satisfying the local dependency condition (A 2.1). In the body of procedure *DependencySet* the set G is regarded as directed graph where the vertices are represented by the events of T and the edges by tuples $\alpha \mapsto \beta$. The tuple $\alpha \mapsto \beta$, for example, represents an edge from vertex α to vertex β. By $reachable(\alpha, G)$ we denote the set of vertices that are reachable from vertex α in G. The set T is meant to be $enabled(s)$, where s is the currently processed state. Accordingly, the set S in Algorithm 3 is intended to be $ample(s)$. The output of the *ComputeAmpleSet* is an ample set $ample(s)$ satisfying the first three conditions of the ample set constraints.

Algorithm 3: Computation of *ample*(s)

```
1  procedure set ComputeAmpleSet(T)
2      foreach α ∈ T such that α randomly chosen do
3          b := true;
4          S := DependencySet(α, T);                    /* (A 2.1) holds */
5          I := T \ S;
6          foreach β ∈ I do /* checking whether S fulfils (A 2.2) */
7              if there is a path β → γ1 → ... → γn → γ in EnableGraph_M
               such that   γ1, ..., γn, γ ∉ S ∧ γ depends on S then
8                  b := false;
9                  break
10             end if
11         end foreach
12         if b ∧ (S is a stutter set) then             /* checking (A 3) */
13             return S
14         end if
15     end foreach
16     return T
17 end procedure

18 procedure set DependencySet(α, T)
19     G := ∅;
20     foreach (β, γ) ∈ Dependent_M ∩ (T × T) do
21         G := {β ↦ γ} ∪ G
22     end foreach
23     return reachable(α,G)
24 end procedure
```

The first step of computing *ample*(s), in case that T is a non-empty set, is choosing randomly an event α from T. After that, a subset S of all enabled events in s in regard to α is computed such that condition (A 2.1) is satisfied (line 4). The set of events S is determined by means of the *DependencySet* procedure (lines 18-24). Once the set S in regard to the randomly chosen event α is computed, we test whether there may be an event β that is not from S and from which a finite execution fragment

$$\sigma = s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_n} s_n \xrightarrow{\gamma} s_{n+1}$$

can start such that an event γ dependent on S may be enabled before executing some event from S (i.e. $\gamma_1, \ldots, \gamma_n \notin$ *ample*(s)). This we do by means of looking for paths in $EnableGraph_M$ having as a starting point the event β and reaching an event $\gamma \notin S$ which is dependent on S. In other words, in lines 6-11 of procedure *ComputeAmpleSet* we test if S further satisfies the second local dependency condition (A 2.2). If there is some event $\beta \in I$ for which condition (A 2.2) is violated we choose randomly the next event from T in order to compute a new potential ample set. Otherwise, if for all $\beta \in I$ there is no path

in $EnableGraph_M$ that presumptively represents an execution in TS_M violating (A 2.2), we check whether S fulfils the stutter condition (line 12). The procedure $ComputeAmpleSet$ in Algorithm 3 runs until an appropriate ample set has been found or all potential ample sets fail to satisfy conditions (A 2) and (A 3) (the we return T). A proof of the correctness of Algorithm 3 can be found in [10].

The Ignoring Problem. Condition (A 3), which requires adding only of stutter events to the ample sets of each state (assuming that (A 1) and (A 2) are also satisfied), can sometimes cause ignoring of certain (non-stutter) events in the reduced state space. Ignoring of non-stutter events may happen when the reduction results in a cycle of stutter events only. If some events are ignored in the reduced state space of the model, then computing ample sets w.r.t. (A 1) through (A 3) may not be sufficient to preserve some of the LTL_{-X} properties. The issue is also known as the *ignoring* problem [19].

To ensure that no events in the reduced state space are ignored, the cycle condition (A 4) should be guaranteed by the reduced state space. We establish (A 4) by means of the following condition:

(A 4') Strong Cycle Condition
Any cycle in the reduced state space has at least one fully expanded state.

Using the strong cycle condition (A 4') is a sufficient criterion for (A 4) (Lemma 8.23 in [4]) and easier to implement. Since at least one of the states should be fully expanded in any cycle, we expand fully each state s with an outgoing transition reaching an expanded state generated before s, as well as each state with a self loop. Note that this method of implementing the strong cycle condition (A 4') is approximative because it expands fully states unnecessarily sometimes. We have chosen this way of realising (A 4') in order to generalise our algorithm of calculating ample sets for different exploration strategies. This technique of implementing (A 4) has been also proposed in other works like in [5].

Expanding a State by Applying the Ample Events Only. To apply the ample set approach for the consistency checking algorithm, we change the way each state is expanded. Thus, the respective changes in Algorithm 1 take place in lines 7-13 of the algorithm. Basically, we can replace the code in the **else** branch of Algorithm 1 by calling the procedure *compute_ample_transitions* in Algorithm 4 with the currently processed state s as argument.

Algorithm 4 summarises the computation of the ample events in each state and the execution of those in the reduced state space. The presented procedure *compute_ample_transitions* gets as argument the state being expanded. The computation of the successor states and the insertion of the new determined transitions are realised by the procedure *execute_event*.

In Algorithm 4 all enabled events in the currently processed state s will be assigned to T (line 2). After that, an ample set S satisfying (A 1) through (A 3) is computed by means of the procedure $ComputeAmpleSet$. If the test of the cycle condition in line 7 fails for each loop-iteration, then only the events from S will be executed in s. Otherwise, the full expansion of s will be forced (lines 8-10), if a transition from S reaches an already expanded state s' ($s' \notin Queue$)

generated before s or it is s itself ($id(s) \geq id(s')$).

Algorithm 4: Computation of the Ample Transitions

```
1  procedure compute_ample_transitions(s)
2      T := compute all enabled events in s;
3      S := ComputeAmpleSet(T);
4      foreach evt ∈ S do
5          s' := execute_event(s,evt);
6          T := T \ {evt}
7          if (id(s) ≥ id(s')) ∧ s' ∉ Queue then        /* check (A 4) */
8              foreach e ∈ T do
9                  execute_event(s,e)
10             end foreach
11             break          /* state s has been fully explored */
12         end if
13     end foreach
```

5 Discussion and Evaluation

Discussion. In Section 4, we presented the background of the ample set theory and our implementation of partial order reduction (Algorithms 3 and 4). Our algorithm reduces the original state space of an Event-B machine M by using the dependency relation $Dependent_M$ and the enable graph $EnableGraph_M$. $Dependent_M$ and $EnableGraph_M$ are computed prior to the model checking by using a static analysis on the events of M. We chose to determine the dependency and enabling relations between the events in this way for performance reasons. Computing the respective relations between events on-the-fly in each state can sometimes be expensive since we use constraint based analyses in addition to syntactic analysis. In fact, timeouts are set by default in ProB for diminishing the possibility that the overhead caused by static analysis and partial order reduction outweighs the improvement achieved by the reduction of the state space. ProB can also apply partial order reduction without using its constraint solving facilities. In this case, the determination of the dependency and enabledness between events is provided by inspecting their syntactic structure only. This, however, often results in less state space reduction.

The reduction of the state space by using partial order reduction cannot only be influenced by the independence of the events of the model being verified, but also by the type of the checked property. For instance, deadlock preservation is guaranteed by any ample set satisfying conditions (A 1) and (A 2) [12], [19]. We adapted the implementation to this fact to gain more state space reduction when a model is checked for deadlock freedom only.

Another factor that can influence the effectiveness of the reduction is the number of the stutter events. For example, if we check the full invariant I, then every event that trivially fully preserves I is a stutter event. Systems specified in Event-B often have a very low number, if any, of events that trivially fulfil

the invariant. This means that partial order reduction will probably only yield minor state space reduction in such cases. A possible way to detect more stutter events w.r.t. I is to use either proof information (from the Rodin provers) or ProB for checking invariant preservation for operations: any event which we can prove to preserve the invariant now becomes a stuttter event.

Evaluation. We have evaluated our implementation of partial order reduction on various models that we have received from academia and industry.[1] A part of those experiments are presented in Table 1. In particular, we wanted to study the benefit of the optimisation on models with large state spaces.

Besides having sizeable state spaces, the particular models should also have a certain number of independent concurrent events. Otherwise, the possibility of reducing the state space is very minor. If, for instance, we have a system where there is no pair of independent events or a system where any two independent events are never simultaneously enabled, then no reductions of the state space can be gained at all.

Table 1 - Part of the Experimental Results (times in seconds)

Model	Algorithm	States	Transitions	Analysis Time	Model Checking Time
Counters	MC	3,974	11,485	-	3.417*
	MC+POR	961	1,807	< 0.001	0.823*
	MC-NoINV	110,813	325,004	-	73.167
	MC-NoINV+POR	152	154	0.010	0.097
BPEL v6	MC	2,248	4,960	-	7.437
	MC+POR	2,248	4,960	0.748	7.884
	MC-NoINV	2,248	4,960	-	6.944
	MC-NoINV+POR	847	1,004	0.640	2.670
Token Ring	MC	8,196	45,077	-	14.291
	MC+POR	8,176	40,565	0.011	14.671
	MC-NoINV	8,196	45,077	-	13.814
	MC-NoINV+POR	4,776	12,129	0.016	7.807
Sieve	MC	8,328	28,436	-	215.138
	MC+POR	8,142	25,237	12.437	217.754
	MC-NoINV	8,328	28,436	-	220.864
	MC-NoINV+POR	6,421	14,557	12.439	186.101
Phil v2	MC	2,350	4,528	-	9.086
	MC+POR	2,347	4,390	0.406	9.354
	MC-NoINV	2,350	4,528	-	8.870
	MC-NoINV+POR	2,346	4,336	0.378	9.167

(*) Invariant Violation

We have performed four different types of checks in order to measure the performance of our implementation of partial order reduction. By all types of tests we used the mixed depth-first/breadth-first search of ProB for the exploration of the state space. The four types of checks are abbreviated in Table 1 as follows:

[1] The models and their evaluations can be obtained from the following web page http://nightly.cobra.cs.uni-duesseldorf.de/por/

MC: Model checking by using the standard consistency checking algorithm.
MC+POR: Model checking with partial order reduction.
MC-NoINV: Model checking by using the standard consistency checking algorithm without invariant violations checking.
MC-NoINV+POR: Model checking with partial order reduction without invariant violations checking.

The consistency checking algorithm and the partial order reduction algorithm are respectively Algorithm 1 and Algorithm 4. For the evaluations we used model checking for searching for deadlocks and invariant violations only.[2] Due to the fact that checking for deadlock freedom only requires the satisfaction of the ample set conditions (A 1) and (A 2) for the reduced search, we additionally observed experiments with MC-NoINV+POR. For this type of checks, the results produced by MC-NoINV+POR were compared with the results of MC-NoINV.

One specification, *Counters*, in Table 1 is given that represents the best case for the reduced search in ProB. *Counters* is a toy example aiming to show the benefit of partial order reduction when each event in the model is independent from the executions of all other events. The worst case, when no reductions of the state space are gained, is represented by checking *BPEL v6* with MC+POR. *Phil* [7] and *BPEL* [3] are case studies of the dinning philosophers problem with four philosophers and of a business process for a purchase order, respectively. Both are carried by a stepwise development via refinement; their last refinement versions *Phil v2* and *BPEL v6* are presented in Table 1. *Token Ring* is a B model of a token ring protocol and *Sieve* an Event-B model formalising a parallel version (for four processes) of the algorithm of sieve of eratosthenes for computing all prime numbers from 2 to 40.

All measurements were made on an Intel Xeon Server, 8 x 3.00 GHz Intel(R) Xeon(TM) CPU with 8 GB RAM running Ubuntu 12.04.3 LTS. The Analysis times in Table 1 are the measured runtimes for the static analysis of each machine. If the POR option is not set in an experiment, no static analysis is performed. Each experiment has been performed ten times and its respective geometric means (states, transitions and times) are reported in the results.

In general, the most considerable reductions of the state space were gained with the reduced search when only deadlock freedom checks were performed. We consider both the reductions of the number of states and transitions. In one case (*BPEL v6*), no reductions of the state space were gained using the reduced search MC+POR. However, the model checking runtimes in those cases are not significantly different from the model checking runtimes for the standard search MC. As expected, significant reduction of the state space and thus the overall time for checking the *Counters* model were gained by both reduction searches MC+POR and MC-NoINV+POR. For the test cases MC and MC+POR of *Counters* an invariant violation was found which led to a termination of the respective search. Interesting results were obtained when applying any of the reduced searches on the *Phil v2* model. Although the model has a great magnitude

[2] Another options like finding a goal or searching for assertion violations have not been checked while model checking the particular model.

of independence, the coupling between the events is so tight that no significant reductions can be gained.

6 Related Work

Several works have been devoted to optimising the ProB model checker for B and Event-B. In this section, we refer to some of the techniques have been developed and analysed for the ProB model checker.

Symmetry reduction is a technique successfully implemented in ProB for combating the state space explosion problem. Using the fact that symmetry is induced by the deferred sets in B, two sorts of exhaustive symmetry reduction algorithms in ProB have been implemented: the graph canonicalisation method [18] and the permutation flooding method [15]. The general idea of both techniques is to check only a single representative of each symmetry class of equivalent states during the consistency check of the model being verified. An approximative symmetry reduction method [16] based on computing symmetry markers for states of B machines has been also implemented in ProB. The idea of the method is that two states are considered to be symmetrically equivalent if they have the same symmetrical marker. All three methods showed good performance results when model checking B or Event-B models with a certain degree of symmetry induced by B's deferred sets.

Another notion of optimising the ProB model checker has been presented in [6]. The idea of this work is to improve the efficiency of the model checker by using the already discharged proof information from the front-end environment. The verification technique, known as proof assisted model checking, is used by default in ProB and has shown a performance improvement up to factor two on various industrial models.

Other techniques, such as using mixed breadth-first/depth-first search strategy and heuristic functions for performing directed model checking [13], have been also suggested as optimisation methods for the standard ProB model checker.

7 Conclusion and Future Work

Partial order reduction has been very successful for lower-level models such as Promela, but has had relatively little impact for higher-level modelling languages such as B, Z or TLA$^+$. Inspired by Event-B's more simpler event structures and more distributed nature, we have started a new attempt at getting partial order reduction to work for high-level formal models. We have presented an implementation of partial order reduction in ProB for Event-B (and also classical B) models. The implementation makes use of the ample set theory for reducing the state space and uses new constraint-based analyses to obtain precise relations of influence between events. Our evaluation of the reduction method has shown that considerable reductions of the state space can be gained for models with a high degree of independence and concurrency. We also observed that check-

ing only for deadlock freedom tends to provide more significant reductions than checking simultaneously for invariant violations and deadlock freedom.

Our approach of satisfying the Cycle condition (A 4) is an approximative method for loop detection during the reduced expansion of the state space. Finding possible cycles in the reduced state space simply by checking whether the currently processed state has an outgoing transition to an already expanded state can cause less state space reductions, since the full exploration of a state can also be forced when no true cycles are discovered. For this reason, future work will concentrate on improving the reduction algorithm w.r.t. the Cycle detection condition. Further work will need to be done in elaborating the reduction algorithm presented in this work for the LTL model checker [17] in ProB.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.
2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
3. I. Ait-Sadoune and Y. Ait-Ameur. A Proof Based Approach for Modelling and Verifying Web Services Compositions. ICECCS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.
4. C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.
5. J. Barnat, L. Brim, and P. Rockai. Parallel Partial Order Reduction with Topological Sort Proviso. In *SEFM*, pages 222–231. IEEE Computer Society, 2010.
6. J. Bendisposto and M. Leuschel. Proof Assisted Model Checking for B. In K. Breitman and A. Cavalcanti, editors, *Proceedings of ICFEM 2009*, volume 5885 of *LNCS*, pages 504–520. Springer, 2009.
7. P. Boström, F. Degerlund, K. Sere, and M. Waldén. Derivation of Concurrent Programs by Stepwise Scheduling of Event-B Models. *Formal Aspects of Computing*, pages 1–23, 2012.
8. E. Clarke, O. Grumberg, M. Minea, and D. Peled. State Space Reduction using Partial Order Techniques. *International Journal on STTT*, 2(3):279–287, 1999.
9. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
10. I. Dobrikov and M. Leuschel. Optimising the ProB Model Checker for B using Static Analysis and Partial Order Reduction (Technical Report). http://www.stups.uni-duesseldorf.de/mediawiki/images/5/5b/Pub-DobrikovLeuschelPORtechreport.pdf, 2014.
11. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.
12. P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In K. G. Larsen and A. Skou, editors, *CAV*, volume 575 of *LNCS*, pages 332–342. Springer, 1991.
13. M. Leuschel and J. Bendisposto. Directed Model Checking for B: An Evaluation and New Techniques. In J. Davies, L. Silva, and A. da Silva Simão, editors, *SBMF'2010*, volume 6527 of *LNCS*, pages 1–16. Springer, 2010.
14. M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *STTT*, 10(2):185–203, 2008.

15. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry Reduction for B by Permutation Flooding. In *Proceedings B'2007*, volume 4355 of *LNCS*, pages 79–93. Springer-Verlag, 2007.
16. M. Leuschel and T. Massart. Efficient Approximate Verification of B via Symmetry Markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, January 2007.
17. D. Plagge and M. Leuschel. Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. *STTT*, 12(1):9–21, Feb 2010.
18. E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry Reduced Model Checking for B. In *Proceedings TASE 2007*, pages 25–34. IEEE, 2007.
19. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *Applications and Theory of Petri Nets*, pages 491–515, 1989.