Optimising the ProB Model Checker for B using Partial Order Reduction* (technical report)

Ivaylo Dobrikov, Michael Leuschel

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

Abstract. Partial order reduction has been very successful at combatting the state explosion problem [4, 11] 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. Further, we give a comprehensive description for elaborating the implementation into the LTL model checker of ProB.

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

1 Introduction

ProB [16] 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

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

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 [10, 13, 27] 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, 10, 11].

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 [16] 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. We give also a comprehensive description of the $\text{LTL}^{[e]}$ model checking algorithm in ProB and consider ways of incorporating the reduction method for model checking LTL_{-X} properties in ProB. 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

2.1 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:

Event with local variables:

event $e =$
  any
    t /* the local variables */
  where
    $G(x,t)$ /* the guards */
  then
    $S(x,t,x')$ /* the actions */
  end

Event without local variables:

event $e =$
  when
    $G(x)$ /* the guards */
  then
    $S(x,x')$ /* the actions */
  end

In the definition above x stands for the evaluation of the variables before the execution of the event e and x' for the evaluation of the variables after the execution of the event e. In the **any** clause the parameters t of the event will be defined, these will be typecasted and restricted in the guards of the event. Note that events may have no parameters. In that case the **any** clause will be omitted and the keyword **when** is used instead of **where**. We will denote in this work $G(x,t)$ as the guard of the event e. Basically, in Event-B $G(x,t)$ is a predicate which is a conjunction of all particular guards 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. It is possible that an event does not assign any variable of the machine. In this case all variables remain unchanged and the actions block consists of the **skip** declaration only.

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 at s. An event e that is enabled at some state s can be executed and as a result of executing its actions a state s' is reached. Each state s at which e is enabled we will denote as a *before-state* of e and each state reached by e will be characterized as an *after-state* of e.

In this work we are particularly interested in how events of an Event-B machine are related to each other. Since it is often the case that events have common write and read variables, they can affect each other in the process of their execution. For example, an event e_1 may enable or disable another event e_2 after its execution if e_1 assigns variables whose values will be read in the guard of the event e_2. On the other hand, events that do not affect one other and do not interfere also exist and are called *independent* events. In this article we will define and compute such types of dependence and independence relations and explain how we take advantage of such information in order to optimise model checking. In Section 3 we will give more detailed definitions of these relations between events.

2.2 Notation and Basic Definitions

When we talk about the state space of a finite-state Event-B machine M we mean the resulting state transition graph after the exploration of all possible states of the machine M. The state transition graph of an Event-B machine will be denoted as a *transition system* defined as a tuple

$$TS_M = (S, S_0, Events_M, R, AP, L),$$

where S is the set of states, $S_0 \subseteq S$ is a set of initial states, $Events_M$ the set of events of M, $R \subseteq S \times Events_M \times S$ the set of transitions, AP the set of atomic propositions, and $L : S \to 2^{AP}$ a labeling function assigning to each state s a set of atomic propositions $L(s)$ (basically $L(s)$ comprises all atomic propositions that hold in s). A transition $(s, e, s') \in R$ will often be written as follows: $s \xrightarrow{e} s'$.

When we talk about enabled events in a particular state s, we mean all events whose guards all hold in s. The set of all events that are enabled in a state s will be denoted by $enabled(s)$. If $enabled(s) = \varnothing$ for some state s in TS_M, then we say that s is a *deadlock state* or just a *deadlock*.

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 $ample(s) = enabled(s)$.

By definition, an event in Event-B may have parameters and non-deterministic assignments. Thus, in some state s an event e can be executed in several ways, 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. Formally, this means that an event e is stutter with respect to a property ϕ if for each transition $s \xrightarrow{e} s'$ in TS_M we have $L(s) \cap AP_\phi = L(s') \cap AP_\phi$, where AP_ϕ is the set of the atomic propositions in ϕ. By property, we basically mean an LTL formula or invariant of an Event-B machine. In some literature sources like in [11] the *stutter* events are referred as *invisible* events and events that are *non-stutter* as *visible* events.

A *path* in TS_M is a finite or infinite alternating sequence of states and events $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} \ldots$ in TS_M such that for all $i \geq 0$ we have $(s_i, e, s_{i+1}) \in R$. By $\pi^i = s_i \xrightarrow{e_i} s_{i+1} \xrightarrow{e_{i+1}} \ldots$ we denote the suffix of the path π. By means of the temporal logic LTL [23] one can make assertions about the temporal behaviours of a system. In [22] an extension of LTL, denoted by LTL$^{[e]}$, has been introduced allowing to state propositions also on transitions. For a finite set of atomic propositions AP, an LTL$^{[e]}$ formula is formed inductively as follows:

- $true$ and each $a \in AP$ is an LTL$^{[e]}$ formula
- $[e]$ is an LTL$^{[e]}$ for each event $e \in Events_M$
- if ϕ, ϕ_1 and ϕ_2 are LTL$^{[e]}$ formulae, then so are $\neg\phi$, $\phi_1 \vee \phi_2$, $X\phi$, and $\phi_1 U \phi_2$.

An LTL$^{[e]}$ formula ϕ is said to be satisfied by a path π in TS_M (denoted by $\pi \models \phi$) by means of the following semantics:

- $\pi \models true$
- $\pi \models p \Leftrightarrow \pi = s_0 \ldots$ and $p \in L(s_0)$, for $p \in AP_\phi$
- $\pi \models [e] \Leftrightarrow |\pi| \geq 2$ and $\pi = s_0 \xrightarrow{e} \pi^1$ for $e \in Events_M$
- $\pi \models \neg\phi \Leftrightarrow \pi \not\models \phi$
- $\pi \models \phi_1 \vee \phi_2 \Leftrightarrow \pi \models \phi_1$ or $\pi \models \phi_2$
- $\pi \models X\phi \Leftrightarrow |\pi| \geq 2$ and $\pi^1 \models \phi$
- $\pi \models \phi_1 U \phi_2 \Leftrightarrow$ there is a $k \geq 0$ such that $\pi^k \models \phi_2$ and $\pi^i \models \phi_1$ for all $0 \leq i < k$

Using the boolean connectivities $\neg$ and $\vee$ other boolean operators such as $\wedge$ and $\Rightarrow$ can be derived: $\phi_1 \wedge \phi_2 \equiv \neg(\neg\phi_1 \vee \neg\phi_2)$ and $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$. The temporal operators F (*finally*), G (*globally*), R (*release*), and W (*weak-until*) can be derived using the basic LTL operators $\neg$, $\vee$, and U:

$F\ \phi \equiv true U \phi$
$G\ \phi \equiv \neg(true\ U \neg\phi)$
$\phi_1 R \phi_2 \equiv \neg(\neg\phi_1 U \neg\phi_2)$
$\phi_1 W \phi_2 \equiv \neg(true\ U \neg\phi_1) \vee (\phi_1 U \phi_2)$

We say that a state s in TS_M satisfies an LTL$^{[e]}$ formula ϕ if for every path π starting in s we have $\pi \models \phi$. Subsequently, an Event-B model M satisfies an LTL$^{[e]}$ formula ϕ if for each initial state $s \in S_0$ of TS_M we have $s_0 \models \phi$. By $M \models \phi$ we will denote that the model M satisfies the formula ϕ.

A closure of an LTL$^{[e]}$ formula ϕ, denoted by $Cl(\phi)$, is the smallest set of formulae containing ϕ and which satisfies the following rules:

- $\psi \in Cl(\phi) \Leftrightarrow \neg\psi \in Cl(\phi)$ ($\neg\neg\phi$ is identified with ϕ)
- $\psi_1 \vee \psi_2 \in Cl(\phi) \Leftrightarrow \psi_1, \psi_2 \in Cl(\phi)$
- $X\psi \in Cl(\phi) \Leftrightarrow \psi \in Cl(\phi)$
- $\neg X\psi \in Cl(\phi) \Leftrightarrow X\neg\psi \in Cl(\phi)$
- $\psi_1 U \psi_2 \in Cl(\phi) \Leftrightarrow \psi_1, \psi_2, X(\psi_1 U \psi_2) \in Cl(\phi)$

A subset of formulae $F \subseteq Cl(\phi) \cup AP$ is consistent for a state $s \in S$ if it satisfies the following rules:

- for each atomic proposition $a \in (F \cap AP) \Leftrightarrow a \in L(s)$,
- $\psi \in F \Leftrightarrow (\neg\psi) \notin F$ for every $\psi \in Cl(\phi)$,
- $\psi_1 \vee \psi_2 \in F \Leftrightarrow \psi_1, \psi_2 \in Cl(\phi)$ for every $\psi_1 \vee \psi_2 \in Cl(\phi)$,
- if s is not a deadlock, then $(\neg X\psi) \in F \Leftrightarrow X\neg\psi \in F$ for every $\neg X\psi \in Cl(\phi)$,
- if s is a deadlock, then $(\neg X\psi) \in F$ for every $X\psi \in Cl(\phi)$,
- $\psi_1 U \psi_2 \in Cl(\phi) \Leftrightarrow \psi_2 \in F$ or $\psi_1, X(\psi_1 U \psi_2) \in F$ for every $\psi_1 U \psi_2 \in Cl(\phi)$,

A pair (s, F), where s is a state and F a consistent subset of $Cl(\phi)$, is called an *atom*. Using the *tableau construction* algorithm from [19] for checking $M \models \phi$ is based on attempting to construct a directed graph $\mathcal{A}(TS_M)$ that has an infinite α-path

$\pi_\alpha = (s_0, F_0) \xrightarrow{e_0} (s_1, F_1) \xrightarrow{e_1} (s_2, F_2) \xrightarrow{e_2} \ldots$ where $F_i \subseteq Cl(\neg\phi)$ for all $i \geq 0$ such that:

1. for every edge $(s_i, F_i) \xrightarrow{e_i} (s_{i+1}, F_{i+1})$ and for every $X\psi \in F_i$ it follows that $\psi \in F_{i+1}$,
2. $\pi = s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots$ is a path in TS_M, and
3. for every $i \geq 0$ and for every $\psi_1 U \psi_2 \in F_i$ there exists some $j \geq i$ such that $\psi_2 \in F_j$.

Clearly, $(s_i, F_i) \xrightarrow{e_i} (s_{i+1}, F_{i+1})$ is an edge in $\mathcal{A}(TS_M)$ if and only if $s_i \xrightarrow{e_i} s_{i+1}$ is a transition in TS_M and for every formula $X\psi \in F_i$ it follows that $\psi \in F_{i+1}$. For finite state systems M the transition system TS_M and so the graph $\mathcal{A}(TS_M)$ have finite number of states. An infinite α-path π_α is then represented by a finite path π_1 leading to an atom (s_k, F_k) that is an entry point of a strongly connected component (SCC) C in $\mathcal{A}(TS_M)$. An SCC C is called *self-fulfilling* if for every atom (s, F) in C and for every formula $\psi_1 U \psi_2 \in F$ there is an atom (s', F') in C such that $\psi_2 \in F'$. We then say that if there exists a path in $\mathcal{A}(TS_M)$ starting in (s_0, F_0) reaching a self-fulfilling SCC where $s_0 \in S_0$ and $\neg\phi \in F_0$, then $M \not\models \phi$ (a counterexample has been found). Otherwise, if there is no such a path in $\mathcal{A}(TS_M)$, we have shown that $M \models \phi$.

2.3 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 in this section.

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 achieved, 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 8). 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 guard $G(x, t)$ holds for the current variables' evaluation will be executed and possible new successor states *succ* will be generated. Subsequently, a transition will be added to the state space (line 12) and the state *succ* will be adjoined to the queue (line 14) 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 11-18.

Algorithm 1: Consistency Checking

```
1  Queue := {root} ; Visited := {}; Graph := {};
2  while Queue is not empty do
3      if random(1) < α then
4          state := pop_from_front(Queue)                    /* depth-first */
5      else
6          state := pop_from_end(Queue)                      /* breadth-first */
7      end if
8      if error(state) then
9          return counter-example trace in Graph from root to state
10     else
11         for all succ,evt such that state -evt-> succ do        the code to
12             Graph := Graph ∪ {state -evt-> succ};              be optimised
13             if succ ∉ Visited then
14                 push_to_front(succ, Queue);
15                 Visited := Visited ∪ {succ}
16             end if
17         end for
18
19     end if
20 end while
21 return ok
```

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 $read_{\mathbf{G}}(e)$ and $read_{\mathbf{S}}(e)$ 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.

3.1 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, $e_2 \in enabled(s)$ the executions $s \xrightarrow{e_1} s_1 \xrightarrow{e_2} s'$ and $s \xrightarrow{e_2} s_2 \xrightarrow{e_1} s''$ are both 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 ($write(e_1) \cap write(e_2) = \varnothing$).

From the three conditions above one can infer that two events that are syntactically independent cannot disable each other since the effect of executing the one event cannot change the value of each variable in the guard of the other event ((SI 1) and (SI 2)). And, additionally, both events cannot interfere each other as they write different variables ((SI 3)), and each variable written by the one event is not read in the action part of the other event ((SI 1) and (SI 2)). Thus, the definition of syntactic independence ensures independence according to Def. 1.

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.

3.2 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 tautologies (e.g., $1 = 1$) then *test_path* will return *true* if the given sequence of events is possible from some state of M. Accordingly, if Φ is an obvious inconsistency (e.g., $1 = 2$) then *test_path* will return $false$ as there is no state s such that $s \models \Phi$.

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_S(e1) ∩ write(e2) ≠ ∅ ∨ write(e1) ∩ read_S(e2) ≠ ∅) then
5          return true              /* events influence each others' effect */
6      else
7          return
8          ((read_G(e1) ∩ write(e2) ≠ ∅ ∧ test_path(G_e1 ∧ G_e2, · --e2--> ·, ¬G_e1))
9          ∨ (write(e1) ∩ read_G(e2) ≠ ∅ ∧ test_path(G_e2 ∧ G_e1, · --e1--> ·, ¬G_e2))
10     end if
11 end procedure
```

The procedure *dependent* presents a refined strategy for determining the dependency between two events. On syntactical level we would say that two events are dependent if their *write* sets are not disjoint or if the *write* set of the one event has variables in common with the *read* set of the other one. As we already have seen (in Example 1), the syntactic analysis is not precise enough to exactly determine how two events are related to each other. Therefore, in lines 8-9 in Algorithm 2 we further check if the events can disable each other by means of

the *test_path* procedure. In order to test whether two events are independent, we need to check the two independence conditions *enabledness* and *commutativity*. Obviously, the commutativity conditions for two events may not be satisfied if both events have write variables in common (line 2) or if at least one of the events may write a variable read in the actions part of the another event (line 4). If the tests in line 2 and in line 4 do not pass, then we just need to examine if some of the events can disable the other one in order to show whether they are independent (the *enabledness* condition).

Once we have entered the **else** branch, we test the enabledness condition. The enabledness condition is tested by the two disjunction arguments in lines 8 and 9. 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 $read_{\mathbf{G}}(e_1) \cap write(e_2) \neq \varnothing$) 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.

3.3 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 labelled 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 $write(e_1) \cap read_{\mathbf{G}}(e_2) = \varnothing$, 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 $test\_path(G_{e_1} \wedge \neg G_{e_2}, \cdot \xrightarrow{e_1} \cdot, G_{e_2})$.

Algorithm 3: Determining Enabling Relations

```
1 procedure boolean can_enable(e1, e2)
2     if (e1 = e2) ∨ (write(e1) ∩ read_G(e2) = ∅) then
3         return false
4     else
5         return test_path(G_e1 ∧ ¬G_e2, · -e1-> ·, G_e2)
6     end if
7 end procedure
```

4 Partial Order Reduction Algorithm based on Ample Sets

In this section we introduce the ample set theory 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.

4.1 The Ample Set Requirements

An ample set is a subset of the enabled events, chose for expansion. All events not in the ample set will be ignored (leading to a state space reduction).

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 contains an enabled event e, then there exists a state s in C such that $e \in ample(s)$.

4.2 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).

Theorem 1 (Sufficient Local Criteria for (A 2)). *Let s be a state 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.*

Proof. By contraposition. Let conditions (A 2.1) and (A 2.2) hold for $ample(s)$. Assume that (A 2) does not hold. Then there exists an execution fragment

$$\sigma = s \xrightarrow{\beta_1} s_1 \xrightarrow{\beta_2} \ldots \xrightarrow{\beta_n} s_n \xrightarrow{\beta_{n+1}} \ldots$$

where $\beta_1, \ldots, \beta_n \notin ample(s)$ and β_{n+1} is dependent on $ample(s)$.

Since (A 2.1) holds for $ample(s)$ it follows that any $\alpha \in ample(s)$ is enabled in s_1 (β_1 is independent from $ample(s)$ by (A 2.1)). Furthermore, β_2 cannot be dependent on $ample(s)$ since $\beta_2 \notin ample(s)$ and β_1 cannot enable dependent action by (A 2.2). Thus, it follows that β_2 is independent from $ample(s)$. Continuing applying the same procedure inductively for the residual states s_i and events β_i with $2 \leq i \leq n$ on $ample(s)$ one can conclude that β_{n+1} is not dependent on $ample(s)$. The independence of β_{n+1} to $ample(s)$ contradicts the assumption that there exists a fragment execution σ for which (A 2) is violated. Hence, (A 2) is satisfied by any $ample(s)$ fulfilling conditions (A 2.1) and (A 2.2).□

4.3 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 4 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 4 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 4: Computation of $ample(s)$

Data: $EnableGraph_M$, $Dependent_M$
Input: The set of events T enabled in the currently processed state s ($T = enabled(s)$)
Output: A subset of T satisfying (A 1) - (A 3)

```
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. $\gamma_1, \ldots, \gamma_n \notin \mathit{ample}(s)$). This we do by means of looking for paths in $\mathit{EnableGraph}_M$ 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 $\mathit{EnableGraph}_M$ 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 4 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).

In the following we will present our proof of correctness for computing an ample set satisfying condition (A 1) to (A 3) by means of Algorithm 4. The main statement, the procedure *ComputeAmpleSet* returns a set satisfying (A 1) to (A 3), will be given by means of a theorem. For proving the theorem we will use three lemmas which show that the result returned by the procedure *ComputeAmpleSet* accordingly satisfies the ample set conditions (A 1), (A 2.1), and (A 2.2). The stutter condition (A 3) will not be handled specifically for proving the theorem as we assume at this point that the procedure for checking whether S is a stutter set is correct.

Lemma 1. *Let A be a set computed by means of the* `ComputeAmpleSet` *procedure for some set of events T. Then, it is satisfied that A is an empty set if and only if T is an empty set.*

Proof. Let $T = \varnothing$. In this case the outer **foreach**-loop will not be entered and the argument T of the procedure *ComputeAmpleSet* will be returned as a result (line 16). This infers that A is also an empty set.

Let $T \neq \varnothing$. Then, there are two ways of computing the set A. The first one is when for no one of the events $\alpha \in T$ a set S can be computed that is returned as a result in line 13. In this case *ComputeAmpleSet* will return the set T, which by assumption is a non-empty set. The second possibility for computing A by means of *ComputeAmpleSet* is when there exists an event $\alpha \in T$ such that a set S is determined which is returned in line 13. In that case S has at least one element, the event α, since $\alpha \in \mathit{reachable}(\alpha, G)$. Note that the currently computed set S is returned as a result if for all $\beta \in I$ the **if**-condition in line 7 does not hold and all events in S should be stutter events.□

Lemma 1 states that $\mathit{ComputeAmpleSet}(T) = \varnothing$ if and only if T is an empty set. Hence, (A 1) is satisfied by the procedure *ComputeAmpleSet* in Algorithm 4. As next, we want to show that each set A computed by the procedure

ComputeAmpleSet fulfils condition (A 2). This statement we will show by showing that each A satisfies both local dependency conditions (A 2.1) and (A 2.2). We already have shown in Theorem 1 that (A 2.1) and (A 2.2) are sufficient criteria for (A 2). Thus, proving that A satisfies (A 2.1) and (A 2.2) will infer that Algorithm 4 also satisfies the Dependency condition (A 2).

Lemma 2. *Let A be a set of events computed by means of the procedure ComputeAmpleSet for some set of events T. Then, any $\beta \in T \setminus A$ is independent of A.*

Proof. First, if the procedure *ComputeAmpleSet* returns T as a result, it is clear that A $(= T)$ satisfies condition (A 2.1). If $A \subsetneq T$, then the returned subset A is the set computed by the procedure *DependencySet* for some event α in T. Thus, to show that all events $\beta \in T \setminus A$ are independent of A, it suffices to show the following claim:

> Let S be a set of events computed by means of the procedure *DependencySet* in regard to a set of
> events T and an event $\alpha \in T$. Then, any $\beta \in T \setminus S$ is independent of S.

We prove the claim by contraposition. Assume there is an event $\gamma \in T \setminus S$ such that γ is dependent on S. This means γ is dependent on some event β that is an element of S. Recall that the set G in procedure *DependencySet* is regarded as a directed graph where the vertices are the elements of T. The procedure spans a directed graph G by adding an edge $\beta \mapsto \gamma$ for each tuple of events (β, γ) in $Dependent_M$ for which both events β and γ are lying in T (see lines 20-22 in Algorithm 4). Note that $\beta \mapsto \beta$ for each $(\beta, \beta) \in T \times T$ are also added to G as the relation $Dependent_M$ is reflexive.

Remark that $reachable(\alpha, G)$ denotes the set S that is returned in line 4 in procedure *ComputeAmpleSet*. By assumption, there is an event $\gamma \in T \setminus reachable(\alpha, G)$ such that there exists an event $\beta \in reachable(\alpha, G)$ with $(\beta, \gamma) \in Dependent_M$. As $\beta \in reachable(\alpha, G)$ there is a path $\alpha \mapsto \alpha_1 \mapsto \ldots \mapsto \alpha_n \mapsto \beta$ in G where $(\alpha, \alpha_1), (\alpha_n, \beta) \in Dependent_M$ and $(\alpha_i, \alpha_{i+1}) \in Dependent_M$ for all $1 \leq i \leq n-1$. The **foreach**-block in procedure *DependencySet* guarantees that each pair $(\alpha', \beta) \in Dependent_M$ is added as an edge to G if α' and β are elements of T. Since γ and β are elements of T, and β is dependent on γ (by assumption) it follows that there is also an edge $\beta \mapsto \gamma$ in G. This implies that γ is also reachable from α which is a contradiction to the assumption $\gamma \in T \setminus reachable(\alpha, G)$. □

Since for each set computed by the *DependencySet* procedure Lemma 2 is satisfied, we can deduce that the local dependency condition (A 2.1) is fulfilled for each set returned by the procedure *ComputeAmpleSet*. It remains to show that the sets computed by Algorithm 4 fulfil also condition (A 2.2). This we will demonstrate by means of the following lemma.

Lemma 3. *Let A be an ample set computed by the procedure ComputeAmpleSet in Algorithm 4 at some state s and let T denotes the set enabled(s). For each $\beta \in T \setminus A$ and for all $n \geq 0$ there is no execution fragment*

$$\sigma = s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_n} s_{n+1} \xrightarrow{\gamma} s'$$

in TS_M such that $\gamma_1, \ldots, \gamma_n, \gamma \notin S$ and γ depends on A.

Proof. By Lemma 2 we know that A fulfils the local dependency condition (A 2.1). In other words, for each $\beta \in T \setminus A$ we know that β is independent from all events in A. Without loss of generality we assume that $A \subsetneq T$. Let β be some event from $T \setminus A$. As next, we show that for all $n \geq 0$ the execution fragment

$$\sigma = s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_n} s_{n+1} \xrightarrow{\gamma} s'$$

with $\gamma_1, \ldots, \gamma_n, \gamma \notin S$ and γ depends on A does not exists in TS_M.

We carry out the proof of the claim by induction on n. In the following we will denote by σ^i, where $i \geq 0$, the execution fragment $s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_i} s_{i+1} \xrightarrow{\gamma} s'$, and by $Paths(EnableGraph_M)$ all paths in the enable graph $EnableGraph_M$ of the currently checked machine M.

Basis Step: Let $n = 0$. Suppose the execution fragment $\sigma^0 = s \xrightarrow{\beta} s_1 \xrightarrow{\gamma} s'$ where $\beta, \gamma \notin A$ and γ depends on A exists in TS_M. Then, there are two cases to consider.

(1) $b \mapsto \gamma \notin Paths(EnableGraph_M)$: If β cannot enable γ, then γ must be enabled in s. By assumption $\gamma \notin A$. We also know that A satisfies condition (A 2.1) and thus by Lemma 2 γ is independent from A. This, however, is a contradiction to the assumption that γ depends on A. It follows that σ^0 does not exist for this case.

(2) $b \mapsto \gamma \in Paths(EnableGraph_M)$: If there is a path $\beta \mapsto \gamma$ in $EnableGraph_M$ such that $\beta, \gamma \notin A$ and γ depends on A, then the set A will be denied as an ample set in procedure *ComputeAmpleSet* as the **if**-condition in line 7 holds for this case. Since A is returned as an ample set by *ComputeAmpleSet* we can infer that σ^0 with $\beta, \gamma \notin A$ and γ dependent on A does not exists in TS_M for this case.

Inductive Step: Assume, for $n = k$, that there is no execution fragment $s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_k} s_{k+1} \xrightarrow{\gamma} s'$ in TS_M such that $\gamma_1, \ldots, \gamma_k, \gamma \notin A$ and γ depends on A. We show that there is no execution

$$\sigma^{k+1} = s \xrightarrow{\beta} s_1 \xrightarrow{\gamma_1} s_2 \xrightarrow{\gamma_2} \ldots \xrightarrow{\gamma_k} s_{k+1} \xrightarrow{\gamma_{k+1}} s_{k+2} \xrightarrow{\gamma} s'$$

in TS_M such that $\gamma_1, \ldots, \gamma_{k+1}, \gamma \notin A$ and γ is dependent on A.

Suppose again that there is such an execution fragment σ^{k+1} in TS_M. Then, we need to consider again two cases.

(1) $\gamma_{k+1} \mapsto \gamma \notin Paths(EnableGraph_M)$: The absence of such an edge $\gamma_{k+1} \rightarrow \gamma$ in $EnableGraph_M$ infers that γ cannot become enabled after the execution of the event γ_{k+1} and as a consequence we deduce that γ must already be enabled in s_{k+1}. This, however, contradicts with the induction hypothesis for σ^k. Hence, in this case there is no sequence σ^{k+1} such that $\gamma_1, \ldots, \gamma_{k+1}, \gamma \notin A$ and γ is dependent on A.

(2) $\gamma_{k+1} \mapsto \gamma \in Paths(EnableGraph_M)$: In the following we intend to construct an enabling path $\pi_{k+1} \in Paths(EnablingGraph_m)$ from the execution fragment σ^{k+1} by means of the following procedure: Beginning with $\pi_0 = \gamma_{k+1} \mapsto$

γ and starting with γ_{k+1} we examine whether γ_k may enable γ_{k+1}. If $\gamma_k \mapsto \gamma_{k+1} \in Paths(EnableGraph_M)$, then we create a new enabling path as follows $\pi_1 = \gamma_k \mapsto \pi_0$. Otherwise, if γ_k cannot enable γ_{k+1}, we set π_1 to be equal to π_0. Continuing this procedure inductively until s is reached we have constructed as a result an enabling path π_{k+1} that is an element of $Paths(EnableGraph_M)$. We consider two cases for the enabling path π_{k+1}.

(2.1) In the first the enabling path starts with β: $\pi_{k+1} = \beta \mapsto \hat{\gamma}_1 \mapsto \ldots \mapsto \hat{\gamma}_j \mapsto \gamma_{k+1} \mapsto \gamma$, where each $\hat{\gamma}_i$ corresponds to some event γ_l in σ^{k+1} with $1 \leq i \leq j$ and $1 \leq l \leq k$. Note that $j \leq k$ as there may be events in σ^{k+1} that cannot be enabled by its preceding events in the execution fragment σ^{k+1}. The path π_{k+1} is an enabling path in $EnableGraph_M$, which means that in this case the **if**-condition in line 7 in Algorithm 4 holds and as a consequence A will be refused as an ample set in procedure *ComputeAmpleSet*. Since A is returned as a result by *ComputeAmpleSet* it follows that there is no execution fragment σ^{k+1} such that $\gamma_1, \ldots, \gamma_{k+1}, \gamma \notin A$ and γ is dependent on A.

(2.2) The second case we need to observe is when $\pi_{k+1} = \hat{\gamma}_1 \mapsto \ldots \mapsto \hat{\gamma}_j \mapsto \gamma_{k+1} \mapsto \gamma$, where each $\hat{\gamma}_i$ corresponds to some event γ_l in σ^{k+1} with $1 \leq i \leq j$ and $1 \leq l \leq k$. In this case we know that $\hat{\gamma}_1$ is enabled in state s since all preceding events of $\hat{\gamma}_1$ in σ^{k+1} may not enable $\hat{\gamma}_1$. By assumption of σ^{k+1} we know that $\hat{\gamma}_1 \notin A$. Thus, it follows that there exists a path $\hat{\gamma}_1 \mapsto \ldots \mapsto \hat{\gamma}_j \mapsto \gamma_{k+1} \mapsto \gamma$ in $EnableGraph_M$ such that $\hat{\gamma}_1, \ldots, \hat{\gamma}_j, \gamma_{k+1}, \gamma \notin A$ and γ dependent on A for some event $\hat{\gamma}_1 \in T \setminus A$. This, however, contradicts with the choice of the set A since no such a set can be returned by the procedure *ComputeAmpleSet* when the variable b is set to $false$.

Thus, we can conclude from the induction proof that for β and for all $n \geq 0$ there is no execution fragment σ^n in TS_M such that $\gamma_1, \ldots, \gamma_n, \gamma \notin A$ and γ is dependent on A. It is readily to see that the proposition is fulfilled for all $\beta \in T \setminus A$. □

Now using the results from Lemma 1, 2, and 3 we can state the following theorem.

Theorem 2. *Every set A computed by means of the procedure ComputeAmpleSet in Algorithm 4 satisfies the ample set conditions (A 1) to (A 3).*

The way of computing an ample set in Algorithm 4 tells us that more than one ample set can exist per state. Randomly choosing an event from T for building an ample set for the particular state s is equivalent to computing all possible ample sets for s and then randomly choosing one of them. Another heuristic for choosing which subset of enabled events in the currently expanded state to be chosen would be always to choose the ample set with the least number of elements in order to achieve maximal state space reduction. Always choosing the ample set with the least number of events is, however, not a premise for achieving maximal state space reduction as discussed in [26]. Therefore, we believe that randomly choosing an ample set should result in an approximatively good state space reduction. Note also that model checking with partial order reduction using randomised choosing of an ample set in each state sometimes can result in

checking different number of states every time the model checker has been run on the same model.

4.4 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 [26].

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 sometimes unnecessarily. 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], [8]. Furthermore, the implementation of condition (A 4) in this way is also a design decision as we want easily to use the same reduction algorithm also for LTL model checking.

4.5 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 5 with the currently processed state s as argument.

Algorithm 5 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 5 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 5: 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
14 end procedure
```

```
15 procedure execute_event(s, evt)
16     compute successor state s' by executing evt from s;
17     Graph := Graph ∪ {s -evt-> s'};
18     if s' ∉ Visited then
19         push_to_front(s', Queue);
20         Visited := Visited ∪ {s'}
21     end if
22     return s'
23 end procedure
```

4.6 Adapting the Reduction Algorithm for the ProB LTL$^{[e]}$ Model Checker

Since ProB also supports LTL$^{[e]}$ model checking for Event-B (as well as B, Z, CSP, and CSP‖B), we are also interested in elaborating the reduction algorithm for the ProB LTL$^{[e]}$ model checker [22] for checking temporal properties on models written in B and Event-B. In this subsection we discuss the adaptation of the reduction algorithm above for reducing the state space in the process of model checking LTL$_{-X}$ formulae. In particular, we consider which ample set conditions should be regarded more carefully in order to adapt the reduction algorithm to also effectively check LTL$_{-X}$ formulae by means of the LTL$^{[e]}$ model checker algorithm in ProB.

LTL$^{[e]}$ Model Checking in ProB The ProB LTL$^{[e]}$ model checker follows the tableau approach from [19] and can check properties specified in LTL$^{[e]}$. The algorithm presented in [22] additionally allows checking of properties stated in Past-LTL$^{[e]}$ and can cope with deadlock states as well as partially explored state spaces.

Given a model M and an LTL$^{[e]}$ formula ϕ, the ProB LTL$^{[e]}$ model checker checks $M \vDash \phi$ by searching for bad paths satisfying $\neg\phi$, i.e. strongly connected components (SCCs) that can be reached from some initial state of M and that are self-fulfilling with respect to $\neg\phi$. If such a path has been found, it will be reported as a counterexample (failure behaviour of M) for ϕ. Otherwise, if no path satisfying $\neg\phi$ has been discovered, we have proven that $M \vDash \phi$. The search for SCCs in the ProB LTL$^{[e]}$ model checker is based on the Tarjan's algorithm [24].

We can distinguish two approaches of checking an LTL$^{[e]}$ formula ϕ on an Event-B model M with the LTL$^{[e]}$ model checker:

- *Static approach:* exploring the entire state space of M and then checking ϕ by means of the tableau search algorithm, or
- *Dynamic approach:* expanding the state space of M while applying the tableau search algorithm.

Both approaches have their advantages and disadvantages. On the one hand, using the static approach one can benefit from the fact that the state space of the model M has already been explored fully and thus various LTL$^{[e]}$ formulae may be checked without re-exploring the state space every time. On the other hand, by dynamically checking models one may profit from the fact that the state space may not be required to be fully explored when a bad path in the yet partially explored state space is found violating the checked property. The dynamical LTL$^{[e]}$ model checking can be very effective especially when the checked model has a very large state space. Checking LTL properties statically and dynamically by means of the definitions above are also known as off-line and on-the-fly LTL model checking in the literature [20], [11].

The tableau algorithm from [22] is implemented in C using a callback mechanism for evaluating the atomic propositions and the outgoing transitions in SICStus Prolog. While constructing the directed graph $\mathcal{A}(TS_M)$, the tableau algorithm expands the state space of M using the same procedure for expanding each state as the consistency checking algorithm do (see Algorithm 1). The reduction presented in this section is based on computing just a subset $ample(s)$ of the set of enabled events $enabled(s)$ in each state. Intuitively, what has changed is just the way of expanding each state in the state space of the model being verified. Since the LTL$^{[e]}$ model checker algorithm uses the same procedure to expand each state we will use this fact to adapt the reduction algorithms for using these for reduced search in LTL$^{[e]}$ model checking in ProB. Basically, we need to consider for which ample set conditions we have to adapt the algorithms 4 and 5 in order to make the reduction of the search graph $\mathcal{A}(TS_M)$ sound and effective for off-line and on-the-fly LTL$^{[e]}$ model checking.

LTL$^{[e]}$ Formulae that are Invariant under Partial Order Reduction Before discussing how the reduction algorithm can be adapted for LTL$^{[e]}$ model checking with partial order reduction, we need to determine first which set of LTL$^{[e]}$ formulae is invariant under reduction by partial order reduction. Formally, we study for which subset C of LTL$^{[e]}$ formulae the equivalence

$$\forall \phi \in C \cdot (TS_M \models \phi \Leftrightarrow \hat{TS}_M \models \phi) \tag{1}$$

is satisfied, where $\hat{TS}_M$ denotes the reduced transition system of TS_M using the ample set theory. The equivalence is not satisfied for formulae with the next-operator X since stutter equivalence does not preserve the truth values of such formulae [21]. The extended version of LTL (LTL$^{[e]}$) defines a new operator $[\cdot]$ that allows one to make assertions about the executions of events along the paths in TS_M. For example, the formula "$[e] \Rightarrow F\ \{x = 2\}$" encodes the property "if e is executed at the current state the variable x will eventually be equal to 2". What we need to examine is whether formulae with the *executed*-operator $[\cdot]$ violate the equivalence (1).

Consider, for example, the LTL$^{[e]}$ formula "$\phi = ([e_1] \Rightarrow F\ \{x = 2\})$" and the transition system TS_M on the left side of Figure 1. Obviously, $TS_M \not\models \phi$ as there is a path $\pi = s_0 \overset{e_1}{\rightarrow} s_1 \overset{e_2}{\rightarrow} s_3 \overset{e_3}{\rightarrow} s_3 \ldots$ in TS_M that violates the formula. The ample set approach will reduce the state space TS_M as shown on the right side of Figure 1. The reduction algorithm will choose $ample(s_0) = \{e_2\}$ because visibly e_1 modifies the variable x which is used in the atomic proposition $\{x = 2\}$. This means that e_1 will be considered as a non-stutter event as it may potentially change the value of the atomic proposition in ϕ. The reduced transition system $\hat{TS}_M$ at the same time satisfies ϕ as the only path in $\hat{TS}_M$ is $s_0 \overset{e_2}{\rightarrow} s_2 \overset{e_1}{\rightarrow} s_3 \overset{e_3}{\rightarrow} \ldots$ which obviously does not violate ϕ.

The example shows that, in general, LTL$^{[e]}$ formulae with the execute-operator do not fulfil the equivalence $TS_M \models \phi \Leftrightarrow \hat{TS}_M \models \phi$. Thus, the set of LTL$^{[e]}$ formulae that is invariant under partial order reduction is the set of all LTL$^{[e]}$ formulae without the next-operator X and without the execute-operator $[\cdot]$. This subset of formulae we will denote by LTL$_{-X}$.

The Static Approach The static approach of LTL$^{[e]}$ model checking in ProB may be explained as a two-phase process: expanding the state space of the model M and in the subsequent step checking a set of LTL$^{[e]}$ formulae by means of the tableau approach. The main advantage of the approach is that various formulae may be checked once the entire state space of the model has been expanded. On the contrary, performing LTL$^{[e]}$ model checking in this way demands the exploration of the entire state space which in many cases may be very large.

Applying partial order reduction for the static approach has some subtle differences from the static approach without reduction. The static approach with reduction will be completed in two steps: constructing the reduced state space and then using the LTL$^{[e]}$ model checking algorithm to check the LTL$_{-X}$ formula in the reduced state space. However, for each new formula ϕ the reduced state space in regard to ϕ should be constructed. This requirement is necessary

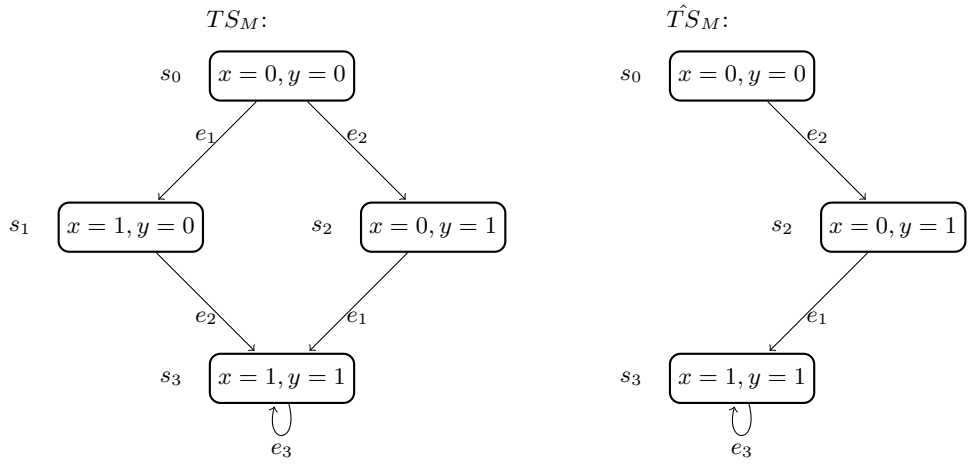

Fig. 1. Ample-set reduction does not preserve the truth value for the formula $[e] \Rightarrow F\ \{x = 2\}$.

because of the Stutter condition (A 3) of the ample set method. For a given model M, the set of stutter events in M will be determined in regard to the formula being checked. Thus, every time a new formula is checked the set of stutter events changes and as a consequence the corresponding reduced state space should be constructed.

To adapt the reduction algorithm for the static approach we construct the state space by using a graph traversal algorithm that uses the *compute_ample_transitions* procedure in Algorithm 5 for expanding each reachable state. Basically, we can use the consistency checking algorithm (Algorithm 1) to expand the reduced state space of an Event-B model in case that we modify it as follows: removing the **if**-statement in lines 8-9 and replacing the pseudo code in lines 11-18 by the call *compute_ample_transitions*(*state*), where *state* is the currently processed state. The exploration strategy of the modified version of the consistency checking algorithm is not relevant for the computation of the ample sets since the implementation of the Strong Cycle condition (A 4') in Algorithm 5 preserve the soundness of computing ample sets by different exploration strategies.

The Dynamic Approach In contrast to the static approach, in the dynamic approach the transition system TS_M of the model is created while constructing the tableau graph $\mathcal{A}(TS_M)$ for the negation of the checked LTL$^{[e]}$ property. One can consider to use the *compute_ample_transitions* procedure from Algorithm 5 for the expansion of the states of the reduced transition system $\hat{TS}_M$. However, we should look closely at the way the Cycle condition (A 4') can be ensured in

the dynamic approach. In the first place, a cycle in the transition system TS_M does not necessarily correspond to a cycle in the search graph $\mathcal{A}(TS_M)$. This means that having a cycle

$$\pi = s_i \rightarrow s_{i+1} \rightarrow \ldots \rightarrow s_{i+k} \rightarrow s_i$$

in TS_M does not imply that we have a path in $\mathcal{A}(TS_M)$ of the form

$$\rho_\pi = (s_i, F_i) \rightarrow (s_{i+1}, F_{i+1}) \rightarrow \ldots \rightarrow (s_{i+k}, F_{i+k}) \rightarrow (s_i, F_{i+k+1})$$

with $F_i = F_{i+k+1}$. Moreover, the path ρ_π may not exist in $\mathcal{A}(TS_M)$ since the condition for existing of an edge $(s_j, F_j) \rightarrow (s_{j+1}, F_{j+1})$ in $\mathcal{A}(TS_M)$ additionally requires that for every formula $X\ \psi \in F_j$ the sub-formula ψ is an element of F_{j+1}.

Additionally, the LTL$^{[e]}$ model checker uses the Tarjan algorithm for finding self-fulfilling SCCs. The Tarjan algorithm is based on a depth-first search for finding SCCs. One can profit from the depth-first search using the fact that an atom having an outgoing edge to an atom on the search stack is closing a cycle in $\mathcal{A}(TS_M)$. In this way, we can identify the cycles in the reduced tableau graph $\mathcal{A}(\hat{TS}_M)$, and by changing the implementation accordingly, we can check the Strong cycle condition (A 4') without checking more than the sufficient number of states. Recall that the procedure for checking (A 4') in Algorithm 5 is necessary but not sufficient as it may cause that states are unnecessarily fully expanded.

To make use of the observations above, one has to revise the way the Strong Cycle condition (A 4') should be checked for the dynamic approach of LTL$^{[e]}$ model checking. The idea is to expand fully a state s of the reduced transition system $\hat{TS}_M$ if it is certain that there is a back transition from an atom (s, F) closing a cycle in $\mathcal{A}(\hat{TS}_M)$. Therefore, we replace the Strong cycle condition (A 4') by the following condition:

(D 4) Dynamic Cycle Condition

Any cycle in the reduced search graph $\mathcal{A}(\hat{TS}_M)$ has at least one atom (s, F) such that state s is fully expanded in $\hat{TS}_M$, i.e. $ample(s) = enabled(s)$.

The next step would be to incorporate the ample set reduction method in the LTL$^{[e]}$ model checker of ProB. The procedure for computing the ample sets for the LTL$^{[e]}$ model checker will be the same as for the consistency checking algorithm up to the satisfaction of the Cycle condition (A 4). For ensuring (A 4) we will use condition (D 4) instead of (A 4'). Accordingly, the realisation of (D 4) should take place during the construction of the tableau graph $\mathcal{A}(T\hat{S}_M)$. From the technical point of view, this means that we should extend the tableau algorithm in ProB, which is implemented in C, in regard to checking (D 4). Apart from that, the procedure for expanding some state s will be changed to execute just the events from $ample(s) \subseteq enabled(s)$, where $ample(s)$ is the set of events computed with respect to the ample set conditions (A 1) through (A 3). A state s in $\hat{TS}_M$ will be fully expanded if there is an atom (s, F) in $\mathcal{A}(\hat{TS}_M)$ such that an edge from (s, F) exists going back to an atom on the search stack.

5 Discussion and Evaluation

5.1 Discussion

In Section 4, we presented the background of the ample set theory and our implementation of partial order reduction (Algorithms 4 and 5). Our algorithm reduces the original state space of an Event-B machine M by using the dependence 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 decreasing 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) [14, 26]. 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.

5.2 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

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

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
Fact v2	MC	112,185	381,510	-	208.150
	MC+POR	112,185	381,510	0.589	230.434
	MC-NoINV	112,185	381,510	-	197.181
	MC-NoINV+POR	27,628	62,950	0.476	50.051
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:

MC: Model checking by using the standard consistency checking algorithm.

MC+POR: Model checking by combining the standard consistency checking algorithm with the partial order reduction algorithm.

MC-NoINV: Model checking by using the standard consistency checking algorithm without invariant violations checking.

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

The consistency checking algorithm and the partial order reduction algorithm are respectively Algorithm 1 and Algorithm 5. 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 to the executions of all other events. The worst case, when no reductions of the state space are gained, is represented by checking *Fact v2*, and *BPEL v6* with MC+POR. *Fact v2* is an Event-B model of a simple parallel algorithm for integer factorisation. The factorisation algorithm's model was re-created from [12] for three computational slave processes searching for a factor of 53. In *Fact v2* the guard of the event *newround* was weakened. *Phil* [9] 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 two cases (*Fact v2* and *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 of independence, the coupling between the events is so tight that no significant reductions can be gained.

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

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 [25] and the permutation flooding method [17]. 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 [18] 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 [15], have been also suggested as optimisation methods for the standard ProB model checker.

The notion of the enable graph for Event-B models has been first introduced in [7]. In this work enable graphs are used to encode the information about independence[3] and dependence of events by means of enabling predicates. In addition, the authors of this work suggest a method for optimising model checking by skipping the evaluation of the predicates in some states by means of evaluating the enabling predicates of the enable graph. Additionally, an algorithm is proposed for constructing flow graphs of Event-B models as well as possible applications of flow graphs are discussed.

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

[3] The definition of independence between events in [7] is different from the definition of independence in respect to partial order reduction. In [7] two events are considered to be independent if each of the events cannot influence the guard of the other one.

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

Next, we intend to integrate the reduction algorithm also in the ProB LTL$^{[e]}$ model checker. In this work we discussed how to elaborate the reduction algorithm for the LTL model checker in ProB. We considered two approaches (static and dynamic approach) for providing LTL model checking in ProB using partial order reduction. We plan to implement both approaches, as well as to make a thorough evaluation.

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. J. Bendisposto and M. Leuschel. Automatic flow analysis for Event-B. In D. Giannakopoulou and F. Orejas, editors, *Proceedings of Fundamental Approaches to Software Engineering (FASE) 2011*, volume 6603 of *LNCS*, pages 50–64. Springer, 2011.
8. D. Bosnacki, S. Leue, and A. Lluch-Lafuente. Partial-Order Reduction for General State Exploring Algorithms. *STTT*, 11(1):39–51, 2009.
9. 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.
10. 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.
11. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.
12. F. Degerlund. Scheduling Performance of Compute-Intensive Concurrent Code Developed Using Event-B. *TUCS Technical Reports 1051*, pages 1–20, 2012.
13. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

14. 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.
15. 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.
16. M. Leuschel and M. Butler. ProB: An Automated Analysis Toolset for the B Method. *STTT*, 10(2):185–203, 2008.
17. 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.
18. 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.
19. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 97–107, New York, NY, USA, 1985. ACM.
20. D. Peled. Combining partial order reductions with on-the-fly model-checking. pages 377–390. Springer-Verlag, 1994.
21. D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Inf. Process. Lett.*, 63(5):243–246, 1997.
22. 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.
23. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
24. R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.
25. E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry Reduced Model Checking for B. In *Proceedings TASE 2007*, pages 25–34. IEEE, 2007.
26. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *Applications and Theory of Petri Nets*, pages 491–515, 1989.
27. A. Valmari. A stubborn attack on state explosion. In *CAV*, pages 156–165, 1990.