Under consideration for publication in Formal Aspects of Computing

Optimising the ProB Model Checker for B using Partial Order Reduction

Ivaylo Dobrikov[1] and Michael Leuschel[1]

[1]University of Düsseldorf, Germany

Abstract. Partial order reduction has been very successful at combatting the state explosion problem [BK08], [CEGP99] 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 Event-B, with its much more fine-grained events and thus increased potential for event-independence and partial order reduction. In this work, we provide a detailed description of a partial order reduction for explicit state model checking in ProB. The technique is evaluated on a variety of models. The implementation of the method is discussed, which is based on new constraint-based analyses. Further, we give a comprehensive description for elaborating the implementation into the LTL model checker of ProB for checking LTL$_{-X}$ formulae.

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

1. Introduction

ProB [LB08] 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. ProB consists also of an LTL model checker [PL10] that enables one to check properties expressed in LTL$^{[e]}$, an extended version of LTL. 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 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 usually requires to restrict the domains of the variables of the machine. 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 and 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 omitted. There are several methods of partial order reduction [CGMP99], [G96], [V89a] 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 [BK08], [CGMP99], [CEGP99].

Correspondence and offprint requests to: Ivaylo Dobrikov, Institut für Informatik, Heinrich-Heine Universität Düsseldorf, Universitätstr. 1, D-40225 Düsseldorf, e-mail: dobrikov@cs.uni-duesseldorf.de

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 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 [LB08] for explicit state model checking of models written in B [A96] and Event-B [A10]. The implementation is evaluated on different case study models and thoroughly discussed, as well as a proof of correctness is provided. Additionally, we give 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. Indeed, while partial order reduction has proven very effective for lower level formalisms such as Promela, it has thus far made little impact on high-level modelling languages such as B, Z or TLA^+ (some recent exceptions are [RMQ10] for TLA^+ and [ZSS+14] in the context of the PAT model checker). Our intuition is that the more fine-grained nature of events in Event-B should substantially increase the potential for partial order reduction.

This work was initially published as an SEFM conference paper in 2014 [DL14]. Some changes were made to this version, as well as additional material was included. The most significant change is the corrected version of Algorithm 4 in [DL14]. A proof of correctness to the corrected version of Algorithm 4 is provided in Section 4. Additionally to the contributions in the conference article, more examples have been added for more clear and convenient presentation of the work in [DL14], as well as new definitions and remarks were included. Further, we discuss ways of adapting our reduction algorithm into the ProB $\text{LTL}^{[e]}$ model checker [PL10] for efficient checking of LTL_{-X} properties of B and Event-B models in ProB. A comprehensive outlook of the logical formalism $\text{LTL}^{[e]}$ is given in the Preliminaries section. The ideas of applying partial order reduction for checking LTL_{-X} properties in ProB for B are presented in Section 5. We also elaborated on the Discussion section (Section 6.1) and added more related work in terms of existing implementations of partial order reduction in other prominent model checkers (Section 7.2).

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 that will be used in the later sections. In Section 3, we discuss and define formally relations between events that are relevant for this work. Section 4 presents the ample set method and our reduction algorithm accompanied with a proof of correctness. Additionally, two different approaches are presented and discussed for adapting the reduction algorithm for $\text{LTL}^{[e]}$ model checking in ProB in Section 5. The evaluation and the discussion of the implementation are given in Section 6. The related work is outlined in Section 7. 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 generally described as shown in Fig.1.

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 typecast 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 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
```

Fig. 1. A General Event Structure.

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 which later may be read in the guard of e_2. On the other hand, events that do not affect one other and do not interfere each other 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 event relations.

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 also denoted as a *transition system* defined as a 6-tuple

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

where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $Events_M$ a set of events of M, $R \subseteq S \times Events_M \times S$ a set of transitions, AP the set of atomic propositions, and $L : S \to 2^{AP}$ a state-labelling function. A transition $(s, e, s') \in R$ will often be written as follows: $s \xrightarrow{e} s'$. The state-labelling function assigns to each state s a set of atomic propositions $L(s)$, $L(s)$ comprises all atomic propositions that hold in s. For a given Event-B machine M, the set of atomic propositions are first-order logic formulae that are built from B predicates over the variables and constants of M.

When we talk about an enabled event in a particular state s, we mean an event the guards of which 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. That is, for a given 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.

By means of the temporal logic LTL [P77] one can make assertions about the temporal behaviours of a system. In [PL10] an extension of LTL, denoted by $\text{LTL}^{[e]}$, was introduced allowing to state propositions also on transitions. A similar event extension of LTL was also introduced in [CCO+04], where the extension is denoted as State/Event-LTL and the definition there is limited to infinite paths.

For a finite set of atomic propositions AP and a finite set of transition propositions TP, an $\text{LTL}^{[e]}$ formula is formed inductively as follows:

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

In the context of Event-B, an atomic proposition is a first-order logic formula that is built from B predicates over the constants and variables of the Event-B machine intended to be checked. For example, if we check the $\text{LTL}^{[e]}$ formula $\phi = (x > 1) \, \mathcal{U} \, (y = 1 \wedge z > x)$ in the context of Event-B, then the set of atomic propositions AP_ϕ with respect to ϕ is equal to $\{x > 1, y = 1, z > x\}$, where $x > 1$, $y = 1$, and $z > x$ are the atomic propositions of ϕ. Later, we will concentrate on certain class of $\text{LTL}^{[e]}$ formulae, which we will denote by LTL_{-X}. The class LTL_{-X} consists of all $\text{LTL}^{[e]}$ formulae without the $[\cdot]$ and X operators.

An $\text{LTL}^{[e]}$ 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:

- $\pi \models true$
- $\pi \models a \Leftrightarrow \pi = s_0 \ldots$ and $a \in L(s_0)$, for $a \in AP_\phi$
- $\pi \models [e] \Leftrightarrow |\pi| \geq 2$ 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 $\pi^1 \models \phi$
- $\pi \models \phi_1 \, \mathcal{U} \, \phi_2 \Leftrightarrow$ 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:

$$F \, \phi \equiv \; true \, \mathcal{U} \, \phi$$
$$G \, \phi \equiv \neg(true \;\; \mathcal{U} \, \neg\phi)$$
$$\phi_1 \, \mathcal{R} \, \phi_2 \equiv \neg(\neg\phi_1 \, \mathcal{U} \, \neg\phi_2)$$
$$\phi_1 W \phi_2 \equiv \neg(true \;\; \mathcal{U} \, \neg\phi_1) \vee (\phi_1 \, \mathcal{U} \, \phi_2)$$

We say that a state s in TS_M satisfies an $\text{LTL}^{[e]}$ formula ϕ if for every path π starting in s we have $\pi \models \phi$. Subsequently, an Event-B model M satisfies an $\text{LTL}^{[e]}$ 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 $\text{LTL}^{[e]}$ 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)
- $\psi_1 \vee \psi_2 \in Cl(\phi) \Leftrightarrow \psi_1, \psi_2 \in Cl(\phi)$
- $X\psi \in Cl(\phi) \Leftrightarrow \psi \in Cl(\phi)$
- $\neg X\psi \in Cl(\phi) \Leftrightarrow X\neg\psi \in Cl(\phi)$
- $\psi_1 \, \mathcal{U} \, \psi_2 \in Cl(\phi) \Leftrightarrow \psi_1, \psi_2, X(\psi_1 \, \mathcal{U} \, \psi_2) \in Cl(\phi)$

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 $X\psi \in Cl(\phi)$,
- $\psi_1 \,\mathcal{U}\, \psi_2 \in Cl(\phi) \Leftrightarrow \psi_2 \in F$ or $\psi_1, X(\psi_1 \,\mathcal{U}\, \psi_2) \in F$ for every $\psi_1 \,\mathcal{U}\, \psi_2 \in Cl(\phi)$,

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 [LP85] for checking $M \models \phi$ is based on attempting to construct a directed graph $\mathcal{A}(TS_M)$ that has an infinite path

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

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

Such paths as π_α are called also *α-paths*.

Clearly, $(s_i, F_i) \stackrel{e_i}{\to} (s_{i+1}, F_{i+1})$ is an edge in $\mathcal{A}(TS_M)$ if and only if $s_i \stackrel{e_i}{\to} s_{i+1}$ 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 $\psi_1 \,\mathcal{U}\, \psi_2 \in F$ 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 $M \models \phi$.

An event is called a *stutter* event if it preserves the truth value of each atomic and transition proposition of the property being checked. Formally, this means that an event e is stutter with respect to an LTL$^{[e]}$ property ϕ if for each transition $s \stackrel{e}{\to} s'$ in TS_M we have $L(s) \cap AP_\phi = L(s') \cap AP_\phi$ and $e \notin TP_\phi$, where AP_ϕ and TP_ϕ are the sets of the atomic propositions and transition propositions in ϕ, respectively. In some literature sources like in [CEGP99] the *stutter* events are referred as *invisible* events and events that are *non-stutter* as *visible*.

A *path* in TS_M is a finite or an infinite alternating sequence of states and events $\pi = s_0 \stackrel{e_0}{\to} s_1 \stackrel{e_1}{\to} \ldots$ in TS_M such that for all $i \geq 0$ we have $(s_i, e, s_{i+1}) \in R$. By $\pi^i = s_i \stackrel{e_i}{\to} s_{i+1} \stackrel{e_{i+1}}{\to} \ldots$ we denote the suffix of the path π. Two paths π_1 and π_2 are called *stutter-equivalent* if both have identical state labellings after collapsing in each of them every finite sequence of identically labelled states to a single state. Two transition systems TS_1 and TS_2 are stutter-equivalent if for each path π_1 in TS_1 there exists a path π_2 in TS_2 that is stutter-equivalent to π_1 and vice versa.

2.3. Exhaustive Consistency Checking for B in ProB

Since the main contribution of this work is the optimisation of the algorithm for consistency checking of Event-B and B in ProB, we will give a quick overview of it in this section. The consistency checking algorithm (Algorithm 1) can be used to search a B model for deadlocks, invariant violation errors, assertion violation errors, as well as for a user-specified goal predicate. The algorithm can be used for verifying only invariant properties, i.e. properties that set conditions (called also invariant conditions) on all states of the model and can be verified by traversing each reachable state of the model and checking whether it satisfies the invariant condition.

The invariant of an Event-B machine is a condition on the state variables, which must hold permanently. Formally, an invariant is fulfilled by the respective machine when it is true in all reachable states. Typically, an invariant is proven in Event-B by induction: one proves that the initialisation establishes the invariant and that every event preserves the invariant, i.e. provided that the invariant and the guard of an event hold, the invariant still holds after execution of the event. This proof schema requires an invariant to be inductive[1];

[1] An invariant I of an Event-B model is called inductive if it satisfies both conditions: I holds in all initial states of the model and each event preserves I. Note that not all invariants proven to hold in all reachable states of a finite Event-B machine are inductive.

this is something we do not check in this paper nor does ordinary model checking (see Section 5.3 in [LB08]). In general, the invariant I of an Event-B machine is a conjunction of predicates and can be specified as an LTL formula $G\ (I)$ that, in particular, is an invariant property.

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
```

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.

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. The *read* and *write* sets of an event specify the set of variables that are read and written by the event, respectively.

Definition 3.1 (Read and Write Sets of an Event). Let e be an event of some Event-B machine M. Then, we define the following sets concerning the variables of M which are read and modified by e:

- $\mathit{read}(e)$ denotes the set of all variables that are read by e. As variables can be read in both parts, in the guards and the actions, of e one can further define the following two types of *read* sets:
  - $\mathit{read}_{\mathbf{G}}(e)$: the set of variables that are read in the guards' part of e, and
  - $\mathit{read}_{\mathbf{S}}(e)$: the set of variables that are read in the actions' part of e.
- $\mathit{write}(e)$ denotes the set of all variables that are modified by e.

Note that the local variables of an event are not considered in the *read* sets of the event.

Example 3.1 (Read/Write Sets). Consider the following event:

```
event e =
  any
    t
  where
    t ∈ {2, 3, 4}
    (x > 2 ∗ t) ∧ (z > 10)
  then
    y := x + y
  end
```

Then, e has the following read sets: $\mathit{read}(e) = \{x, y, z\}$, $\mathit{read}_{\mathbf{G}}(e) = \{x, z\}$, and $\mathit{read}_{\mathbf{S}}(e) = \{x, y\}$. On the other hand, the write set of e consists only of the variable y as this is the only variable which is assigned in the actions' part of e.

In the rest of this work we will assume that each event is deterministic in order to simplify the presentation. However, our implementation does not make this assumption.

3.1. Introducing Independence

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

Definition 3.2 (Independence). Two events e_1 and e_2 are independent if for any state s with e_1, $e_2 \in \mathit{enabled}(s)$ the executions $s \overset{e_1}{\to} s_1 \overset{e_2}{\to} s'$ and $s \overset{e_2}{\to} s_2 \overset{e_1}{\to} s''$ 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 ($\mathit{read}(e_1) \cap \mathit{write}(e_2) = \varnothing$).
(SI 2) The write set of e_1 is disjoint to the read set of e_2 ($\mathit{write}(e_1) \cap \mathit{read}(e_2) = \varnothing$).
(SI 3) The write sets of e_1 and e_2 are disjoint ($\mathit{write}(e_1) \cap \mathit{write}(e_2) = \varnothing$).

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. 3.2.

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 events in Example 3.2. Apparently, e_1 and e_2 are not syntactically independent as (SI 1) is violated ($\mathit{read}(e_1) \cap \mathit{write}(e_2) = \{x\}$). 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 $\mathit{write}(e_1) \cap \mathit{read}(e_2) = \varnothing$, 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.

Example 3.2 (Event Dependency).

event e_1 =	**event** e_2 =
when	**when**
$x \in \mathbb{N}$	$z \geq 1 \wedge z \leq 10$
then	**then**
$y := y + 1$	$x := z \parallel z := z + 1$
end	**end**

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 event is feasible under certain constraints. This will form the basis of our analysis.

Definition 3.3 (The *test_event* procedure). For a given Event-B machine M, let Φ and Ψ be B predicates for M, and e an event from M. Then, we define **test_event** as follows:

$$test_event(\Phi, e, \Psi) = \begin{cases} true & \text{if there is a transition } s \xrightarrow{e} s' \text{ such that } s \models \Phi \text{ and } s' \models \Psi \\ false & \text{otherwise} \end{cases}$$

The predicates Φ and Ψ are used in order to constrain the search for e transitions in the state space of M. If, for example, Φ and Ψ are both tautologies (e.g., $1 = 1$) then *test_event* will return *true* if e is enabled in some state of M. Accordingly, if Φ is an obvious inconsistency (e.g., $1 = 2$), then *test_event* will return $false$ as there is no state s such that $s \models \Phi$.

Back to determining independence of events, 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 interfere each others' effect */
6      else
7          return (    (read_G(e1) ∩ write(e2) ≠ ∅ ∧ test_event(G_e1 ∧ G_e2, e2, ¬G_e1))
8                    ∨ (write(e1) ∩ read_G(e2) ≠ ∅ ∧ test_event(G_e2 ∧ G_e1, e1, ¬G_e2))    )
9      end if
10 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 3.2), the syntactic analysis is not precise enough to exactly determine how two events are related to each other. Therefore, in lines 7-8 in Algorithm 2 we further check if the events can disable each other by means of the *test_event* 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 that is read in the actions part of the other 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 7 and 8. 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_event* procedure. If, for example, e_2 assigns a variable that is read in the guard G_{e_1} of e_1 (i.e. if $\mathit{read}_{\mathbf{G}}(e_1) \cap \mathit{write}(e_2) \neq \varnothing$) 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 $\mathit{test_event}(G_{e_1} \wedge G_{e_2}, \cdot \xrightarrow{e_2} \cdot, \neg G_{e_1})$. 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 e_1 cannot be disabled after the execution of e_2.

Remark 3.1 (The Necessity for Clear Separation of *read* into $\mathit{read}_{\mathbf{S}}$ and $\mathit{read}_{\mathbf{G}}$). If two events e_1 and e_2 are considered as independent by means of procedure *dependent* in Algorithm 2, then one of the conditions that must be satisfied is the following one:

$$\mathit{read}_{\mathbf{S}}(e_1) \cap \mathit{write}(e_2) = \varnothing \wedge \mathit{write}(e_1) \cap \mathit{read}_{\mathbf{S}}(e_2) = \varnothing.$$

The predicate implies that each two independent events should satisfy the requirement: no one of the events should write variables that are read in the action part of the other event. A requirement for independent events that may not be obvious at first glance. Let us observe, for example, the following two events:

event e_1 =	**event** e_2 =
when	**when**
$x \geq 1$	$y \geq 1$
then	**then**
$x := x + 1$	$y := y + x$
end	**end**

Both events are not race dependent as they write different variables and additionally, neither e_1 nor e_2 can disable the other one because no one can influence the guard of the other event. However, e_1 and e_2 do not satisfy the commutativity condition for independent events from Definition 3.2. This can be readily sketched if we execute the event sequences $\langle e_1, e_2\rangle$ and $\langle e_2, e_1\rangle$ from some state in which the variables x and y are both greater or equal to 1. In that case, the effect of executing $\langle e_1, e_2\rangle$ will be different from the effect of executing $\langle e_2, e_1\rangle$, i.e. that $\langle e_1, e_2\rangle$ reaches a state different from the state reached by $\langle e_2, e_1\rangle$. This behaviour clearly violates the commutativity condition for independence of events as one can see in Fig. 2. In such a case we also say that e_1 and e_2 interfere each other.

In the next sections, we will often talk about events that are dependent or independent to a set of events. The following definition defines what the relations dependency and independence of event with regard to a set of events mean.

Definition 3.4 (Set Dependency/Independence). Let e be an event of some given Event-B machine M and $E \subseteq \mathit{Events}_M$ a non-empty set of events of M. We say that e is dependent on E if there is at least one event $e' \in E$ such that $\mathit{dependent}(e, e')$ evaluates to *true*. Otherwise, if for all events $e' \in E$ the procedure call $\mathit{dependent}(e, e')$ evaluates to *false*, then we say that e is independent to E, i.e. e is independent to each $e' \in E$.

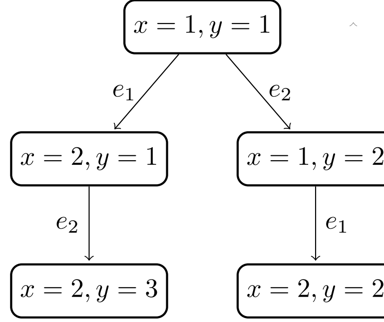

Fig. 2. Execution of dependent events.

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 3.5 (Enable Graph). An enable graph for an Event-B machine M is a directed edge graph $\texttt{EnableGraph}_M = (V, E)$, 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 $\texttt{EnableGraph}_M$.

In Definition 3.5, $e_1 \mapsto e_2$ 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 the *test_event* procedure. The call of *test_event* for testing whether e_1 can enable e_2 is then $test_event(G_{e_1} \wedge \neg G_{e_2}, \cdot \xrightarrow{e_1} \cdot, G_{e_2})$. The procedure *can_enable* is listed in Algorithm 3.

Algorithm 3: Determining Enabling Relations

1 **procedure boolean** can_enable(e_1, e_2)
2   **if** $(e_1 = e_2) \vee (write(e_1) \cap read_{\mathbf{G}}(e_2) = \varnothing)$ **then**
3   **return** *false*
4   **else**
5   **return** $test_event(G_{e_1} \wedge \neg G_{e_2}, \cdot \xrightarrow{e_1} \cdot, G_{e_2})$
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. In addition, the correctness of the reduction algorithm is presented.

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 possible 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 $e \in ample(s)$.

The intuition behind the first requirement (A 1) is to guarantee that each state that has at least one successor state in the original state space also has at least one successor state in the reduced state space. At the same time, (A 1) states that each deadlock state in the full state space is preserved by the reduction method. The most important condition for the correctness of the approach is the Dependency Condition (A 2). It ensures that each path being excluded in the process of reduction can be reconstructed from a path in the reduced state space using the properties of independent events, making sure in this way that no paths that may be crucial for the verification of the system are omitted. Guaranteeing both ample set conditions (A 1) and (A 2) suffices to use ample set reduction for checking models for deadlock freedom [GW91], [V89a].

To check linear-time properties expressed in LTL_{-X} by means of partial order reduction additionally the conditions (A 3) and (A 4) must be satisfied in order to guarantee the correctness of the approach. Condition (A 3) ensures the exclusion only of paths that are stutter equivalent to the paths being added to the reduced state space. The last ample set condition (A 4) makes sure that events are not ignored in the reduced state space. In this way, the Cycle Condition (A 4) guarantees that the full and the reduced transition systems are stutter-equivalent.

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 at 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 4.1 (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 the Dependency Condition (A 2) for all execution fragments in the original state space starting at s.

Proof. By contradiction. 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

$$\sigma = s \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} s_n \xrightarrow{e_{n+1}} \ldots$$

where $e_1, e_2, \ldots, e_n \notin \mathit{ample}(s)$ and e_{n+1} is dependent on $\mathit{ample}(s)$.

Since $e_1 \notin \mathit{ample}(s)$, by (A 2.1), we have that e_1 is independent of $\mathit{ample}(s)$. As next, we ask whether e_2 can be dependent on $\mathit{ample}(s)$. Suppose that e_2 depends on $\mathit{ample}(s)$. Then, there are two cases to be considered:

(i) e_1 enables e_2, and
(ii) e_1 does not enable e_2, i.e. e_2 is already enabled in s.

For case (i), by (A 2.2), it follows that e_2 must be independent of $\mathit{ample}(s)$ since $e_1 \notin \mathit{ample}(s)$. Thus, we have a contradiction to the assumption that e_2 depends on $\mathit{ample}(s)$. For case (ii), as e_2 is enabled in s and $e_2 \notin \mathit{ample}(s)$, then we can conclude, by (A 2.1), that e_2 is independent of $\mathit{ample}(s)$, which also contradicts the assumption that e_2 depends on $\mathit{ample}(s)$. Consequently, we have that e_2 does not depend on $\mathit{ample}(s)$.

Continuing applying the same procedure inductively for the residual states s_i and events e_i with $2 \leq i \leq n$ on $\mathit{ample}(s)$ one can conclude that e_{n+1} is not dependent on $\mathit{ample}(s)$. The independence of e_{n+1} to $\mathit{ample}(s)$ contradicts the assumption that there exists a fragment execution σ for which (A 2) is violated. Hence, (A 2) is satisfied by any $\mathit{ample}(s)$ fulfilling conditions (A 2.1) and (A 2.2). □

Remark 4.1 ((A 2.1) and (A 2.2) are Sufficient, but not Necessary Criteria for (A 2)). The local conditions (A 2.1) and (A 2.2) are sufficient local criteria for (A 2), but not necessary. Note that (A 2.1) and (A 2.2) together set a stronger condition on the ample sets than (A 2) as there could be sound ample sets that indeed fulfil the Dependency Condition (A 2), but not the local dependency conditions. This can be demonstrated by means of the following example. Consider the Event-B machine *Example* and its state space graph, both depicted in Fig. 3. Both events e_1 and e_2 are enabled at s_0 and obviously both events are independent to each other (e_1 and e_2 are syntactically independent). Looking at the transition system of the Event-B machine *Example* we can easily conclude that both sets $\{e_1\}$ and $\{e_2\}$ are valid ample sets satisfying the Dependency Condition (A 2). However, neither $\{e_1\}$ nor $\{e_2\}$ fulfil the local condition (A 2.2) since both events e_1 and e_2 can enable e_3 which in turn depends on both sets of events $\{e_1\}$ and $\{e_2\}$.

4.3. Computing $\mathit{ample}(s)$

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. $\mathit{Dependent}_M$ and $\mathit{EnableGraph}_M$ are the dependent relation and the enable graph computed for the corresponding Event-B machine M, respectively (see Algorithm 2 and Definition 3.5). 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 a 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 $\mathit{reachable}(\alpha, G)$ we denote the set of vertices that are reachable from vertex α in G. The set T is meant to be $\mathit{enabled}(s)$, where s is the currently processed state. Accordingly, the set S in Algorithm 4 is intended to be $\mathit{ample}(s)$. The output of the *ComputeAmpleSet* is an ample set satisfying the first three conditions of the ample set constraints.

The first step of computing $\mathit{ample}(s)$, 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 with 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 with respect 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 searching 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 further test if S violates the second local dependency condition (A 2.2). If there is

MACHINE Example
VARIABLES
x, y
...
EVENTS
Initialisation
begin
act1 : $x, y := 0, 0$
end
Event $e_1 \widehat{=}$
when
grd1 : $x = 0$
then
act1 : $x := x + 1$
end
Event $e_2 \widehat{=}$
when
grd1 : $y = 0$
then
act1 : $y := y + 1$
end
Event $e_3 \widehat{=}$
when
grd1 : $x + y = 2$
then
act1 : $x := 2$
act2 : $y := 2$
end
END

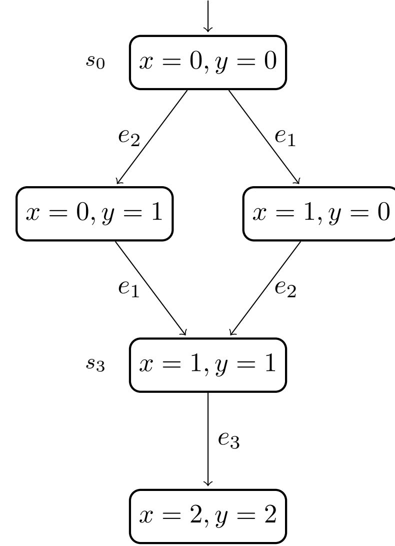

Fig. 3. (A 2.1) and (A 2.2) are sufficient, but not necessary conditions for (A 2)

some event $\beta \in I$ for which condition (A 2.2) is violated, then 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 4 runs until a valid ample set has been found or all potential ample sets fail to satisfy conditions (A 2) and (A 3) (then 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 (see Theorem 4.2). We will prove Theorem 4.2 with the aid of three lemmas where each of them states that the result returned by *ComputeAmpleSet* satisfies respectively the ample set conditions (A 1), (A 2.1), and (A 2.2). The stutter condition (A 3) will not be handled specifically for theorem's proof as we assume at this point that the procedure for checking whether S is a stutter set is correct.

Lemma 4.1. Let A be a set computed by means of the procedure *ComputeAmpleSet* 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 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 is determined by the *DependencySet* procedure and accordingly we can conclude that it has at least one element, the event α, since $\alpha \in reachable(\alpha, G)$. Thus, S is a non-empty set also in the second case. 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. $\square$

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(set T)
2      foreach α ∈ T such that α randomly chosen do
3          boolean b := true;
4          set S := DependencySet(α,T);                        /* (A 2.1) holds */
5          set 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(event α, set T)
19     set 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
```

Lemma 4.1 states that $ComputeAmpleSet(T) = \varnothing$ if and only if $T = \varnothing$. 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 is shown by proving that A satisfies both local dependency conditions (A 2.1) and (A 2.2). We already have shown in Theorem 4.1 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 A also fulfils the Dependency Condition (A 2).

Lemma 4.2. 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, i.e. (A 2.1) is fulfilled by 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 A is a set computed by the procedure *DependencySet* for some event $\alpha \in T$. Thus, showing that all events $\beta \in T \setminus A$ are independent of A, it is equivalent to showing 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 contradiction. Assume there is an event $\gamma \in T \setminus S$ such that γ depends on S. That is, γ is dependent on some event β which 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 elements of T (see lines 20-22 in Algorithm 4).

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 4.2 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 4.3. 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 $\gamma_1, \ldots, \gamma_n, \gamma \notin A$ and γ depends on A.

Proof. By Lemma 4.2 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 of 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 $\gamma_1, \ldots, \gamma_n, \gamma \notin A$ and γ depends on A does not exist 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 Event-B 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) $\beta \mapsto \gamma \notin Paths(EnableGraph_M)$: 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 4.2 γ is independent of 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) $\beta \mapsto \gamma \in Paths(EnableGraph_M)$: 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 refused 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 exist 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 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 $\gamma_{k+1} \to \gamma$ in $EnableGraph_M$ infers that γ cannot become enabled after the execution of the event γ_{k+1} and as a consequence we can 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 $\pi_0 = \gamma_{k+1} \mapsto \gamma$ 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 at the end we

have constructed as a result from σ^{k+1} an enabling path π_{k+1} that is an element of $Paths(EnableGraph_M)$. Then, we consider two cases for the enabling path π_{k+1}.

(2.1) In the first case the enabling path starts with β, i.e. $\pi_{k+1} = \beta \mapsto \hat{\gamma}_1 \mapsto \ldots \mapsto \hat{\gamma}_j \mapsto \gamma_{k+1} \mapsto \gamma$, 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*. Owing to the fact that A was 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} cannot 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 $false$ (the inner **foreach**-loop (lines 7-11) considers all enabled events at s in $T \setminus A$).

Thus, we can conclude from the induction proof that for $\beta \in T \setminus A$ 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 4.1, 4.2, and 4.3 we can state the following theorem.

Theorem 4.2. 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 reveals 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 [V89a]. 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.

Condition (A 3) guarantees the exclusion of paths that are stutter-equivalent to the paths in the reduced model. Computing the set of stutter events in an Event-B machine depends on the property being checked. In general, one says that an event is a stutter event if it cannot influence the value of the checked property. For instance, if we check an Event-B machine for consistency, i.e. whether the events of the machine preserve its invariant, then the stutter events of the machine are all events that trivially preserve the invariant. On the other hand, all events that cannot be trivially proven to satisfy the invariant are considered as non-stutter. Similarly, one can determine the non-stutter events of an Event-B machine with respect to an $\mathrm{LTL}^{[e]}$ formulae ϕ. An event e is considered to be non-stutter with respect to ϕ when e modifies a variable that is used in some B predicate constituting an atomic proposition in ϕ being checked or if ϕ contains $[e]$ as a transition proposition. All events that are not non-stutter with respect to ϕ are considered as stutter events.

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 wit respect to (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 [V89a].

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 [BK08]) 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 sometimes it expands fully states 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 [BBR10] and [BLL09]. Furthermore, the implementation of condition (A 4) in this way is also a design decision as we want easily to reuse the reduction algorithm for LTL model checking in ProB (see Section 5).

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 an argument.

Algorithm 5: Computation of the Ample Transitions

```
1  procedure compute_ample_transitions(state s)
2      set T := compute all enabled events in s;
3      set S := ComputeAmpleSet(T);
4      foreach evt ∈ S do
5          state 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 was fully explored */
12         end if
13     end foreach
14 end procedure
-----------------------------------------------------------------
15 procedure execute_event(state s, event 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
```

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 an argument the state being currently processed. The computation of the successor states and the insertion of the new determined transitions are realised by the procedure *execute_event* in lines 15-23.

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 ($id(s) \geq id(s')$).

5. 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 [PL10] 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.

5.1. LTL$^{[e]}$ Model Checking in ProB

The ProB LTL$^{[e]}$ model checker follows the tableau approach from [LP85] and can check properties specified in LTL$^{[e]}$. The algorithm presented in [PL10] 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 $M \models \phi$ 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 satisfy $\neg\phi$. 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 $\neg\phi$ was discovered, we have proven that $M \models \phi$. The search for SCCs in the ProB LTL$^{[e]}$ model checker is based on the Tarjan's algorithm [T72].

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. The full state space exploration is usually avoided when a bad path is found in the tableau search graph explored so far. The dynamic approach can be very effective especially when the model being checked 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 [P94], [CEGP99], respectively.

The tableau algorithm from [PL10] is implemented in C using a callback mechanism for evaluating the atomic propositions and the outgoing transitions in SICStus Prolog. While constructing the search 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 the previous subsection 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 checked. 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 which ample set conditions have to be adapted and how the algorithms Algorithm 4 and Algorithm 5 can be re-used 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.

MACHINE Example
VARIABLES
x, y
INVARIANTS
...
EVENTS
Initialisation
begin
act1 : $x, y := 0, 0$
end
Event $e_1 \widehat{=}$
when
grd1 : $x = 0$
then
act1 : $x := x + 1$
end
Event $e_2 \widehat{=}$
when
grd1 : $y = 0$
then
act1 : $y := y + 1$
end
Event $e_3 \widehat{=}$
when
grd1 : $x = 1$
grd2 : $y = 1$
end
END

Fig. 4. LTL$^{[e]}$ Model Checking for Event-B.

5.2. LTL$^{[e]}$ Formulae Preserved by Partial Order Reduction

Before discussing how the reduction algorithm can be adapted for LTL$^{[e]}$ model checking with partial order reduction for Event-B, 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

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

is satisfied, where $\widehat{TS}_M$ 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, in general, such formulae are not invariant under stuttering [PW97]. 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 "$[e_1] \Rightarrow F\ \{x = 2\}$" encodes the property "the execution of e_1 in the current state implies that the variable x will eventually be equal to 2". What we need to examine is whether formulae with the *executed*-operator $[\cdot]$ violate equivalence (1).

Example 5.1 (LTL$^{[e]}$ **Formulae with Execute Operator).** Consider, for example, the LTL$^{[e]}$ formula "$\phi = ([e_1] \Rightarrow F\ \{x = 2\})$" and the Event-B machine depicted in Fig. 4. The transition system TS_M on the left side of Fig. 5 illustrates the full state space of the machine in Fig. 4.. As we can easily see, the events e_1 and e_2 defined in the Event-B machine are syntactically independent. Further, e_3 is independent to both events e_1 and e_2. To apply the ample set approach for checking ϕ on *Example* we also need to determine the stutter events with respect to ϕ. Event e_1 is considered as a non-stutter event since ϕ comprises e_1 as a transition proposition. Further, neither e_2 nor e_3 modifies x and thus we can safely assume that both events e_2 and e_3 are stutter events with respect to ϕ.

Consider the two transition systems TS_M and $\widehat{TS}_M$ in Fig. 5. The transition graph TS_M illustrates the full state space of the Event-B machine from Fig. 4, while $\widehat{TS}_M$ represents the reduced transition system by applying the ample set approach with respect to ϕ. The reduction takes place in the initial state s_0 of the machine. In s_0 it suffices to choose only one of the enabled events e_1 and e_2 in order to guarantee (A 2) since e_1 and e_2 are independent and the execution of each will not enable an event which is dependent on the other one. However, the only valid ample sets in s_0 are $\{e_2\}$ and $\{e_1, e_2\}$ since $\{e_1\}$ is excluded as a valid ample set because of (A 3). Hence, $ample(s_0) = \{e_2\}$ is sufficient in regard to the ample set approach.

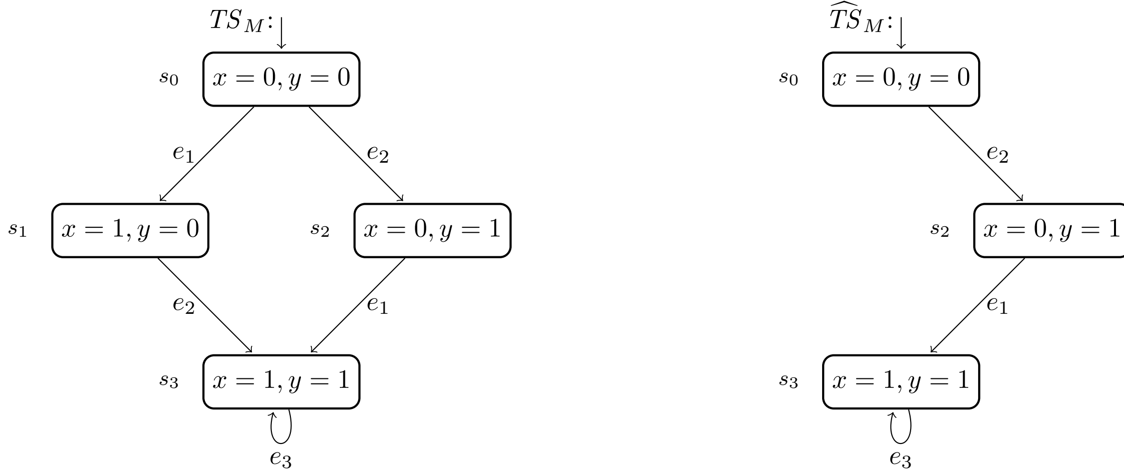

Fig. 5. Ample set reduction does not preserve the truth value for the formula $[e_1] \Rightarrow F\ \{x = 2\}$.

Checking ϕ on TS_M and $\widehat{TS}_M$ yields different results. Obviously, $TS_M \not\models \phi$ as the path $\rho = s_0 \stackrel{e_1}{\to} s_1 \stackrel{e_2}{\to} s_3 \stackrel{e_3}{\to} s_3 \ldots$ in TS_M is a counter example for ϕ and $\widehat{TS}_M \models \phi$ since no path in $\widehat{TS}_M$ does execute e_1 from the initial state. The example shows that, in general, $\mathrm{LTL}^{[e]}$ formulae with the execute-operator do not fulfil the equivalence $TS_M \models \phi \Leftrightarrow \widehat{TS}_M \models \phi$. Thus, the set of $\mathrm{LTL}^{[e]}$ formulae that is preserved by partial order reduction is the set of all $\mathrm{LTL}^{[e]}$ formulae without the next-operator X and without the execute-operator $[\cdot]$. This subset of $\mathrm{LTL}^{[e]}$ formulae we denote with LTL_{-X}.

5.3. The Static Approach

The static approach of $\mathrm{LTL}^{[e]}$ 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 $\mathrm{LTL}^{[e]}$ 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 explored. On the contrary, the static approach 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 $\mathrm{LTL}^{[e]}$ model checking algorithm to check the respective 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 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 procedure *compute_ample_transitions* in Algorithm 5 for expanding each reachable state. Basically, we can use the consistency checking algorithm (Algorithm 1) to compute the reduced state space of an Event-B model by adapting 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. Note that the procedure *compute_ample_transitions* is not sensitive with respect to the exploration strategy. That is, whatever exploration strategy we choose the reduction of the state space graph by means of *compute_ample_transitions* remains sound. However, using the implementation in Algorithm 5 for fulfilling (A 4') is not optimal because it may cause the fully expansion of more states than

necessary. As a result, one could consider a more efficient implementation for satisfying (A 4') by using the fact that, for depth-first search, every cycle has an edge that goes back to a state in the queue.

5.4. 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 $\widehat{TS}_M$. 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 $\mathcal{A}(\widehat{TS}_M)$, 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 $\widehat{TS}_M$ if it is certain that there is a back transition from an atom (s, F) closing a cycle in $\mathcal{A}(\widehat{TS}_M)$. 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 $\mathcal{A}(\widehat{TS}_M)$ has at least one atom (s, F) such that state s is fully expanded in $\widehat{TS}_M$, 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 $\mathcal{A}(\widehat{TS}_M)$. 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 $\widehat{TS}_M$ will be fully expanded if there is an atom (s, F) in $\mathcal{A}(\widehat{TS}_M)$ such that an edge from (s, F) exists going back to an atom on the search stack.

6. Discussion and Evaluation

6.1. Discussion

6.1.1. The Approach

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 dependency relation $Dependent_M$ and the enable graph $EnableGraph_M$. $Dependent_M$ and $EnableGraph_M$ are computed prior to 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 the syntactic analyses. In fact, timeouts are set by default in ProB for diminishing the possibility that the overhead caused by the 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) [GW91], [V89a]. 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 with respect to I is to use either proof information (e.g., from the Rodin provers) or ProB for checking invariant preservation for operations: every event which we can prove to preserve the invariant will be considered as a stutter event.

Explicit state model checking is a practical and convenient method for automatic verification of finite state systems. On the other hand, verification of infinite state systems will be not possible by means of model checking as not all possible states of the system can be explored. Thus, model checking is generally considered as unsuitable for verification of infinite state systems. However, as discussed in earlier papers [L08], [LB08] ProB can deal quite well with infinite state systems, in the sense that counterexamples can be discovered by means of different state space exploration strategies: depth-first, breadth-first, and mixed breadth/depth-first search. This was also one of the motivations to design the implementation of the ample set method to guarantee sound state space reductions for different exploration strategies (see Section 4).

If the ProB model checker is ran on an Event-B model with infinite state space, then verification will never be possible as one can keep running the model checker until it either finds a counterexample or it runs out of memory. However, explicit state model checking in ProB with partial order reduction can sometimes be used for verifying deadlock absence of infinite state Event-B machines. Recall that for checking a system just for deadlocks using the ample set technique for state space reduction it suffices to require that only the ample set conditions (A 1) and (A 2) are satisfied by each ample set of each state. Using this fact one can infer that for certain infinite state systems the absence of deadlock can be verified when model checking with partial order reduction. This relies on the fact that both conditions (A 1) and (A 2) together cannot guarantee that events will be ignored in the reduced state space.

To make this more clear consider an Event-B machine M with one initial state s_0 and with two events $e_1 = skip$ and $e_2 = x := x+1$, where $x \in \mathbb{Z}$. Obviously, M has an infinite state space as e_2 is enabled in each state and increments the variable by 1. Choosing $\{e_1\}$ as an ample set in the initial state s_0 can be considered as a sound ample set at s_0 if we look just for deadlock errors. As a consequence, the reduction method from Section 4 will reduce the state space of M to one state with e_1 as a self loop. In this case the reduced search will terminate and exit with the result that M is deadlock-free. Thus, explicit state model checking in ProB with partial order reduction can in some cases verify certain infinite state Event-B machines to be deadlock-free.

6.1.2. Correctness of the Approach

During the development we tested our reduction algorithm at first on different models that we constructed in order to demonstrate its correctness. One way of demonstrating the correctness of the algorithm was to show that the reduction technique preserves the relevant error states from the original (full) state space. With relevant error states we mean the states that are intended to be found or not found in the state space of the model. If, for example, we perform just deadlock-freedom check the relevant error states are the deadlocks.

The correctness of the reduction algorithm could be to some extent confirmed by testing whether errors such as deadlock and invariant violation errors are also preserved in the reduces state space of the model being checked. However, in case the model is error-free we needed to test the correctness of the reduction by means of other heuristics. We could to some degree assure that the reductions in such cases were sound by performing coverage analyses after the verification with the reduced search. We have used two types of coverage analyses for ensuring the soundness of the reductions for the particular model: coverage of the events presented in the model and domain coverage of the variables and constants in the model. The events coverage analysis checks whether all events executed in the non-reduced system have been executed at least once in the reduced system, while the domain coverage analysis examines whether the intervals in which the single variables range match for the reduced and the non-reduced system.

Using both analyses for advocating the correctness of the reduction search for an error-free model makes sense when invariant violation search is performed. Checking only for deadlock-freedom does not explicitly require that all events executed in the original state space should be also executed in the reduced state space. If, for example, the observed specification has a pure skip event $evt := skip$ and only deadlock absence checking is performed, then in each state the set $\{evt\}$ is a valid ample set since *evt* does not read or write any variables and only conditions (A 1) and (A 2) should be satisfied (both conditions are sufficient to guarantee that deadlock states are preserved in the reduced state space). In that case, choosing for some states only *evt* to be executed will lead to ignoring all other enabled events in those states and possibly to ignorance of some of the events in the reduced state space of the model. However, in this case ignorance of events is not relevant for proving the model for deadlock absence since an *evt* loop is always present in each state of the state space.

In addition, we have formally proven the correctness of our reduction algorithm (see Section 4.3). Indeed, in the course of providing a formal proof for Algorithm 4 in [DL14] we have found particular cases for which the algorithm may calculate ample sets that do not satisfy the local dependency condition (A 2.1). As a result of this, we revised the algorithm in [DL14] and replaced it by its corrected version in this work. Accordingly, a proof of correctness of Algorithm 4 has been added to Section 4.3 and the implementation in ProB has been adapted.

One could ask why not use a formal language to specify the reduction algorithms and then proving whether the specification satisfies the desired properties, for example, by using a proof assistant tool such as Rodin [ABHV06] or Isabelle [NWP02]. In the first place, proving the correctness of the reduction algorithm presented in Section 4.3 appeared to be essential as in our experience with partial order reduction there had been so many little details that were of importance to be regarded that one could easily lose track of the correctness of the approach. Therefore, providing a proof of correctness is vital to convince ourself and the readers that the method we have presented is sound. In the second place, giving the proof of correctness in a fully mathematical way is in our opinion more concise and does not distract from the main contribution in this work, namely tackling the state space explosion problem for Event-B by means of partial order reduction. We think that providing a formal verification of our algorithms using a proof assistant tool is of practical interest and such a work makes more sense to be elaborated in an article on its own. An interesting approach similar to that presented in [ELN+13] could be to specify and verify our model checking algorithms from Section 2 and Section 4 using a theorem prover such as Isabelle. After proving the correctness of the algorithms one could let the theorem prover to generate code that could be used as a reference implementation of the implementation of partial order reduction introduced in this work.

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

Model	Algorithm	States	Transitions	Analysis Time	Model Checking Time
Counters	Deadlock + Inv.	4,550	13,136	-	4.921*
	Deadlock + Inv. (POR)	944	1,769	< 0.152	1,227*
	Deadlock	110,813	325,004	-	81.950
	Deadlock (POR)	152	154	0.154	0.158
Conc v1	Deadlock + Inv.	128,562	290,558	-	339.447
	Deadlock + Inv. (POR)	128,562	267,669	6,549	487.733
	Deadlock	128,562	290,558	-	244.356
	Deadlock (POR)	91,312	143,382	0.327	248.714
BPEL v6	Deadlock + Inv.	2,248	4,960	-	6.946
	Deadlock + Inv. (POR)	2,248	4,960	2.242	9,503
	Deadlock	2,248	4,960	-	6.162
	Deadlock (POR)	570	634	0.640	1.975
Token Ring	Deadlock + Inv.	16,389	90,133	-	33.633
	Deadlock + Inv. (POR)	16,218	67,516	0.087	39.028
	Deadlock	16,389	90,133	-	33.416
	Deadlock (POR)	14,287	36,292	0.081	31.144
Sieve	Deadlock + Inv.	8,328	28,436	-	240.099
	Deadlock + Inv. (POR)	8,109	24,374	14.757	250.680
	Deadlock	8,328	28,436	-	250.398
	Deadlock (POR)	5,106	11,378	7.145	180.843
Phil v2	Deadlock + Inv.	2,351	4,528	-	8.558
	Deadlock + Inv. (POR)	2,338	4,257	0.721	11.912
	Deadlock	2,351	4,528	-	8.954
	Deadlock (POR)	2,332	4,149	0.592	11.146

(*) Invariant Violation

6.2. Evaluation

We have evaluated our implementation of partial order reduction on various models that we have received from academia and industry.[2] 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.

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:

Deadlock+Inv.: Model checking for deadlocks and invariant violations.

Deadlock+Inv.(POR): Model checking for deadlocks and invariant violations with partial order reduction.

Deadlock: Model checking for deadlocks only.

Deadlock(POR): Model checking only for deadlocks with partial order reduction.

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.[3] 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 **Deadlock(POR)**. For this type of checks, the results produced by **Deadlock(POR)** were compared with the results of **Deadlock**.

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

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

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

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 of both tables.

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 **Deadlock+Inv.(POR)**. *BPEL* [AA09] is a case study of a business process for a purchase order. *Phil* [BDSW12] is an Event-B model representing a solution of the dinning philosophers problem with four philosophers. Both *BPEL* and *Phil* are carried by a stepwise development via refinement; their last refinement versions *Phil v2* and *BPEL v6* are presented in Table 1. The test case *Conc v1* is a case study from [A10] which is used to show how the development of concurrent programs can be supported by using Event-B. The development of the system is conducted via refinement and the test results of the first refinement of the model, *Conc v1*, are given in Table 1. It should be noticed that the original machine *Conc v1* from [A10] has an infinite state space. As a consequence, we did some minor changes in the Event-B model in order to make the model finite state. At last, *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.

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 **Deadlock+Inv.(POR)**. However, the model checking runtime in this case is not significantly different from the model checking runtime for the standard search **Deadlock+Inv.**. As expected, significant reduction of the state space and thus the overall time for checking the *Counters* model were gained by both reduction searches **Deadlock+Inv.(POR)** and **Deadlock(POR)**. For the first two types of test cases 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 could be gained.

7. Related Work

7.1. Optimisations of the ProB Model Checker

A great deal of work has been devoted to optimising the ordinary ProB model checker for B and Event-B. In this subsection, 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 [TLSB07] and the permutation flooding method [LBST07]. 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 [LM07] 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 [BeL09]. 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 [LBe10], 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 [BeL11]. In this work

enable graphs are used to encode the information about independence[4] and dependence of events by means of enabling predicates. In addition, the authors suggest a method for optimising model checking using the information from the enable graph. The idea is to speed up the state space exploration by omitting the evaluation of the guards of events that are known to be disabled in states being currently explored. The information of the disabledness of an event in some particular state is derived by means of evaluating the enabling predicates in 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.

7.2. Model Checking with Partial Order Reduction

Partial order reduction has been shown to be a very effective technique for optimising automatic verification of concurrent systems by means of model checking. Many prominent model checkers make use of partial order reduction for yielding smaller verification times. In this subsection, we will give a short overview of the application of the method in various model checkers and its impact on verifying systems formalised in low-level formalisms.

SPIN [H03] is a verification tool primarily used for the formal verification of multi-threaded software applications specified in Promela, the formalism supported by SPIN. Partial order reduction has been established as an effective technique for optimising the verification runs of Promela models [HD94], [CGMP99]. The strategy for reducing the state space in SPIN is similar to ours, which reduces the number of states by calculating ample sets. The implementation of partial order reduction in SPIN looks for one process satisfying the ample set conditions in the currently processed state. If such a process is found, then only the actions of this process are executed in the particular state.

A similar idea for increasing the performance of partial order verification techniques by refining the dependency relation was introduced in [GP93]. The authors of [GP93] practically demonstrate that verification by partial order reduction can in some cases substantially profit from the refined dependencies and improve the performance of both time and memory requirements. The partial order verification technique using more refined dependency determination was implemented in SPIN. Evaluations of the algorithm have shown that one can reduce the memory requirements of the verification of some real protocols up to factor of five compared to the partial order verification algorithm using an unrefined dependency relation.

DIVINE [BBH+13] is another explicit state model checker which uses partial order reduction for better runtime performance. In particular, DIVINE supports state space reduction by means of partial order reduction for parallel LTL model checking [BBR10]. The implementation of partial order reduction in DIVINE uses a topological sort proviso for guaranteeing the correct construction of the reduced state space graph with respect to the cycle condition (A 4) in order to be also compatible with parallel exploration strategies. The reduced search in DIVINE is available for the DVE specification language, which is one of its input languages. Similar to Promela the DVE specifications are composed of processes specifying the behaviour of the system that are the basic modelling unit in DVE.

Partial order reduction has also been successively applied for efficient explicit state model checking in the LTSmin model checker [KLM+15]. LTSmin is equipped with multi-core algorithms for on-the-fly LTL model checking with partial order reduction, as well as multi-core symbolic model checking, and provides support for the analysis and verification of systems specified in different modelling languages, such as Promela, DVE, Upaal, and others. The link between the various modelling language front-ends and the verification algorithms of LTSmin is established through a common interface called PINS (Partitioned Next-State Interface). By means of PINS the modelling languages details are abstracted away utilizing a corresponding implicit state space. The optimisation of the algorithms for explicit state model checking by means of partial order reduction is realised through using read, write and guard/transition dependency matrices. The reduction of the state space is accomplished by computing in each state a subset of enabled transitions fulfilling the conditions for stubborn sets[5] [V89a], [V89b]. Partial order reduction in LTSmin is available only for explicit state model

[4] The definition of independence between events in [BeL11] is different from the definition of independence with respect to partial order reduction. In [BeL11] two events are considered to be independent if each of the events cannot influence the guard of the other one.

[5] The stubborn set method from Valmari is another approach for reducing the state space by means of partial order reduction. [V89a], [V90]

checking and the reduction method can be used in combination with parallel LTL model checking, where the ignoring problem is solved by means of the parallel cycle proviso from [LW11].

Partial order reduction is used for improving LTL model checking and refinement checking in PAT [SLD08], a framework which among others provides support for analysing and verifying concurrent systems formalised in the process algebra CSP#. The reduction technique implemented in PAT exploits and extends the ideas for applying partial order reduction for process algebras and refinement checking in [V96] and [W99].

The result in Section 5.2, which states that partial order reduction does not preserve $\mathrm{LTL}^{[e]}$ formulae with the execute operator, was also obtained in [BBC+09] for the state/event-LTL (SE-LTL) formalism in [CCO+04]. To overcome this limitation the authors in [BBC+09] propose a new type of stutter-equivalence, state/event stutter-equivalence, as well as a new logic fragment of SE-LTL, weak SE-LTL. Properties specified in the weak SE-LTL fragment are preserved by state/event stutter-equivalence. Both, the state/event stutter-equivalence and the weak SE-LTL fragment, enable one to apply partial order reduction to SE-LTL.

8. Conclusions 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 for explicit state model checking 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 $\mathrm{LTL}^{[e]}$ 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 of these .

In Remark 4.1 in Section 4.2 we considered an example of an Event-B machine in which we demonstrated that the local dependency conditions (A 2.1) and (A 2.2) are sufficient, but not necessary criteria for (A 2). We showed that no reduction at the initial state of the Event-B machine in Fig. 3 could be performed by means of (A 2.2) although choosing to execute only one of the events at the initial state would have not lead to a violation of (A 2). This example shows that there may be a potential to refine the reduction algorithm in terms of (A 2.2) in order to achieve much more reductions. Future work will be concerned in studying whether the reduction algorithm in Section 4 may be refined in terms of the local dependency condition (A 2.2) in order to gain more state space reductions.

At last, we are planing to elaborate the reduction algorithm to be used also in combination with other available optimisations of the ProB Model Checker such as symmetry reduction and directed model checking. Additionally, we plan a thorough comparison of our implementation of partial order reduction with implementations of partial order reduction in other model checkers.

Acknowledgement

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

References

[A96] Abrial, J.-R.: *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[A10] Abrial, J.-R.: *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[AA09] Ait-Sadoune, I., Ait-Ameur, Y.: *A Proof Based Approach for Modelling and Verifying Web Services Compositions.* ICECCS '09, pages 1-10, Washington, DC, USA, 2009, IEEE Computer Society.

[ABHV06] Abrial, J-R., Butler, M., Hallertede, S., Voisin, L. *An open extensible tool environment for Event-B.* ICFEM 2006, volume 4260 of LNCS, pages 588605. Springer, 2006.

[BBC+09] Bene, N., Brim, L., ern, I., Sochor, J., Vaekov, P., Zimmerova, B.: *Partial Order Reduction for State/Event LTL.* iFM 2009, volume 5423 of LNCS, pages 307-321, Springer Berlin Heidelberg, 2009.

[BBH+13] Barnat, J., Brim, L., Havel, V., Havlek, J., Kriho, J., Leno, M., Rokai, P., Still, V., Weiser, J. *DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs* CAV, volume 8044 of LNCS, pages 863-868, Springer, 2013

[BBR10] Barnat, J., Brim, L., Rockai, P.: *Parallel Partial Order Reduction with Topological Sort Proviso.* In SEFM, pages 222-231, IEEE Computer Society, 2010.

[BDSW12] Boström, P., Degerlund, F., Sere, K., Waldén, M.: *Derivation of Concurrent Programs by Stepwise Scheduling of Event-B Models.* Formal Aspects of Computing, pages 1-23, 2012.

[BeL09] Bendisposto, J., Leuschel, M.: *Proof Assisted Model Checking for B.* In ICFEM, volume 5885 of LNCS, pages 504-520, Springer, 2009.

[BeL11] Bendisposto, J., Leuschel, M.: *Automatic Flow Analysis for Event-B.* In FASE, volume 6603 of LNCS, pages 50-64, Springer, 2011.

[BK08] Baier, C., Katoen, J.-P.: *Principles of Model Checking.* The MIT Press, 2008.

[BLL09] Bosnacki, D., Leue, S., Lluch-Lafuente, A.: *Partial-Order Reduction for General State Exploring Algorithms.* STTT, 11(1): 39-51, 2009.

[CCO+04] Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N. Sinha, N. *State/Event Based Software Model Checking* In iFM, volume 2999 of LNCS, pages 128-147, 2004.

[CEGP99] Clarke,Jr., Edmund M., Grumberg, O., Peled, D. A.: *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[CGMP99] Clarke, E.M., Grumberg, O., Minea, M., Peled, D.: *State Space Reduction using Partial Order Techniques.* International Journal on STTT, 2(3):279-287, 1999.

[DL14] Dobrikov, I., Leuschel, M.: *Optimising the ProB Model Checker for B Using Partial Order Reduction.* In SEFM, volume 8702 of LNCS, pages 220-234, 2014.

[ELN+13] Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J-G.: *A Fully Verified Executable LTL Model Checker.* In CAV, volume 8044 in LNCS, pages 463-478, Springer, 2013.

[G96] Godefroid, P.: *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem,* volume 1032 of LNCS, Springer, 1996.

[GP93] Godefroid, P., Pirottin, D. *Refining Dependencies Improves Partial-order Verification Methods* In CAV, volume 697 of LNCS, Springer Berlin Heidelberg, 1993.

[GW91] Godefroid, P., Wolper, P.: *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties.* In CAV, volume 575 of LNCS, pages 332-342, Springer, 1991.

[H03] Holzmann, G.: *Spin Model Checker, the: Primer and Reference Manual* Addison-Wesley Professional, First edition, 2003

[HD94] Holzmann, G., Peled, D.: *An Improvement in Formal Verification* In Proceedings FORTE 1994, pages 197-211.

[KLM+15] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T. *LTSmin: High-Performance Language-Independent Model Checking* In TACAS, volume 9035 of LNCS, pages 692-707, Springer Berlin Heidelberg, 2015

[L08] Leuschel, M.: *The High Road to Formal Validation: Model Checking High-Level versus Low-Level Specifications.* In ABZ, volume 5238 of LNCS, pages 4-23, Springer, 2008.

[LBST07] Leuschel, M., Butler, M., Spermann, C., Turner, E.: *Symmetry Reduction for B by Permutation Flooding.* In Proceedings B'2007, volume 4355 of LNCS, pages 79-93, Springer, 2007.

[LB08] Leuschel, M., Butler, M.: *ProB: An Automated Analysis Toolset for the B Method.* STTT, 10(2): 185-203, 2008.

[LBe10] Leuschel, M., Bendisposto, J.: *Directed Model Checking for B: An Evaluation and New Techniques.* In SBMF' 2010, volume 6527 of LNCS, pages 1-16, Springer, 2010.

[LM07] Leuschel, M., Massart, T.: *Efficient Approximative Verification for B via Symmetry Markers.* In Proceedings International Symmetry Conference, pages 71-85, January 2007.

[LP85] Lichtenstein, O., Pnueli, A.: *Checking that Finite State Concurrent Programs Satisfy Their Linear Specifications.* In POPL'85, pages 97-107, New York, NY, USA, 1985, ACM.

[LW11] Laarman, A., Wijs, A.: *Partial-Order Reduction for Multi-Core LTL Model Checking.* In HVC 2014, volume 8855 of LNCS, pages: 267-283, Springer, 2014.

[NWP02] Nipkow, T., Wenzel, M., Paulson, L. C. *Isabelle/HOL–A proof assistant for Higher-Order Logic.* Springer-Verlag Berlin, Heidelberg, 2002.

[P77] Pnueli, A.: *The Temporal Logic of Programs.* In Proceedings of 18th IEEE Symposium on Foundations of Computer Science (SFCS '77), pages 46-57, IEEE Computer Society Press, 1977.

[P94] Peled, D.: *Combining Partial Order Reduction with On-the-fly Model-Checking.* In Proceedings of the Sixth Workshop on CAV, volume 818 of LNCS, pages 377-390, Springer, June 1994.

[PL10] Plagge, D., Leuschel, M.: *Seven at one stroke: LTL Model Checking for High-level Specifications in B, Z, CSP, and more.* STTT, 12(1): 9-21, Feb 2010.

[PW97] Peled, D., Wilke, T.: *Stutter-invariant Temporal Properties are Expressible without the Next-time Operator.* Information Processing Letters, 63(5): 243-246, 1997.

[RMQ10] Rosa, C. D., Merz, S., Quinson, M.: *A Simple Model of Communication APIs - Application to Dynamic Partial-order Reduction.* In Proceedings of 10th International Workshop on Automated Verification of Critical Systems (AVoCS 2010), volume 35, 2010.

[SLD08] Sun, J., Liu, Y., Dong, J. S.: *Model Checking CSP Revisited: Introducing a Process Analysis Toolkit.* In Proceedings of ISoLA, pages 307-322, Springer Berlin Heidelberg, 2008.

[T72] Tarjan, R.: *Depth First Search and Linear Graph Algorithms.* SIAM Journal on Computing 1(2), pages 146-160, 1972.

[TLSB07] Turner, E., Leuschel, M., Spermann, C., Butler, M.: *Symmetry Reduced Model Checking for B.* In TASE, pages 25-34, IEEE, June 2007.

[V89a] Valmari, A.: *Stubborn Sets for Reduced State Space Generation.* In Applications and Theory of Petri Nets, pages 491-515, 1989.

[V89b] Valmari, A.: *Eliminating Redundant Interleavings During Concurrent Program Verification.* In PARLE, volume 366 of LNCS, pages 89-103, Springer, 1989.

[V90] Valmari, A.: *A Stubborn Attack On State Explosion.* In CAV, pages 156-165, 1990.

[V96] Valmari, A.: *Stubborn Set Methods for Process Algebras.* In DIMACS, pages 213231, volume 29, 1996.

[W99] Wehrheim, H.: *Partial order reductions for failures refinement.* In Proceedings of the the 6th International Workshop on Expressiveness in Concurrency, Electronic Notes in Theoretical Computer Science, volume 27, pages 71-84, 1999.

[ZSS+14] Zheng, M., Sanán, D., Sun, J., Liu, Y., Dong, J. S., Gu, Y. *State Space Reduction for Sensor Networks Using Two-Level Partial Order Reduction* VMCAI, pages 515-535, 2013.